ADAM & EVE
An Abstract Dataflow Machine and Its Programming Language

Diplomarbeit

zur Erlangung des akademischen Grades
"Diplom-Ingenieur"
in der Studienrichtung
Informatik

eingereicht von

Wolfgang Schreiner

Angefertigt am
Institut für Mathematik
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz

Eingereicht bei
o.Univ.-Prof. Dr. Bruno Buchberger

September 1990

Abstract

This thesis describes the design and the implementation of the ADAM Abstract Dataflow Machine and the functional programming language EVE that will be used for programming this machine. ADAM is an abstract parallel computer with a hybrid distributed memory architecture that combines features from register machines and from dataflow computers. Its processor has a rather conventional set of two-address instructions augmented by primitives for the creation and synchronization of parallel tasks that allow the simulation of dataflow behaviour. Special memory units on each ADAM module are responsible for the management of non-strict data structures. EVE is a simple purely functional language that shows most of the features known in functional programming: strong typing, higher order functions, non-strict functions and data types. A prototype of the ADAM concept has been implemented on a multi-transputer system and shows very promising results.

Keywords: *Parallel Computation, Dataflow Architectures, Functional Programming.*

Für meine Eltern

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I want to thank *Prof. Bruno Buchberger* whose untiring efforts in organizing the institute, raising funds and setting up academic and industrial contacts have been the basis for the development and the work of the parallel computation group at RISC-Linz. His activities to promote computer science research in Austria, especially his initiation of the "Austrian Center for Parallel Computation", are a valuable contribution to the future development of our country. I am grateful for his confidence in me and hope that my work will meet this attitude.

Moreover, I want to thank all the members of the parallel computation laboratory at RISC-Linz whose engagement and enthusiasm is the backbone of a lot of valuable and interesting work: *Klaus Kusche*, who has organized the lab for almost two years now and whose idea initiated my thesis, gave me much valuable help for my work and proof-read several parts of this paper; with *Kurt Siegl*, I have had many long and interesting discussions on parallel programming and parallel languages; *Peter Zettler's* work on LINDA is certainly going to become very important for us; *Hans-Wolfgang Loidl* will continue my work by his diploma thesis on the design and implementation of a compiler for the EVE language.

The course on the semantics of parallel programming languages held by *Gerhard Funk* in the summer term of 1990 gave me many interesting insights into the connection between different parallel programming paradigms. I want to express my thanks to all participants of this course with whom we organized after each official lecture a private one, in particular (additionally to the members of the parallel group) to *Werner Landerl*. We would be very happy if he also joined our group.

Furthermore, I want to acknowledge the support of all other students, assistants and faculty at RISC-Linz. It is really a great feeling to work with a team that has so much background in so many varying fields of computer science and mathematics ranging from theoretical foundations to details of software-engineering. I also want to mention the hard work done by our systems manager *Georg Gratzl* and his assistant *Hubert Hofbauer* in order to keep the computing environment of RISC-Linz always running and up to date. I am also grateful for the patience of our secretary *Maria Wögerbauer* when she faced the long periods that I occupied the laser printer with this thesis.

Chapter 1

Introduction

Parallel Computation is today one of the most advanced and most exciting topics in computer science. While in the past the performance of computers with single processors could be steadily enhanced year for year, in the future we can expect essential increases of computing power only by parallel architectures. Today many research institutions, from both the academic and the commercial world, work on the design of parallel architectures, languages, algorithms and tools. A lot of applications that require high computing performance have already been parallelized. This ongoing trend towards parallel computation will strongly intensify in the future.

Research on parallel computation has already a long tradition at RISC-Linz, especially the research on **languages and algorithms for parallel symbolic computation**. This research has its roots in the seventies (see [Buchberger, 1978]) and resulted in the concept of the parallel L-Machine and the associated L-Language (see [Buchberger, 1985]). An experimental prototype of this machine and the language has been operative for several years now.

One of the latest highlights in the institute's research has been the design of the **new L-Language LL**, a parallel general purpose programming language designed to meet the needs of symbolic computation (see [Kusche, 1988]). For reasons of optimum hardware usage and highest possible efficiency, the new L-Language was decided to rely on **explicit coarse-grain parallelism**. The programmer himself has to specify the parallelism on the level of whole functions or even larger parts of the program. A verification system that allows to prove the correctness of LL programs has been developed in [Funk, 1989].

In contrast to this approach, there has also been much international research on how to exploit the **implicit fine-grain parallelism** inherent in most (if not all) algorithms. Several hardware architectures have been developed in order to utilize this kind of parallelism in a very efficient way, where **dataflow machines** seem to be one of the most promising approaches (see [Arvind and Iannucci, 1987]). Especially for **functional languages** which provide a very large amount of implicit parallelism these kinds of architectures are of great interest (see [Arvind *et al.*, 1988a]). However, also for conventional machines the techniques used for exploiting implicit parallelism (e.g. dataflow analysis) are very important, since they allow the **automatic parallelization** of basically sequential programs (see e.g. [Boyle, 1989]).

The parallel computation group at RISC-Linz has therefore started to investigate the usefulness of the dataflow approach for applications in symbolic computation: The goal of these investigations has been the design and implementation of an **Abstract Dataflow Machine** (short ADAM) programmed in the functional language EVE and simulating the architecture of a dataflow computer on a conventional parallel machine. The kernel of the ADAM architecture has been subject of substantial changes in the course of the design process:

- The original idea initiating these investigations was to simulate the architecture of some concrete **pure dataflow architecture** such as the British MANCHESTER DATAFLOW MACHINE ([Gurd *et al.*, 1985]), the Japanese SIGMA-1 ([Yuba *et al.*, 1985]), the TTDA ([Arvind and Nikhil, 1988]) or the ETS MONSOON ([Papadopoulos, 1988]), both developed at the MIT. Based on the assumption that conventional von Neumann architectures are not very suitable for massively parallel computing, the goal of all of these approaches is the development of new microprocessor architectures that efficiently support fine-grain parallelism, have low communication overhead and tolerate long memory latencies.

- However, one of the preliminary goals of our investigations is actually not to develop new hardware architectures but to get some experience how far the dataflow approach is suited for applications in symbolic computation. In order to achieve this goal, it is not really necessary to simulate some concrete dataflow machine but only to simulate *dataflow behaviour*. Since we intended to implement the ADAM architecture on a conventional parallel transputer system ([Inmos, 1988b]), the idea arised to investigate how a conventional processor architecture could be enabled to make use of the dataflow model. The result of these investigations is a **hybrid dataflow architecture** that basically consists of an conventional instruction set extended by special *parallelization and synchronization instructions.*

The term "Abstract Dataflow Machine" is therefore somewhat misleading. The ADAM architecture does actually not represent a dataflow machine but a machine that is able (but not at all forced) to simulate dataflow behaviour. For our purposes, this approach has in particular the following advantages:

- The ADAM architecture is neither fixed to any parallelization nor to any synchronization strategy. In fact, only the *compiler* determines where to generate parallelism and how to synchronize parallel program paths. We may therefore experiment with various parallelization strategies simply by replacing one code generator of the compiler by another one.

- Utilizing the ADAM's more or less conventional register structure, the compiler may make use of the *traditional compilation techniques* for register machines. This may help to decrease the time and effort spent on writing a code generator for the compiler.

- In many cases, the compiler may *generate more efficient code* by optimizing register utilization, removing superfluous synchronization instructions, serializing parallel program paths, etc. By avoiding the "finest-grain parallelism" that is actually not utilized by any architecture, parallelization overhead may be minimized.

Independently on these considerations, the idea of an **hybrid dataflow/von Neumann architecture** has recently become a promising research area that finds much international interest ([Iannucci, 1988], [Gao *et al.*, 1988]).

However, the ADAM architecture, as defined in this thesis, shows also some restrictions mainly imposed by implementation considerations:

- In contrast to the original intention described in [Schreiner, 1989], the ADAM architecture does not provide any support for *delayed evaluation*. Further considerations revealed that this feature would essentially decrease efficiency and raise a lot of non-trivial problems. Since for our purposes delayed evaluation is certainly a convenient but actually not an essential feature, we decided to omit it in the first approach.

- As defined in this paper, all compound data structures consist of *cells of fixed size*, i.e. variable-size tuples are mapped to linked lists on the heap. However, the only reason for omitting variable-size storage blocks is to simplify memory management. Since all memory instructions consider the heap as a set of variable-size tuples, it is no problem to replace the actual implementation by a more efficient one while preserving code compatibility.

The ADAM architecture is part of a research project and still subject to changes, corrections and improvements. Its substantial kernel has now been stabilized but some features (e.g. the instruction set) may still change. The ADAM concept has been implemented on a multi-transputer system and shows rather promising results.

While our motivation for our activities of the ADAM architecture should be rather clear, we will now explain our reasons for the design of another programming language (named EVE) that will be used for programming the ADAM machine:

The extensive research on functional (respectively dataflow) programming has already resulted in a number of well-known functional languages (MIRANDA ([Turner, 1985]), ML ([Harper *et al.*, 1988]), ID ([Nikhil, 1988]), HASKELL ([Hudak *et al.*, 1990]), ...) that are very similiar in their essential features. This shows that the functional programming community has reached a well-founded consensus on the state of the art. It is not our aim to enrich this variety of languages by a new language with extraordinary new facilities. In fact, any of the languages mentioned above would do the job of programming the ADAM at least as well as the language that we are going to propose. However, these languages have obtained a complexity that makes a rapid implementention of an experimental prototype (of even a language subset) rather difficult.

Since the prevailing purpose of our investigations is the research on dataflow machines and, in the frame of this project, not the research on compiler construction, we have decided to design the small functional language EVE[1] providing the main properties of the languages mentioned above while omitting those features which are not really essential but rather complicate an experimental implementation. The main intentions that we pursue with the construction of this language are as follows:

1. We want to investigate the implementation of functional languages on parallel computers using dataflow principles.
2. We want to experiment with symbolic algorithms programmed in a functional language and executed on parallel architectures.

EVE is small and simple, but it is certainly not a toy language. In particular, the language shows the following **features**:

- EVE is **purely functional** i.e. the evaluation of programs is free of any kinds of side-effects.
- EVE is **non-strict** i.e. functions may return results even if some of their arguments have not yet been evaluated.
- EVE is **strongly typed** i.e. typing errors are detected at compile time.
- EVE is **higher order** i.e. functions may take other functions as arguments and return functions as their results.
- EVE is **safe** i.e. runtime errors are caught by error values and do not lead in an unpredictable break of the program.

[1] EVE is a condensation of the term "Evaluated Expressions", which is a hint at the functional character of the language.

In order to keep the language small, the following features have been explicitly omitted, either because they just represent less important **syntactic sugar** or because they make an experimental implementation essentially more complex:

- Pattern Matching,
- Type Inference and Polymorphism,
- Lambda Expressions and Curried Functions,
- List Comprehensions,
- Delayed Evaluation,
- Abstract Data Types and Modules.

The goal of this thesis is to give both an intuitive and exact description of the ADAM architecture and the EVE language and to describe the current prototype implementation of the ADAM. Except for a general outlook, the thesis does *not* treat the design and implementation of a compiler translating the EVE language into ADAM machine code. This issue is covered by another diploma thesis that is currently under way.

The **structure of the thesis** is now as follows:

- After this introduction, the chapter **Dataflow Programming** presents a general overview on the scope of the thesis. It describes the functional style of programming, illustrates the dataflow model with the help of a few parallel programs and summarizes the current state of the art in the international research on dataflow architectures.
- The chapter **The EVE Programming Language** contains the definition of the functional language EVE. It formally defines the syntax of the language and informally describes its semantics on the basis of several examples. A major program written in EVE shows the capabilities of the language.
- The chapter **The ADAM Abstract Dataflow Machine** provides an informal but detailled description of the ADAM architecture. First the basic idea of the concept is illustrated, then the construction of the machine and its various components are presented in a top-down fashion.
- In the chapter **Compiling the EVE Language into ADAM Code**, we give an overview in which way a planned EVE compiler for the ADAM machine will be constructed. Moreover, we show how several constructs of a functional language can be translated into code executable on the ADAM architecture.
- The last chapter on **The ADAM Implementation and Evaluation** describes the current status of the system's implementation and presents several of its internal details. A few benchmark programs illustrate the performance of the system and the speedups that could be achieved.
- Appendix **The ADAM Operational Behaviour** formally defines the ADAM architecture by an abstract interpreter: this interpreter is presented by a set of algorithms that describe the behaviour of each component of an ADAM worker module.
- Appendix **The ADAM Instruction Set** formally defines the machine instructions of the ADAM architecture: for each instruction, an algorithm is presented that describes the effect of the instruction in the frame of the interpreter given in the previous appendix.

- Finally, Appendix **The ADAM Source Code** contains the whole Occam 2 source code of a prototype implementation of the Adam concept on a multi-transputer system.

Any person interested in this or other projects of the parallel computation group of RISC-Linz is invited to contact the author (Bitnet address: `K315640@AEARN`). We gratefully appreciate all kinds of comments, suggestions and criticism.

Chapter 2

Dataflow Programming

In this chapter, we present an introduction to the dataflow style of parallel programming. After a motivation for our interest in this topic, a sketch of the basic principles of functional programming is given, then the dataflow model is described. A major example of a dataflow program illustrates the essential features of this model. Finally, an overview of several architectures implementing the dataflow principle in hardware is presented.

2.1 Motivation

Writing correct programs for parallel computers is one of the most difficult tasks in computer science. Up to now, progress in parallel computation has been significantly hindered by the fact that parallel programming has been an extremely tedious task. There arise problems that are essentially more difficult than on sequential machines or totally unknown there:

- **Complexity:**

  In sequential programming, only one execution sequence of statements has to be kept in mind. At any moment, there is only one program state to watch and there is only a limited number of possible influences on this state. In the parallel case, the programmer has to keep track eventually of a large number of parallel paths that interfere with each other. Each of the parallel tasks has a local state and the state of each task may often be altered by any other task. Temporal relationships between the different tasks have to be considered and the correct interaction between parallel processes has to be handled. Altogether, the dynamical behaviour of parallel programs can be orders of magnitude more complex than that of sequential ones.

- **Observability:**

  In distributed memory systems, parallel tasks may reside on processors for that no input from and no output to peripheral devices is possible. Debugging erroneous processes on such processors may become an extremely difficult task. This is in particular the case if the available debugging tools are poor (as usual) and do not allow to inspect the state of the desired tasks. Even for parallel processes on a single processor, it may not be possible to watch the state of any process. In this case, the programmer is forced to draw ingenious conclusions by reasoning about the possible sources of error from their effect on other (observable) processes. Moreover, using debugging tools may essentially alter the behaviour of the program investigated and let the error suddenly disappear (see below).

- **Non-Determinism:**

  From one execution of a program to the next, the temporal relationship between parallel processes may (and usually will) essentially alter. Errors that once have occurred may not be able to be reproduced. The program may return different answers on different program runs or may sometimes run into an deadlock. In general, it is extremely difficult (say impossible) to reconstruct the conditions that lead to some particular behaviour of a parallel program. The use of debugging or trace tools has very often the effect that an error that occurred a couple of times cannot be observed any more.

Altogether, writing a parallel program in general represents a great mental challenge for a human being. One main reason for this fact is the lack of programming tools, systems and, in particular, languages that make parallel programming a more natural task. From the software-technological point of view, the state of parallel programming can be compared with the state of computer science in the early 1960's with the first high-level languages and compilers coming into existence.

Today, most parallel programming paradigms still rely on the familiar imperative languages based on sequential computers and the von-Neumann principle of computation. These languages are usually extended by several constructs that allow the creation of parallel tasks, their synchronization and the communication between them, e.g. by

- explicit syntactic constructs derived from the CSP model ([Hoare, 1985]),
- operating system functions from UNIX or MACH ([Tevanian Jr. *et al.*, 1987]),
- the LINDA programming model ([Bjornson *et al.*, 1987]).

Even if some of these programming models (in particular LINDA) are rather high-level and seem to provide sufficient support, the situation is not basically changed. Furthermore, there always remains one point of criticism: the programmer still **thinks in terms of sequential programming**. He basically uses sequential languages to write sequential programs and finally just combines several sequential programs to let them work in parallel.

One radically different approach to this situation is to leave the imperative programming model that is essentially based on sequential von-Neumann computers. Instead of this, the programmer should use a model where **true parallel programs** are written, i.e. without specifying any sequential paths of execution. Such a programming model can be only based on **declarative programming languages**.

Declarative programming languages allow to specify the relation between the given input arguments and the desired output values without defining sequential procedures that compute these values. Languages of this kind are usually based on one of two mathematical concepts, either on **mathematical functions** or **logical relations**. Hence, we can distinguish between

- Functional Programming Languages and
- Logic Programming Languages.

Since the programmer does not specify a sequential program, the implementation of such a language (consisting of compiler and runtime system) may freely choose the way the program is executed, in particular it may let the program be executed in parallel. This kind of execution is especially supported by the fact, that declarative programs are **inherently parallel** down to the lowest level. Roughly spoken, any textually independent parts of such a program may be executed in parallel.

Several strategies have been developed for the implementation of declarative programming languages on sequential as well as on parallel computers. One of these strategies is the **dataflow principle**. This principle is a model for the way a declarative (in particular functional) program may be executed in parallel:

- The dataflow model suggests a particular direction for the parallel implementation of a declarative language. This implementation may usually be based on a conventional already available parallel architecture. However, for almost two decades, several special **dataflow computers** have been proposed that promise the most efficient utilization of the fine-grained parallelism provided by declarative languages.

  The goal of these architectures is to implement the dataflow model directly in hardware, i.e. dataflow processors are no more based on the von-Neumann principle of program execution. While, theoretically, these architectures have a lot of advantages that make them superior to conventional parallel computers, commercial available systems have not yet come into existence. One main reason for this disappointing fact is the large amount of knowledge gathered on the von-Neumann principle of computation, as well from the hardware engineer's as from the software engineer's point of view.

- The long-term research on dataflow architectures lead to the wide-spread view that the dataflow model is basically a model for the the construction of parallel architectures and that the dataflow principle might be only implemented on special hardware. However, this is a misunderstanding of facts, since this model is **independent of the underlying hardware** and may also be implemented on conventional architectures.

  Moreover, the dataflow model suggests a **particular style of programming** regardless of the architecture it is based on. The dataflow style of programming provides several powerful features that are as useful as elegant and have no counterpart in classical (sequential or parallel) programming in imperative programming languages.

In the following sections, we will present the dataflow model using the functional style of programming. Logic programming languages, while basically more powerful than functional languages, have several properties that make their parallelization more difficult. Actually, most parallel implementations of logic languages abandon several important features from logic programming (such as backtracking or unification), which makes them very similiar to functional languages. However, in [Siegl, 1990a] and [Siegl, 1990b], the power of a parallel logic language using dataflow principles is presented on the basis of a concrete application.

2.2 Functional Programming Languages

The basic principles of functional (also called applicative) languages go back to the early days of computer science. One of the first high-level programming languages, LISP ([McCarthy, 1960]), already incorporates several of the essential features of functional programming[1]. Actually, the theoretical foundations of functional programming, summarized in the theory of λ-calculus ([Church, 1941]), have been established before the first computers have yet come into existence.

Most of the modern functional programming languages have their roots in [Landin, 1966] and the famous Turing Award lecture [Backus, 1978]. Today, there exist a variety of functional languages. Among them, the most popular ones are

[1] However, since a lot of imperative extensions have been included to Lisp, only a small subset of the language (Pure Lisp) can be called functional.

- ML ([Cousineau *et al.*, 1985]),
- HOPE ([Burstall *et al.*, 1980]),
- MIRANDA ([Turner, 1985]),
- ID ([Nikhil, 1988]),
- HASKELL ([Hudak *et al.*, 1990]).

However, the differences between most of these languages are merely of syntactic kind. Today, a wide consensus has been established on the essential features of functional languages and most research efforts now concentrate on compilation techniques for their efficient implementation ([Peyton Jones, 1987], [Johnsson, 1987], [Augustsson, 1987]).

For the following discussion, we will use a pseudo-functional language similiar to the syntax of MIRANDA. The examples presented are partially taken from [Bird and Wadler, 1988] that provides an excellent introduction to functional programming.

2.2.1 Basic Principles

A language is called **functional** if a program written in this language is composed of functions that are free of side-effects. If a function is evaluated, it returns a value but has no effect besides that. In particular, it does not interfere with the evaluation of other functions. Hence, the result of a function only depends on the values of its arguments and of globally defined constants but not on any other influence.

As a consequence of this feature, called **Referential Transparency**, all functional programs have the **Church-Rosser Property**. This mathematical property says that, for obtaining the result value of a function, its arguments may be evaluated in any order, in particular **in parallel**. Any terminating evaluation order determines the same result value (see Section 2.2.3).

However, this useful feature imposes the following restrictions on a functional language:

- A functional language has **no variables** (in the meaning of storage locations containing variable values) but only named constants whose values are fixed in their scopes. Hence, there is **no assignment statement** that may alter the value of a variable. Morover, there are no mutable data structures whose components can be individually altered. On the creation of a data structure, all components have to be explicitly defined.
- Since every kind of communication with the outside world is a side-effect, functional languages has **no explicit input/output statements**. Of course, every system implementing a functional language has to communicate with the human user but this is the task of a dialogue shell and not part of the language itself.

E.g. let us consider the following functional program:

$$\begin{array}{rcl} f\ x\ y & = & x + a,\ \textbf{if}\ x < 10 \\ & = & x - a,\ \textbf{otherwise} \\ & & \textbf{where}\ \ a = square\ (y+1) \\ & & \qquad\quad square\ x = x * x \end{array}$$

If the function f is called by $f(5,7)$, the formal parameters x and y are bound to the actual arguments 5 and 7, respectively. The result of f is defined by two equations each guarded by a test condition. Since $5 < 10$, the result of the function is determined by the right hand side $5 + a$ of the first equation. a is defined in the where-clause of the definition as $square(7 + 1)$. Since *square* is in the same clause defined to be the quadratic function, a is bound to 64 and the total result of the function is 69.

This program has computed a rather simple result. Another more complex program involves recursive function definitions and the use of linear lists:

$$\begin{array}{lcl} sumprod\ x\ y & = & 0,\ \textbf{if}\ null\ x\ \textbf{and}\ null\ y \\ & = & head\ x\ +\ sumprod\ (tail\ x)\ y,\ \textbf{if}\ null\ y \\ & = & head\ y\ +\ sumprod\ x\ (tail\ y),\ \textbf{if}\ null\ x \\ & = & head\ x\ *\ head\ y\ +\ sumprod\ (tail\ x)\ (tail\ y),\ \textbf{otherwise} \end{array}$$

This program computes the sum of all products of the corresponding entries of two lists. If one list is shorter than the other one, the remaining elements of the other list are simply added. In *sumprod*, four equations handle the cases that may appear. *null* is a primitive predicate testing whether a list is empty, *head* and *tail* return the first element of a list and the list without the first element, respectively.

However, the same program may be written in a more elegant way:

$$\begin{array}{lcl} sumprod\ [\,]\ \ [\,] & = & 0 \\ sumprod\ a:x\ \ [\,] & = & a + sumprod\ x\ [\,] \\ sumprod\ [\,]\ \ b:y & = & b + sumprod\ y \\ sumprod\ a:x\ \ b:y & = & a * b + sumprod\ x\ y \end{array}$$

This program makes use of a feature called **pattern matching** that allows to select the appropriate equation of a function by matching the actual arguments against the patterns specified in the head of a function definition. E.g. the pattern $a : x$ (with : being the *cons*-operator) matches against any list with at least one element. If the match succeeds, the first element of the list is bound to a and the rest of the list is bound to x. Using this feature, programs may become essentially more compact.

The next functions shows how the handling of lists may be simplified:

$$\begin{array}{lcl} listsum\ x\ y & = & [a + b \mid (a, b) \leftarrow (x, y)] \\ divisors\ x & = & [y \mid y \leftarrow [1 \ldots x],\ x\ mod\ y = 0] \\ triads\ n & = & [(x, y, z) \mid x \leftarrow [1 \ldots n],\ y \leftarrow [x \ldots n],\ z \leftarrow [y \ldots n],\ x^2 + y^2 = z^2] \end{array}$$

All three functions make use of the **list comprehension** facility that is derived from the set quantifier of mathematics. The result of a list comprehension $[e_x|x \leftarrow l, p_x]$ is the list of all expressions e with the free variable x successively bound to all values of the list l that satisfy condition p. If more qualifiers (i.e. binding statements or conditions) on the right hand side of the $|$ occur, they are all evaluated from left to right and only those bindings that have passed all conditions are considered for the result list.

listsum computes a list whose entries are the sums of the corresponding elements of two input lists. *divisors* computes all divisors of a given integer x. *triads* constructs the list of all Pythagorean triads in a given range, i.e. triples of numbers x, y, z such that $x^2 + y^2 = z^z$.

Finally, a last example shows that functions themselves may be handled like any other kinds of objects:

$$\begin{array}{lcl} \mathit{oddlist}\ n & = & \mathit{filter}\ \mathit{odd}\ [1 \ldots n] \\ \mathit{filter}\ p\ l & = & [x \mid x \leftarrow l,\ p\ x] \\ \mathit{odd}\ n & = & (n\ \mathit{mod}\ 2 \neq 0) \end{array}$$

filter is a **higher-order function** taking a list l and a predicate (i.e. a boolean valued function) p as arguments and returning a list that contains all elements of l that fulfill p. *oddlist* constructs the list of all odd integers in a certain range by calling *filter* with the test function *odd*. In general, the use of functions as first-class objects is a very powerfull tool that allows to write concise and efficient programs.

Due to lack of space, we cannot describe a lot of other advanced and very important features of functional programming languages, in particular their sophisticated type systems that have no counterpart in imperative languages. Interested readers may refer to [Bird and Wadler, 1988].

2.2.2 Strictness Properties

Up to now, we have considered functional programs as mere static objects and neglected their dynamic interpretation. In particular, we have not yet discussed the evaluation strategies applicable to functional programs. However, this strategy may have a crucial influence on the dynamic behavior of a program. Consider e.g. the following function definitions:

$$\begin{array}{lcl} \mathit{three}\ x & = & 3 \\ \mathit{undef}\ x & = & \mathit{undef}\ (x+1) \end{array}$$

Suppose we want to evaluate the expression $\mathit{three}(\mathit{undef}(7))$. What will the result of the evaluation be? From the mathematical point of view, the answer is clearly 3, since this is the result that *three* returns regardless of the argument given. However, if we apply the *call by value* evaluation strategy that is used in most imperative languages, the answer is that the evaluation of the expression does not terminate, since the evaluation of the argument does not. On the other hand, if we just *textually* replace every occurrence of the formal parameter x in the body of *three* by the argument $\mathit{undef}(7)$ and then evaluate the body (*call by name*), the result will be 3, as desired.

Hence, there are some evaluation strategies that are complete in the sense that they may return a proper result of a function, even if (some or all of) its arguments are not defined, while other strategies are not. Evaluation strategies that are complete in this sense are called *non-strict* and languages that demand such a strategy are called *non-strict languages*.

The strictness properties of a language may have strong impacts on the runtime behaviour of a program, in particular on its termination. However, also a program's reaction on runtime errors is influenced by its strictness properties, since the computation of e.g. *three* $(1/0)$ may either return 3 or trigger a runtime error.

We will now define the concept of strictness more formally. Let the special symbol $\perp$ (in words *bottom*) serve for two purposes:

1. $\perp$ may denote a non-terminating value, i.e. an expression, whose result value will never be computed.
2. $\perp$ may also denote the error value, i.e. the result of the illegal application of an operator.

We say that an n-ary function *fun* is **strict in** arg_i $(1 \leq i \leq n)$, if the following property holds for all arguments arg_j $(1 \leq j \neq i \leq n)$:

$$fun[arg_1, \ldots, arg_{i-1}, \perp, arg_{i+1}, \ldots, arg_n] \; = \perp .$$

Otherwise we call *fun* **non-strict in** arg_i.

In other words, a function *fun* is strict in an argument arg_i, if

- whenever the evaluation of arg_i does not terminate and return a result, fun also does not terminate and return a result, and
- whenever the evaluation of arg_i results in the error value, fun also results in the error value.

A function is called non-strict, if it is non-strict in any of its arguments. An evaluation strategy is non-strict, if it allows non-strict functions. A language is non-strict, if it demands a non-strict evaluation strategy.

Non-strict languages are preferable since they make programming concepts possible that are not known in strict programming languages. Consider e.g. the following function definitions for prime number computation:

$$\begin{array}{lcl}
prime\ n & = & primes\ !\ n \\
 & & \\
primes & = & 2 : [p \mid p \leftarrow [3, 5, \ldots],\ isprime\ p\ primes] \\
 & & \\
isprime\ t\ p : r & = & \text{true},\ \textbf{if}\ p * p > t \\
 & = & \text{false},\ \textbf{if}\ t\ mod\ p = 0 \\
 & = & isprime\ t\ r,\ \textbf{otherwise}
\end{array}$$

prime n returns the n^{th} prime number by taking the n^{th} element of the infinite list of all primes! *primes* itself is constructed recursively as a sequence starting with 2 and containing all odd numbers p that can not be divided by any element of *primes* up to the square root of p.

The "infinite" list *primes* can be constructed because list construction is non-strict in both of its arguments, i.e. the first element and the rest of the newly constructed list. Hence, this operation may return its result, a cons-cell, even if neither the head nor the tail have yet been evaluated. Only if the value of the head or some element of the tail have to be known, the corresponding arguments have to be evaluated resulting in a list element and another cons-cell, respectively.

Non-strict programming languages are more difficult to implement than strict ones and require generally more computational overhead. In sequential systems, non-strict languages are usually implemented using the **delayed evaluation** (also called **call by need**) strategy that only evaluates an expression if its value is required. However, the terms *non-strict* and *delayed evaluation* should not be mixed, since the one defines a logical concept while the other is just one particular implementation method.

On parallel architectures, non-strictness may be achieved by creating one parallel process for each argument of a function. The function is immediately activated, i.e. *before* any argument values have become available. Only if the value of some argument is required for further computation, the function has to wait for the result of the parallel process and synchronization takes place. This is the way non-strict functions are implemented in the dataflow model that is described in the next section.

2.2.3 Parallel Functional Programming

As mentioned in the previous sections, all functional languages have the Church-Rosser property stating that any terminating evaluation order of a functional program yields the same result. Hence, all arguments of a function may be simultaneously evaluated by parallel processes.

There are a lot of advantages that functional languages provide for parallel execution:

- **Parallelization**

  Functional languages are **implicitly parallel** down to the lowest level. Any implementation of a functional language may utilize different granularities of parallelism, ranging from fine-grained parallelism (on the level of primitive operators) up to coarse-grained parallelism (on the level of whole program blocks). Hence, the automatic parallelization of a functional program is much easier than for a corresponding imperative program. No complicated analysis techniques are required for detecting data dependences, side effects, ...

- **Determinism**

  All functional programs are **fully deterministic**. They always return the same result, if they are given the same input. Runtime-dependent errors derived from special temporal relationships between parallel processes (e.g. deadlocks) cannot occur.

- **Portability**

  Functional programs are **independent of the underlying architecture**, the type and the number of processors, and their configuration. Changes in the hardware do not at all affect the source code of a functional program. Every program that correctly works on some kind of computer architecture will also do so on any other one. In particular, the program may be developed and debugged on a sequential machine using a sequential evaluation strategy and then, without change, ported to a parallel architecture.

Several models for the parallel execution of functional programs have been developed. They usually fall into the categories

- Graph Reduction Models and
- Dataflow Models.

Both of them have gained much attention in the last years. An excellent introduction into the graph reduction model and its application for the implementation of functional programming languages is presented in [Peyton Jones, 1987]. In this thesis, we will concentrate on the dataflow approach as described in the following section.

2.3 The Dataflow Model

2.3.1 Basic Idea

The dataflow principle is one particular model describing the way a functional program may be executed on a parallel machine. The basic idea of the model is as follows:

Take any functional program, e.g.

Figure 2.1: A Dataflow Graph

$$
\begin{aligned}
f\ x\ y \quad &= \quad (a + b) * (-y) \\
&\quad\quad \textbf{where} \;\; a = -x \\
&\quad\quad \phantom{\textbf{where}} \;\; b = x/y
\end{aligned}
$$

The body of a function is always a (mathematical) expression. This expression can be represented by a directed acyclic graph as illustrated in Figure 2.1. The nodes of the graph represent the operations to be performed and its arcs denote the data dependences between these operations. An arc from operation a to operation b indicates that the result value of a is required by b for further computation. Arcs can be named where the names represent the constant names of the functional program. Arcs without predecessors represent the input arguments of the function and the one arc without successor represents its result value.

The dataflow model now gives such a static graph representation a dynamic interpretation. The idea is that so called *tokens* transmit values between the nodes by flowing along the arcs of the graph. Each token is a piece of information constructed of two fields

tag	*value*

where *tag* indicates the instruction the token is on the way to and *value* represents the transmitted value.

As soon as on all input arcs of an operator tokens have arrived, the operator may *fire*: it removes all input tokens, performs the desired operation on the token values and generates a result token on its output arc carrying the result of the operation.

The order in which the operations within a graph fire is undefined. Every operator may fire as soon as tokens have arrived at all of its inputs. Operators may fire simultaneously and the execution time may vary for each operation. A possible sequence of firings for the above expression is illustrated in Figure 2.2.

Figure 2.2: A Firing Sequence

2.3.2 Translating Programs to Dataflow Graphs

In principle, a whole program consisting of a collection of functions may be translated in a single dataflow graph by embedding the body of each function

- with one input arc for each formal parameter and
- with one output arc for the computed result value

in the graph.

However, only those arcs connecting the instructions within a function are fixed at compile time, we call them *static arcs*. The connections between different functions depend on the given input data and can be established only at run-time, we call them *dynamic arcs*. Moreover, more than one activation of a function may (by recursion and parallel execution) simultaneously exist. Hence, we do not represent a whole functional program by a single dataflow graph but by a **wood of graphs** each of which models one function. The following subsections now discuss the questions:

1. How can several simultaneously existing activations of a function be managed in the same dataflow graph?
2. How can a function call in a dataflow graph be connected to the graph of the called function?

Simultaneous Function Activation

For each function, there exists one particular dataflow graph. However, there may exist more than one activations of the function at the same time. Hence, we need some possibility to distinguish the tokens that flow simultaneously through the same graph but belong to different activations of the function. Thus, to every token an additional *context* field is attached that indicates the activation that the token belongs to:

$cont.addr_p$	$value$

The token tag now consists of three components: con represents the context of the function, $addr$ denotes the address of the instruction that the token is on the way to and p indicates the (left or right) input port in the case of binary instructions.

Since tokens that belong to different activations of a function may simultaneously flow through the graph, the firing rule presented in the previous section has to be adjusted:

- A token may fire, if a matching pair of input tokens has arrived, i.e. on all input ports of the operation there exist tokens with the same context field.
- If an operator fires, the result value of the operation is combined with a token that has the same context field as the input tokens.

By this rule, tokens from different activations can no more interfere with each other. However, now on each input arc of a binary operator an arbitrary number of tokens may wait for their matching partners.

Embedding Function Calls in a Dataflow Graph

In order to combine a function call with the corresponding dataflow graph, the following activities are neccessary:

- A context has to be created for the new function activation. This context has to be unique (i.e. must not yet have occurred) for this particular function.
- The caller must send to the activation defined by the new context
  - the argument tokens,
  - the description of its own context and
  - the address of the instruction waiting for the result of the function.
- The callee must return the result token back to the caller defined by the context and the instruction address submitted.

For the implementation of *non-strict* functions, it is important that the activation of a function is independent of the submission of the arguments. The called function must be able to execute and return a result even if none of the arguments have arrived yet.

2.4 Dataflow Programs

Having sketched functional programming languages and the dataflow model, we now want to illustrate the flavour of the dataflow style of parallel programming by means of a major example.

First we want to argue why we think that dataflow programming is different from ordinary functional (or in general declarative) programming:

Theoretically, a "functional programmer" does not have to know anything about details of the underlying hardware and the evaluation strategy used. The only important knowledge is whether the language supports non-strict functions and which of the predefined functions are non-strict and which are not. Hence, dataflow programming should not be distinguishable from ordinary functional programming. Obviously, this is true from the correctness point of view, i.e. any functional program executed using one evaluation strategy remains correct (modulo strictness) when executed by the use of another strategy.

However, dataflow programs, i.e. functional programs using dataflow principles, are generally written in a different style than sequentially executed functional programs. The knowledge about dataflow evaluation strategies helps to avoid synchronization bottlenecks and to enhance the utilization of parallelism inherent in an algorithm. Moreover, many functional programs efficiently working on sequential architectures will not show significant speedups when implemented on parallel architectures, if basic dataflow principles are not considered. In other words, a program is not parallel just because it is functional.

What is now different with dataflow programming? In a nutshell, we can say

> **Dataflow programming is declarative programming with the minimization of data dependences.**

Minimizing data dependences actually meens delaying synchronization to the latest possible moment. One important point of synchronization in functional languages is the conditional construct (also when hidden in pattern matching): since the boolean test condition has to be evaluated before it can be decided which branch to take, the conditional construct is *strict in the test condition*. In dataflow programming, the user should therefore try to delay boolean tests as long as possible.

The following sections present a major example that illustrates these principles.

2.4.1 Prime Number Generation

In this section, we want to design a dataflow program that creates all prime numbers up to some given bound. For this program, we make use of the following idea: in order to decide whether some integer i is prime, it suffices to divide i by all primes up to $\sqrt{i}$. Vice versa, if all primes up to some bound n have already been computed, the integers between n and n^2 may be tested in parallel.

In a functional program, this idea might be implemented as follows. We write a function

$$sieve\ p\ t\ n$$

where the formal parameters are interpreted in the following way:

- p is the ascendend list of all odd primes that have already been computed.
- t is some odd integer in the range m to m^2 where m is the last element (i.e. the maximum) of p.
- n is the upper bound for the primes to be computed. If $t > n$, p is already the result of the whole program.

With the help of *sieve*, the desired result of the program may be formulated as follows:

$$\begin{array}{rcl} primes\ n & = & [],\ \mathbf{if}\ n < 2 \\ & = & [2],\ \mathbf{if}\ n = 2 \\ & = & 2 : sieve\ [3]\ 5\ n,\ \mathbf{otherwise} \end{array}$$

The first two equations handle trivial cases. The third equation contains the call of sieve where [3] is the list of already computed primes and 5 is the first odd integer to be tested (we do not take [ ] and 3 for this purpose, since our choice makes the test conditions in *sieve* simpler, see below).

An obvious way to implement *sieve* would be as follows:

$$\begin{array}{rcl} sieve\ p\ t\ n & = & p,\ \mathbf{if}\ t > n \\ & = & sieve\ p'\ (t+2)\ n,\ \mathbf{if}\ isprime\ p\ t \\ & = & sieve\ p\ (t+2)\ n,\ \mathbf{otherwise} \\ & & \mathbf{where}\ p' = append\ p\ t \end{array}$$

sieve first tests whether $t > n$, i.e. p is already the desired result. If not, t is tested with respect to p. Depending on the result of this test, *sieve* is recursively called either with the old list p or with t appended to p.

However, this solution has little inherent parallelism. Since the conditional construct is strict in the boolean test conditon, *first* t is tested and *after* the result of this test, one of the branches is taken and *sieve* is recursively called.

$$
\begin{array}{lcl}
primes\ n & = & [\,],\ \mathbf{if}\ n < 2 \\
 & = & [2],\ \mathbf{if}\ n = 2 \\
 & = & 2 : sieve\ [3]\ 5\ n,\ \mathbf{otherwise} \\
 & & \\
sieve\ p\ t\ n & = & p,\ \mathbf{if}\ n < t \\
 & = & sieve\ p'\ (t+2)\ n,\ \mathbf{otherwise} \\
 & & \mathbf{where}\ p' = append\ p\ t \\
 & & \\
append\ p : r\ t & = & app\ p : r\ t,\ \mathbf{if}\ t < p * p \\
 & = & p : r,\ \mathbf{if}\ t\ mod\ p = 0 \\
 & = & p : append\ r\ t,\ \mathbf{otherwise} \\
 & & \\
app\ [\,]\ e & = & [e] \\
app\ x : l\ e & = & x : app\ l\ e
\end{array}
$$

Figure 2.3: Prime Number Generator

The only parallelism lies in the overlapped execution of *append* and *sieve* in the second equation. Recall that the *cons* operation on lists is non-strict in its arguments. Hence, *append* may immediately return a result, i.e. the first cons-cell of the result list (with the tail of this cell not yet defined) and *sieve* may start working with this cell. The one process now successively creates the list while the other one simultaneously processes it. This form of *consumer-producer* parallelism is very important in dataflow programming.

How can we improve this solution? The basic problem is that the conditional construct forces the evaluation of its test condition and therefore represents a point of synchronization. In our example, it is not difficult to realize how this synchronization may be delayed to a later moment. The only difference between the second and the third equation is the list of primes submitted to the recursive activation of *sieve*. However, both lists differ only in the last element. In the one case, t is appended to the list, in the other case not. The basic trick is now to shift the decision of whether to append the element or not to the function *append*. *append* may start working and create p' (consisting of the elements of p) regardless whether t is to be appended or not. Only when the last element of the list is processed, this decision has to be made.

Hence, we may reformulate *sieve* as follows:

$$
\begin{array}{lcl}
sieve\ p\ t\ n & = & p,\ \mathbf{if}\ n < t \\
 & = & sieve\ p'\ (t+2)\ n,\ \mathbf{otherwise} \\
 & & \mathbf{where}\ p' = append\ p\ t
\end{array}
$$

A new list p' is created independently of t and forwarded to the next activation of *sieve* that may immediately start to check $t + 2$ with respect to p'. As we mentioned in the beginning of this section, in order to decide the primality of an integer, only all primes up to the square root of this number have to be known. Hence, not only $t + 2$ but all integers up to t^2 may be tested independently of the fact whether t is finally entered into p' or not.

The dynamic behaviour of the whole dataflow program (listed in Figure 2.3) is depicted in Figure 2.4. As soon as the upper bound n has arrived, $\frac{n}{2}$ pipelined processes, one for each odd integer to be tested, may be created. Each activated process activates its right neighbour, receives a prime list from its left neighbour and immediately forwards it to the newly activated process on its right

Figure 2.4: The Primes Pipeline

side. Simultaneously, the process starts a procedure deciding whether its integer is prime or not. As soon as it has finished this decision (and all primes have been forwarded from its left to its right neighbour), the forwarded list can be closed either by appending the tested integer or not.

Basically, the parallelism inherent in this dataflow program is proportional to the number of integers to be tested. However, in practice the speedup achieved will be more limited by the "resistance" of the pipeline, i.e. by the delay that, in each stage of the pipeline, the forwarded list is subject to. For large numbers of primes, this delay will be a significant factor.

2.4.2 Improved Prime Numer Generation

The program designed in the previous section showed a lot of parallelism. However, it was rather crude in the sense that in each stage of the pipeline a totally new list had to be created just for appending a single element. In this section, we will improve this method by using the concept of *difference lists* as described in [Abelson and Sussman, 1985] and [Shapiro, 1987].

Difference lists denote a very important programming concept in non-strict programming. A difference list is a particular list representation that allows the constant-time list concatenation. Each list l is represented by a pair of lists l_1 and l_2 (notation $l_1 \backslash l_2$) such that l is always the *difference* of l_1 and l_2. Two difference lists $l_1 \backslash l_2$ and $l_2 \backslash l_3$ are *compatible* in the sense that they can in a single step be concentated yielding the result list $l_1 \backslash l_3$.

We will now redesign the function *sieve* such that it works on a difference list representation of the primes list. The new version of *sieve* looks as follows:

$$
\begin{aligned}
\mathit{sieve}\ p\ t\ n &= [\,],\ \mathbf{if}\ n < t \\
&= \mathit{cons}\ p\ r\ t,\ \mathbf{otherwise} \\
&\quad\ \mathbf{where}\ r = \mathit{sieve}\ p\ (t+2)\ n
\end{aligned}
$$

$$
\begin{aligned}
\textit{primes } n &= [\,],\ \textbf{if } n < 2\\
&= [2],\ \textbf{if } n = 2\\
&= 2 : 3 : p, \textbf{otherwise}\\
&\quad \textbf{where } p = \textit{ sieve } 3 : p\ 5\ n\\[1em]
\textit{sieve } p\ t\ n &= [\,],\ \textbf{if } n < t\\
&= \textit{cons } p\ r\ t,\ \textbf{otherwise}\\
&\quad \textbf{where } r = \textit{ sieve } p\ (t+2)\ n\\[1em]
\textit{cons } p : l\ r\ t &= t : r,\ \textbf{if } t < p * p\\
&= r,\ \textbf{if } t \textbf{ mod } p = 0\\
&= \textit{cons } l\ r\ t,\ \textbf{otherwise}
\end{aligned}
$$

Figure 2.5: Improved Prime Number Generator

The interpretation of *sieve* has now subtly changed. The input parameter p does no more represent the list of all primes less than t but the list of *all* primes computed. Furthermore, *sieve* returns no more the list of all primes but only the list of primes greater than or equal t. Thus, the recursive call of *sieve* returns the list r of all primes greater than or equal $t + 2$. The part of the primes list that has already been computed by previous *sieve* activations is represented by the difference list $p \backslash r$. For the moment, let us ignore the dynamic interpretation of these facts and concentrate on the static (mathematical) relations:

If $n < t$, the result is the empty list. Otherwise, the result is the list r of all primes greater than t with possibly t consed to the head. The *cons* routine can simply be defined as follows:

$$
\begin{aligned}
\textit{cons } p : l\ r\ t &= t : r,\ \textbf{if } t < p * p\\
&= r,\ \textbf{if } t \textbf{ mod } p = 0\\
&= \textit{cons } l\ r\ t,\ \textbf{otherwise}
\end{aligned}
$$

If the test integer t is larger than the square root of the first prime p, t is itself prime and may be consed to r. If p divides t with remainder 0, the result is r itself. Otherwise, the test has to be continued with the tail l of the prime list.

The dynamic interpretation of the program is a little bit tricky, as one can see from the top-level function of the program:

$$
\begin{aligned}
\textit{primes } n &= 2 : 3 : p\\
&\quad \textbf{where } p = \textit{ sieve } 3 : p\ 5\ n
\end{aligned}
$$

The result of *primes* is the list $(2 : 3 : p)$ where p contains all odd prime numbers larger than 3. However, the list $(3 : p)$ is also input to sieve that itself computes p! Feeding results back to the functions that computed them is only possible in non-strict languages where functions need not from the beginning depend on all arguments.

For the following discussion of the dataflow of the program (listed as a whole in Figure 2.5), the reader may refer to Figure 2.6.

Executing this prime generating program, a pipeline of $n - 1$ *sieve* processes is created such that for each $\textit{sieve}_i$ $(1 \leq i \leq n - 1)$ the following propositions hold (where $n = n_1 = \ldots = n_{n-1}$):

Figure 2.6: The Primes Circuit

(1) $p_1 = p_2 = \ldots = p_{n-1}$.

(2) If $t_i \leq n$, r_i is a tail sublist of p_i.

(3) p_i contains all odd primes less than t_i.

(4) If $t_i \leq n$, r_i contains all odd primes greater than or equal t_i but smaller than n.

A list r is a *tail sublist* of p if the concatenation of some list l with r yields p, i.e. $l \parallel r = p$. The validity of these propositions can be shown as follows:

(1) is obvious since each $sieve_i$ passes p_i unchanged to $sieve_{i+1}$.

(2) holds in the case of r_0: By the second equation of $sieve_0$ and the first equation of $cons_0$, $(5 : r_0)$ is returned to *primes* as the result p. However, the input argument p_0 is just $(3 : p)$. For any r_{i+1}, either r_{i+1} or $(t_{i+1} : r_{i+1})$ is returned as r_i to $sieve_i$. By assumption, r_i is a tail sublist of p_i (and by (1) also of p_{i+1}) and so is r_{i+1}.

(3) holds for $sieve_0$ since $3 = head\ p_0$ is the only odd prime less than $5 = t_0$. Assuming that the proposition holds for $sieve_i$, we now show that it also holds for $sieve_{i+1}$. By assumption, p_i contains all odd primes less than $t_i = t_{i+1} - 2$. If t_{i+1} is not prime, the proposition therefore holds. However, if t_{i+1} is prime, $sieve_{i+1}$ detects this fact by examining all primes of p_{i+1} up to the square root of t_{i+1} and returns as its result the list r_i with head t_i. However, r_i is by (2) a tail sublist of p_i and by (1) also of p_{i+1}. Hence, t_{i+1} is also contained in p_{i+1}.

(4) holds for the last pipeline process $sieve_{n-1}$ that returns the empty list because $n < t_{n-1}$. Each other process $sieve_{i-1}$ (with $i \leq n-1$) receives the list r of all odd primes greater than $t_{i-1} = t_i + 2$ and smaller than n. By (3), $sieve_{i-1}$ may test t_{i-1} with respect to all primes in p_{i-1} that are smaller than or equal the square root of t_{i-1}. Iff t_{i-1} is prime with respect to this test, t_{i-1} is consed to r_{i-1} and returned as result. Otherwise, r_{i-1} itself is returned.

Anyway, the result is the list of all odd primes greater than or equal t_{i-1} and smaller than n.

The partial correctness of the program is a direct consequence of (4). In every activation $sieve_i$, the prime list p_i has an open (i.e. not yet defined) tail but, by (3), contains all primes that have previously been computed. By (2) and (4), the not yet defined tail of p_i is exactly the result list r_i returned by $sieve_{i+1}$. Vice versa, the already defined part of p_i can be represented by the difference list $p_i \backslash r_i$.

Dynamically, these relations can be illustrated by two streams of data flowing in both directions of the pipe:

- In one direction, the total prime list p flows through the pipe. In physical terms, p is a pointer to the storage location that represents the head of the primes list. The first pipeline process *primes* allocates a cons-cell, writes the value 3 in the head of this cell and passes a reference to this cell as first argument $sieve_1$. This reference is forwarded through all activations of *sieve*. As soon as $sieve_1$ has returned p, p is written into the tail of this cell and becomes thereby immediately available to all pipeline processes.
- In the other direction, all computed primes are collected to the stream r and driven back to the direction of the *primes* process. The last pipeline process $sieve_{n-1}$ returns the empty list to its predecessor $sieve_{n-2}$ (and by this to $cons_{n-2}$). If $cons_i$ has detected that t_i is prime, a new cons-cell is allocated and immediately returned to $cons_{i-1}$. t_i is written into the head of the new cell. As soon as the list reference r_i has been received from $sieve_{i+1}$ (i.e. from $cons_{i+1}$), this reference is written into the tail of the cons-cell and becomes thereby available to all other *sieve* processes. However, if t is not prime, then r_i is unchanged passed back through the pipe.

Summarizing, each prime element that is "consed" to the head of the stream r flowing back through the pipe is automatically appended to the primes stream p flowing in the forward direction.

However, we should utter some word of care. Although this revised program has been explained in terms of "streams" of integers "flowing" back and forward through the pipe, we should emphasize that these integers are actually *not* flowing as independent objects. Each *sieve* process only once receives a reference to a list of primes. All primes that are computed by other *sieve* processes are written into this list, hence they become immediately available to each other process. The flow of data is, in this case, just a model describing the complicated data dependences between processes.

This program version still has the same amount of inherent parallelism as that one presented in the previous section. Moreover, it totally copes without unnecessary list copies and is by this far more efficient.

By the program examples sketched in this and the previous section, the reader should have gained some insights into the possibilities of dataflow programming and should have developed some intuition into its particular flavour. Discussions on more advanced and subtle features of this programming methodology can be found in [Arvind *et al.*, 1988b] and [Arvind and Ekanadham, 1987].

2.5 Dataflow Architectures

For almost two decades now, there has been quite a lot of research on the efficient implementation of the dataflow principle in hardware. Several dataflow computers have been proposed in literature promising to efficiently utilize the fine-grain parallelism of declarative programs by hardware primitives for the creation of large numbers of parallel processes, their synchronization and their communication. Some prototypes of dataflow computers have been built in hardware, others have been extensively simulated, e.g.

1. the Manchester Dataflow Computer,
2. the ETL Sigma-1 at the Japanese Electrotechnical Laboratory,
3. the TTDA Tagged Token Dataflow Architecture at the MIT,
4. and its successor, the ETS Explicit Token Store "Monsoon".

In the following subsections, we give a short overview on these approaches. They all belong to the class of **dynamic dataflow architectures** which allow multiple concurrent activations of the same function by extending the token tags with unique activation names.

Comprehensive descriptions of several other (in particular earlier) architectures can be found in [Dennis, 1984] and [Arvind and Culler, 1986]. Most of them were **static dataflow architectures** which imposed the restriction that no more than one activation of a function existed at the same time (and by this prohibited recursion).

2.5.1 The Manchester Dataflow Computer

A machine based on the tagged-token model of dataflow computation has been operational at the University of Manchester since October 1981. This Manchester Dataflow Computer ([Gurd *et al.*, 1985], [Ruggherio and Sargeant, 1987]) has been one of the first dataflow machines actually built in hardware.

The Manchaster Dataflow Computer consists of only one processor whose basic structure is a **ring of four modules** connected to a host system via an I/O switch:

1. Token Queue,
2. Matching Unit and Overflow Unit,
3. Instruction Store,
4. Processing Unit.

The modules operate independently in a pipelined fashion. Tokens are carried in data packets circulating around the ring. Token packets that are destined for the same instruction are paired together in the Matching Unit using a hash table on the token tags. This unit has limited storage capacity such that an Overflow Unit is required for Programs with large data sets. Paired tokens (and those destined for unary instructions) fetch the appropriate instruction from the instruction store that contains the machine-code for the executing program. The instruction and its input data are forwarded to the Processing Unit for execution. Output tokens are transmitted back to the Matching Unit to enable subsequent instructions. The return path passes through the I/O switch that connects the system to a host processor and to the Token Queue. This queue serves as a buffer unit smoothing uneven rates of generation and consumption of tokens in the ring.

The performance of the Manchaster Dataflow Computer has been extensively evaluated by a lot of benchmark programs written in the single assignment language SISAL ([McGraw *et al.*, 1985]).

2.5.2 The ETL Sigma-1

The ETL SIGMA-1 at the Japanese Electrotechnical Laboratory ([Yuba *et al.*, 1985], [Hiraki *et al.*, 1984], [Shimada *et al.*, 1986], [Yuba, 1986], [Hiraki *et al.*, 1986]) has been operative since 1987. It provides a performance of about 600 MFlops with 128 processors. The basic structure of the processors that are connected by a two-level communication network is as follows:

There are five functional units each of which works synchronoulsy as a two stage pipeline:

1. Instruction Fetch Unit (first stage),
2. Matching Memory Unit (first stage),
3. Destination Unit (second stage),
4. Execution Unit (second stage),
5. Structure Memory Unit (both stages).

The processor is based on the tagged-token model of computation. Incoming tokens are forwarded to the Matching Memory Unit where the pairing of tokens is performed. Access to the Instruction Fetch Unit and the corresponding search in the Matching Memory Unit are carried out simultaneously. If the search is not successful, the instruction fetched from the instruction memory is discarded. The Execution Unit performs logical, integer and floating-point arithmetic. The Destination Unit produces an output packet combining the result value of the execution unit together with the destination address sent from the instruction fetch unit. After executing an instruction, the result is sent in package form to the input buffer memories via the network.

The Structure Memory Unit handles arrays that are expressed as named sequences of values. These sequences are called B-structures and obey the single-assignment rule, i.e. any element of a B-structure may be written at most once. The structure memories of all processors connected by the first stage of the interconnection network are shared by all processors of that group.

Another project at the Electrotechnical Laboratory has dealt with the construction of a LISP-based data-driven machine ([Yuba *et al.*, 1983], [Herath *et al.*, 1985], [Yamaguchi *et al.*, 1984], [Yamaguchi *et al.*, 1983]).

2.5.3 The TTDA Tagged Token Dataflow Architecture

The Tagged Token Dataflow Architecture (TTDA) at the MIT is the result of one of the most advanced and longest ongoing activities in dataflow research ([Arvind and Nikhil, 1988]). The TTDA consists of a number of identical dataflow processors and storage units interconnected by an n-cube packet network. The storage units are addressed uniformly in a *global address space*.

Tokens arriving at a dataflow processor enter a pipeline and go through the following units in sequence:

- **Wait-Match Unit:**

  This unit consists basically of a pool of waiting tokens. Tokens destined for monadic operators go straight through this unit to the Instruction-Fetch Unit. If a token is destined for a dyadic operator, the tokens in the Wait-Match Unit are examined to see if the other operand has already arrived. If so, both tokens are passed to the Instruction-Fetch Unit, if not, the token is deposited to wait for it.

- **Instruction-Fetch Unit:**

  The tag on the operand tokens entering the Instruction-Fetch Unit identifies an instruction to be fetched from program memory. The data values of the tokens, the opcode of the instruction, the destination offsets from the instruction and the context field of the token's tag are extracted and passed to the to the next stage.

- **ALU and Compute-Tag Unit:**

  These units are actually two ALUs operating in parallel. The ALU takes the operand data values and the opcode, performs the operation and produces a result. The Compute-Tag Unit takes the context and the instruction offsets and computes the tags for the output of the operator. Both units pass their results to the Form-Tokens Unit.

- **Form-Tokens Unit:**

  This unit takes the data value(s) from the ALU and the tags from the Compute-Tag unit and combines them into result tokens.

Output tokens emerging at the bottom of the pipe are routed according to their destination addresses: either back to the processor pipeline or into the network to some storage unit or another processor.

The store units contain so called **I-Structures** ([Arvind *et al.*, 1987]) i.e. arrays with single assignment semantics and special support for read-write synchronization.

The TTDA has not been built in hardware but extensively simulated on a network of TI-Explorer workstations. A variety of programs have been written in the functional language Id ([Nikhil, 1988], [Nikhil, 1987], [Traub, 1986]) and a lot of interesting results concerning dataflow programming have been gained ([Arvind *et al.*, 1988a], [Arvind and Culler, 1987]).

2.5.4 The ETS Explicit Token Store "Monsoon"

The ETS Explicit Token Store architecture "Monsoon" has been designed as a successor to the TTDA and probably represents the most advanced concept for dataflow computers today ([Papadopoulos, 1988], [Papadopoulos and Culler, 1990]) . The ETS architecture introduces a interesting new concept that is different from the other approaches and removes a main source of inefficiency:

In all the dataflow architectures presented in the previous subsections, some kind of Wait-Match Unit is responsible for the correct synchronization of tokens destined for dyadic operators. However, a token arrived has to be checked against **all other tokens** in the unit waiting for a match. The matching function and its associated store are mostly realized by a large hash table and sophisticated insertion and extraction algorithms. The resulting hardware is highly complex and rather slow.

In the ETS model, operand matching is performed at compiler-designated offsets within a given activation frame. For each function invocation, a contiguous block of memory is allocated. This activation frame contains for each binary instruction a location that represents the rendezvous point for the corresponding tokens. These location offsets may be already fixed at compile time. Hence, the context field of a token destined for a dyadic operator may explicitly hold the address of the location where the partner token is to be met. **No more search for the partner token** is neccessary, hashing algorithms and an associative wait store are no more required.

Besides this innovation, the ETS architecture corrects a number of other inefficiencies inherent in previous dataflow machines. Particular emphasis has been laid on a clear and simple processor design in order to support an implementation in hardware.

Hence, the Monsoon has not only been simulated as the TTDA but two monoprocessor prototypes have already been realized in cooperation with a microprocessor manufacturer (Motorola). A first multiprocessor consisting of 8 dataflow processors (with a performance of 10 million tokens per second each) and 8 I-structure memories has been announced to be operational in the spring of 1991. Motorola will also support a Unix-based single processor/single memory workstation for ID program development.

Chapter 3

The EVE Programming Language

This chapter is a reference manual for the non-strict and purely functional programming language EVE. EVE has been designed as a simple and efficient high-level tool for programming the parallel ADAM machine. On the one hand, this language combines the most important features that are state-of-the-art in functional programming, on the other hand we have omitted all that facilities that are not essential but make a prototype implementation much more difficult.

EVE is rather small and simple, but it is definitely not a toy language. In particular, the language shows the following features:

- EVE **is purely functional:**

  An EVE program is a composition of functions that are all totally free of side-effects. The result value of a function only depends on the values of its arguments and of globally defined constants but not on any other influence. Vice versa, the evaluation of any expression returns a value and has no effect besides that.

- EVE **is non-strict:**

  Functions and data structures are in general non-strict. Therefore, functions may return results even if (some or all of) their arguments have not yet been provided. Furthermore, data structures may be handled, even if (some or all) components of them are not yet evaluated.

- EVE **is strongly typed:**

  The programmer is required to give explicit type annotations to all newly defined objects. Therefore, the system is able to check the correct typing of all definitions and expressions at compile time. The type system of the language includes several structured data types and also allows the construction of recursively defined types.

- EVE **is higher-order:**

  Functions are first-order objects as well as the objects of atomic or structured types. In particular, functions can take other functions as arguments and return functions as results. Moreover, functions may be also stored as part of data structures.

- EVE **is safe:**

  EVE programs never produce runtime errors that force a halt of the execution. Unforeseeable errors like "Division by Zero" are handled by the use of special error values that are returned as the results of invalid computations. This is especially important in a parallel environment since here it is not desirable that one processor halts on an error while the others continue working.

In spite of the fact that EVE has been designed especially for the needs of parallel computation, parallelism is not mentioned anywhere in the language definition. The parallel evaluation of a functional program is not a matter of the definition of the functional language but a matter of its implementation. Any (also sequential) system that fulfills the requirements of the language definition provides a valid implementation of the language.

In order to keep the language small, the following features have been explicitly omitted, either because they just represent less important *syntactic sugar* or because they make an experimental implementation essentially more complex[1]:

- Pattern Matching,
- Type Inference and Polymorphism,
- Lambda Expressions and Curried Functions,
- List Comprehensions,
- Delayed Evaluation,
- Abstract Data Types and Modules.

A former version of this reference manual has been published in [Schreiner, 1989].

3.1 Some Hints for Further Reading

Before we start the description of the EVE language, some hints on notational conventions will help to understand the following sections:

Syntax Notation

The exact syntax rules of the language are given in Section 3.15.3. For the moment, it suffices to distinguish two kinds of syntactic entities:

Terminal symbols, i.e. symbols that can appear in the source code of a program, are written in `typewriter` type style. All program examples are also typed in this style, e.g.

```
F[x]: INT[INT] == x*x
```

Nonterminal symbols, i.e. symbols that represent classes of tokens, but can not appear in a program text themselves, are written in *italic* type style, e.g.

Identifier

Layout

EVE does not impose any restrictions upon the format of source texts (*free format style*). Only the sequence of characters in the text is significant, indentations or other formatting features do not have any effect. As the only exception to this rule, line breaks denote the end of comments, which is described in the following subsection.

[1] A further version of the language will perhaps include some of them.

Comments

start with the token `--` and continue to the end of the line. Everything between this token and the end of the line is ignored. In fact, a line containing a comment is treated, as if it had ended before the `--`. Comments may not start within text constants (see Section 3.4.3).

```
-- This is a comment!
```

Identifiers

consist of a sequence of letters and digits, but have to start with at least one letter. EVE is case-sensitive, i.e. it distinguishes between lower and capital letters, therefore

`ident`, `Ident` and `IDENT`

denote three different identifiers. The end of an identifier is recognized by the first character that is neither a letter nor a digit, therefore two identifiers have to be separated by at least one such character.

Keywords and Predefined Identifiers

are written in capital letters, e.g.

```
IF NULL[l] THEN l ELSE m.
```

Keywords (`IF, THEN, ELSE`) are reserved and may not be redefined, while predefined identifiers (`NULL`) may[2] A list of the keywords and predefined identifiers is given in Sections 3.15.5 and 3.16.

3.2 Overall Program Structure

A program is a sequence of *definition statements* (also called *bindings*) separated by semicolons, i.e.

$Definition_1$;
$Definition_2$;
...
$Definition_n$

Each of these statements binds a name (an *identifier*) to some kind of object. The collection of bindings sets up the environment, in which a (user-given) expression is evaluated.

Typically, the file containing the program's source code is submitted to a compiler and translated into the machine code of the underlying hardware. Using an interactive dialogue shell, the user may enter expressions, which are evaluated within the environment generated by the compilation. The results of these evaluations are printed on the screen or written into an output file.

According to the kind of object an identifier is bound to, there are three kinds of definition statements:

[2]However, to do so shows a rather poor programming style.

Type	*Description*
INT	Integer numbers
BOOL	Boolean values
CHAR	Characters
STR	Strings

Table 3.1: Predefined Types

- Type definitions,
- Constant definitions,
- Function definitions.

The order of definition statements within the source text is significant, the programmer has to obey two simple rules:

1. The definition of an identifier must textually precede its use: The scope of the identifier starts at the point of its definition and extends over the right hand sides of its own definition statement and all the following ones, as well as over the expression to be evaluated[3].
2. Within a level of scope an identifier may be defined at most once, i.e. redefinitions are not allowed. The possibility of redefining predefined identifers is not affected by this rule, since these identifiers are defined in an outer level of scope than the user definitions take place.

These rules require some programming discipline but are not really limitations of EVE. Their only purpose is to facilitate the construction of a compiler for the language.

3.3 Type Definitions

EVE is a *strongly typed* language, i.e. all operators and functions demand arguments of certain types. Typing errors are detected on compilation time and incorrectly typed programs are rejected. All types are *named*, i.e. they have to be introduced by a type definition statement of the form

Ident == *Type*

where *Ident* is the name of the type to be introduced and *Type* is its definition.

EVE knows four predefined types (see Table 3.1) and three constructors that build new types from already existing ones[4] (see Table 3.2).

A type definition may simply rename a previously defined type:

```
Number  ==  INT;
Word    ==  STR;
Letter  ==  CHAR
```

[3] As we will see, there are some relaxations to this rule in order to allow mutually recursive functions (see Section 3.10), data types (see Section 3.5.4) and constants (see Section 3.6).

[4] A fourth type constructor, the *function constructor*, is described in Section 3.11.

Constructor	*Description*
`LIST OF`	Sequence of equal type elements
`TUPLE OF`	Fixed size collection of elements
`UNION OF`	Element of a selected type

Table 3.2: Type Constructors

But it may also introduce a new type that is constructed from other types:

```
Sequence  ==  LIST OF INT;
Person    ==  TUPLE OF name: STR, age: INT, sex: BOOL;
Datum     ==  UNION OF Num == INT, Word == STR
```

For strong type checking, it is important to know, in which cases two types are considered to be equal. As a rule, two types are equal, if and only if

1. both types have the *same type name*, or if
2. one type is derived from the other by a sequence of *renamings*.

In the following example,

```
Number   ==  INT;
Integer  ==  INT
```

type `Number` equals `INT`, as well as `Integer` equals `INT`. But `Number` *does not equal* `Integer`, since both have different type names and no type is a renaming of the other one!

The scope of a user-defined type starts at the right hand side of the definition statement itself, therefore directly recursive type definitions like

```
Foo == TUPLE OF x: INT, y: Foo;
```

are principially allowed, but do not make sense. In contrast to directly recursive types, mutually recursive type definitions can be reasonable in some cases but violate the rule that types have to be defined before they can be used (see Section 3.2). Therefore, EVE provides a special kind of type definition, the *forward type definition*. Its general form is simply

Ident

which makes public the name *Ident* of a new type but not its construction. From the point of this definition on, references to *Ident* are allowed, but the actual type definition has to take place in the same level of scope. Hence, mutually recursive type definitions like

```
Subtree;
Tree      ==  UNION OF Node == INT, Inner == Subtree;
Subtree   ==  TUPLE OF value: INT, left: Tree, right: Tree
```

are possible. However, a sensible recursive type definition must contain at least one union type (see Section 3.5.4), but the compiler is not forced to check whether type definitions are reasonable or not[5].

In the following sections, we will provide a detailed description of the different kinds of types and type constructors.

3.4 Atomic Types

A type is called *atomic*, if objects of this type can not be partioned into smaller semantic entities. EVE knows three predefined atomic types,

- Integers (`INT`),
- Booleans (`BOOL`),
- Characters (`CHAR`).

3.4.1 Integers

The data type `INT` provides signed, fixed-size integer values in a range limited by the predefined integer constants

`MININT` and `MAXINT`

denoting the lower bound and the upper bound of the range, respectively. The values of these constants depend on the particular implementation but the assertion `MININT` $< 0 <$ `MAXINT` can be expected to hold[6].

Numerical expressions, i.e. expressions of type `INT`, look like common mathematical expressions, e.g.

```
a*(b+1)/c-d.
```

EVE provides five binary integer operators and one unary operator (see Table 3.3) that obey the usual priority rules (the unary `-` binds stronger than `*`, `/`, and `%` that again bind stronger than `+` and the binary `-`).

The result of a remainder operation may be either `0` or it has the sign of the dividend regardless of the sign of the divisor, e.g.

```
-17 %  5 = -2,
 17 % -5 =  2.
```

The predefined function associated to each operator represents the prefix version of this operator[7], therefore the expression

---

[5] Even if objects of a meaningless type are constructed, they can not cause any serious harm. They are just not very useful.

[6] Typically, these constant values equal -2^n and $2^n - 1$ on n-bit machines.

[7] The neccessity for the existence of these functions is explained in Section 3.11

Operator	*Function Name*	*Description*
`+`	`ADD`	Addition
`-`	`SUB`	Subtraction
`*`	`MUL`	Multiplication
`/`	`DIV`	Division
`%`	`MOD`	Remainder
`-`	`NEG`	Negation

Table 3.3: Integer Operators

Function	*Description*
`ABS`	Absolute Value
`ODD`	Is number odd?

Table 3.4: Integer Functions

```
(-a)*(b-c)
```

is equivalent to

```
MUL[NEG[a],SUB[b,c]].
```

The use of fixed-size integer numbers raises some problems, since integer operations may return values that are outside the range of type `INT` (*"Overflow"*) or not defined (*"Zero Division"*). EVE handles these errors by the use of special *error values*, a detailed description of the error behaviour can be found in Section 3.13.

Two more predefined functions operating on integer values are listed in Table 3.4.

- `ABS[i]` returns the absolute value of the integer `i`. This function may return the error value, if `i` is negative and has no positive counterpart within the range of `INT` or vice versa.
- `ODD[i]` returns `TRUE`, if `i % 2 = 0` does not hold.

3.4.2 Booleans

The data type `BOOL` provides two predefined constants, the values

`TRUE` and `FALSE`.

For expressions of type `BOOL`, three operators (and associated functions) have been defined:

In addition to the usual priority rules (`NOT` (`~`) binds stronger than `AND` (&) which binds stronger than `OR` (|)), the binary operators obey the *short circuit* discipline: if a conjunction or disjunction is evaluated, the evaluation starts with the left operand. If this partial result already determines the result of the whole operation (i.e. if it is `TRUE` in disjunctions or `FALSE` in conjunctions) the second operand is not evaluated. We say:

Operator	*Function Name*	*Description*
`&`	`AND`	Conjunction
|	`OR`	Disjunction
`~`	`NOT`	Negation

Table 3.5: Boolean Operators

`AND` (&) and `OR` (|) are both strict in their first argument.

The strictness of a function or data structure may have a strong influence on the runtime behaviour and even on the termination of a program. Therefore we will discuss this important point in Section 3.12 in greater detail.

3.4.3 Characters

The data type `CHAR` provides the characters of the US-ASCII character set[8].

Character constants consist of a single character enclosed by single quotation marks ('). Valid characters between these marks are

- Any *printable character* of the ASCII character set except the backslash character (\) and the single quotation mark ('), e.g.

  `'a'`, `'b'`, `'Z'`, `'␣'`, `'*'`, `'$'`.

- An *abbreviation* of the form `\l`, where `l` is a character code representing a special character (see Table 3.6), e.g.

  `'\n'` *(New Line)*, `'\''` *(' itself)*,

- A *code sequence* of the form *ddd*, where *ddd* is a three digit decimal ASCII code number from 000 to 255, e.g.

  `'\036'` *($)*, `'\065'` *(A)*, `'\127'` *(DEL)*.

Within a character constant, it is illegal to use the backslash character in another way than just described. A character constant (i.e. the sequence *'c'*, where *c* denotes a character) must not be splitted over several lines, therefore the end of a line and the comment token `--` are not allowed to appear within it.

EVE has three predefined functions handling the conversion between characters, from characters to integers, and vice versa (see Table 3.7).

- `CAP[ch]` returns the capital of `ch`, if `ch` is a letter, and `ch` itself, otherwise.
- `CHR[int]` returns the character with ASCII code `int`, if `int` is in the range 0–255, and the error value[9] otherwise.
- `ORD[ch]` returns the integer value denoting the ASCII code of character `ch`.

[8] A table of this character set can be found in Section 3.15.2.
[9] For further explanation on error handling, see Section 3.13.

Special Character	*Description*	*ASCII code*
`\'`	' in character constants	044
`\"`	" in string constants	034
`\\`	\ itself	092
`\b`	Backspace	008
`\f`	Form Feed	009
`\n`	Line Feed	010
`\r`	Carriage Return	013
`\t`	Horizontal Tab	009
`\_`	String Interrupt	—

Table 3.6: Special Characters

Function	*Description*
`CAP`	Capitalize a letter
`CHR`	Character from ASCII code
`ORD`	ASCII code of a character

Table 3.7: Character Functions

3.4.4 Comparison Operators

Table 3.8 shows the usual comparison operators that have been predefined on all three atomic types.

These operators are overloaded in the sense, that each of them is a placeholder for one of three functions, one function for each atomic type. The asterisk * in the function names of Table 3.8 has actually to be replaced by `I (INT)`, `B (BOOL)`, or `C (CHAR)` to get the particular comparison function for each data type, e.g.

```
ILT[2,5], BNE[TRUE,FALSE], CEQ['f','G'].
```

The comparison order for each type is defined as follows:

- `INT`: The comparison order of integer values is obvious.

Operator	*Function*	*Description*
`<`	`*LT`	Less Than
`<=`	`*LE`	Less or Equal
`=`	`*EQ`	Equal
`#`	`*NE`	Not Equal
`>=`	`*GE`	Greater or Equal
`>`	`*GT`	Greater Than

Table 3.8: Comparison Operators

- `BOOL`: Boolean values are ordered by `FALSE < TRUE`.
- `CHAR`: Characters are compared according to their ASCII values.

3.5 Structured Types

EVE knows three *type constructors* that generate new structured types from previously defined types,

- the *List Constructor* (`LIST OF`),
- the *Tuple Constructor* (`TUPLE OF`),
- the *Union Constructor* (`UNION OF`),

and one predefined structured type,

- the *String* type (`STR`).

3.5.1 Lists

Lists are variable-size sequences of elements of equal type. List types are defined by statements of the form

Ident `==` `LIST OF` *SimpleType*

where *SimpleType* denotes the name of a previously defined type.

Some examples of list types are

```
NumSeq    ==  LIST OF INT;
Word      ==  LIST OF CHAR;
Sentence  ==  LIST OF Word
```

EVE does not allow nested type definitions, so statements like

```
Sentence == LIST OF LIST OF CHAR
```

are invalid and have to be replaced by the introduction of auxiliary type definitions (as shown above).

Every list type possesses its own *constructor function*, which has the same name as its associated type and builds lists of this type. E.g. if we define

`NumSeq == LIST OF INT`,

then the expression

```
NumSeq[2,3,5,7,11,13,17]
```

Function	*Description*
`::`	Construction of a new list (infix)
`CONS`	Construction of a new list (prefix)
`HEAD`	Selection of the list head
`TAIL`	Selection of the list tail
`NULL`	Test a list for emptiness

Table 3.9: List Operators and Functions

constructs a list of type `NumSeq` which contains the first seven prime numbers. These elements are entered into the list from right to left, therefore 2 is the first element (the *head*) of the constructed list.

The empty number sequence (i.e. the list without any elements) is constructed by

```
NumSeq[ ].
```

All predefined generic list operators and functions[10] are listed in Table 3.9. However, these functions are *not* first-order objects and can therefore not be used as arguments to higher order functions (see Section 3.11).

- The operator `::` is the infix equivalence of the generic function `CONS`. It inserts its first argument of type *Any* into its second argument which is a list of type *AnyList*, where

  AnyList `== LIST OF` *Any*.

  E.g. both of the expressions

  ```
  2 :: NumSeq[3,5,7,11],
  CONS[2,NumSeq[3,5,7,11]]
  ```

  are equivalent to

  ```
  NumSeq[2,3,5,7,11].
  ```

- The components of a list can be extracted by the function `HEAD` and `TAIL` that return the first element of a list (its *"head"*) and the list without its first element (the *"tail"*). As a simplification, both of these selector functions (and only these!) may be applied to a list expression in the form

  List`.`*Selector*

  E.g., both of the expressions

  ```
  HEAD[NumSeq[2,3,5,7]]
  NumSeq[2,3,5,7].HEAD
  ```

  return the value 2, and both of the expressions

[10] In this case, *generic* means *operating on all kinds of lists.*

```
TAIL[NumSeq[2,3,5,7]]
NumSeq[2,3,5,7].TAIL
```

return the list `NumSeq[3,5,7]`.

It is illegal to apply `HEAD` or `TAIL` to an empty list. How this error is handled, is described in Section 3.13.

- `NULL[list]` returns `TRUE`, if and only if `list` is empty i.e. does not contain any element.

3.5.2 Strings

A *string* is simply a list of characters. Therefore, the predefined standard identifier `STR` can be user-defined by[11]

```
STR == LIST OF CHAR.
```

Because of this interpretation, all predefined list selectors and functions can also be applied to strings.

String constants consist of a (possibly empty) sequence of characters enclosed by double quotation marks ("). E.g.

```
"A string"
```

is a simplified version of the list construction

```
STR['A',' ','s','t','r','i','n','g'].
```

The rules for valid characters within string constants are basically the same as for character constants, with two exceptions:

- *Double quotation marks* (") have to be encoded by the sequence \", while single quotation marks stand for themselves. Hence, the string constant

  ```
  "He said \"I don't like programming\"."
  ```

  generates the string value

  ```
  He said "I don't like programming".
  ```

- String constants may be *split over several lines* by the use of the special character _. The first occurrence of this character within a string constant denotes an interruption of the string, after the next occurrence of _ (which may happen several lines afterwards) the string is continued. Between an string interruption and its continuation only white space (see section 3.15.2) and comments are allowed. E.g. the string constant

  ```
  "This is a stri\_  -- Break
  \_ng interruption."
  ```

  generates the string value

  ```
  This is a string interruption.
  ```

[11] In spite of this fact, it is strongly recommended not to define ones own private string type by `WORD == LIST OF CHAR`, because any implementation may handle the type `STR` in a more efficient way. It is a better strategy to rename the predefined type by `WORD == STR`, since renamings do not create a new type.

3.5.3 Tuples

Tuples are fixed-size collections of elements with equal or different types (i.e. they represent the counterpart to the concept of *cartesian products* in denotational semantics). Tuple types are defined by statements of the form

$$Ident \texttt{ == TUPLE OF } Id_1 : Type_1 \texttt{ , } Id_2 : Type_2 \texttt{ , } \ldots\ Id_n : Type_n$$

where Id_1 to Id_n are the *field names* of the tuple and $Type_1$ to $Type_n$ their corresponding types, e.g.

```
Coordinate  ==  TUPLE OF x: INT, y: CHAR;
Name        ==  TUPLE OF first: Word, last: Word;
Address     ==  TUPLE OF street: Word, number: INT
```

Similar to lists, the components of a tuple must have explicit type names, anonymous type components, such as in

```
Name == TUPLE OF first: LIST OF CHAR, last: Word
```

are not allowed.

Tuples are always constructed as a whole by the use of *constructor functions*. These functions have the same name as their associated types, e.g.

```
Coordinate[3,'A']
```

creates a new tuple with `x`-field set to 3 and `y`-field set to `'A'`.

The field names of a tuple are always local to the tuple, so definitions like

```
First == TUPLE OF x: INT,  y: INT;  -- Fields x and y
Secnd == TUPLE OF x: CHAR, y: CHAR; -- Fields x and y, too!
```

are perfectly valid.

Components of a tuple can be selected by use of the *selectors* that are associated to each tuple type and have the names of the fields to be selected.

The application of these selectors is (similar to the list selectors) of the form

$$Tuple.Selector$$

E.g. if `name` is of type `Name` (as defined above), then the expressions

```
name.first,
name.last
```

return the first and the second component of the tuple, respectively.

3.5.4 Unions

Sometimes, the type of an object is not known at compile time, but there is only a fixed set of possibilites. In these cases, the *union constructor* may be used. A union type is rather a collection of types than a single type itself (it represents a concept equivalent to the *disjoint unions* in denotational semantics). A *variant* of a union type is one of the types collected in the union. The definition of a union type is of the form

$$\mathit{Ident} \texttt{ == UNION OF } \mathit{Id}_1 \texttt{ == } \mathit{Type}_1 \texttt{ , } \mathit{Id}_2 \texttt{ == } \mathit{Type}_2 \texttt{ , } \ldots \texttt{ , } \mathit{Id}_n \texttt{ == } \mathit{Type}_n$$

where Id_1 to Id_n are the variants of the union, i.e. renamings of the types Type_1 to Type_n. Although definitions of union types look similar to definitions of tuple types, it is important that one does not mix them up: In contrast to tuples, unions are not collections of constants but *collections of types*. Elements of a union type have as their actual type exactly *one variant* of the union.

The following statements are examples for the definition of tuple types:

```
Data        ==  UNION OF Num == INT, Word == STR;
Person      ==  UNION OF Man == Male, Woman == Female;
Coordinate  ==  UNION OF Pol == Polar, Cart == Cartesian
```

An object of type `Data` may now be either an integer or a string, as well as a `Coordinate` value may be given either in polar or in cartesian form.

A union value is constructed by the application of a *conversion function* which converts an expression from a type that equals one variant of the union to the union type itself. This function has the name of its associated union type but is augmented by the name of the variant. Its general form is

UnionType`$`*Variant*`[`*expression*`]`.

where the $ separates the name of the union type from the name of the variant. Thus, the expressions

```
Data$Num[317],
Data$Word["Hello"]
```

create values that are both of type `Data`, but hold different variants of this type.

As with tuples, the variant names of a union type are local to the union and can therefore be used for definitions outside the union.

As already mentioned, the definition of a name must textually precede all of its applied occurrences. In order to alllow mutually recursive type definitions, EVE knows *forward type definitions* of the form

Ident.

A forward type definition makes public only the name *Ident* of a type, but not its construction. From the point of this definition on, references to this type are allowed, but the actual type definition has to take place within the same level of scope. E.g. in the definition sequence

```
Subtree;
Tree      ==  UNION OF Node == INT, Inner == Subtree;
Subtree   ==  TUPLE OF value:INT, left: Tree, right: Tree
```

the forward reference of `Tree` to `Subtree` is resolved by the use of a forward type definition.

Section 3.8 describes in detail how the actual variant of a union value can be determined and how a union value can be reconverted to its original type.

3.6 Constant Definitions

Constant definitions introduce names for anonymous values. Their general form is

Ident : *SimpleType* == *Expression*

Ident is the constant name to be introduced, *SimpleType* denotes the name of a previously defined type and *Expression* provides the value of the constant. The result value of *Expression* has to be of type *SimpleType*.

The following statements define some constant names:

```
i:  INT   ==  (2+3)*5;
s:  STR   ==  "The art of programming";
b:  BOOL  ==  (1 > 0) & ("test".HEAD = 't')
```

The scope of a constant name starts at the point of its definition, hence the following chain of definitions is perfectly valid:

```
n:  INT  ==  -217;
m:  INT  ==  ABS(n);
a:  INT  ==  n+m
```

Of course, constant definitions are not restricted to constants of atomic types, but every structured type also has to be given an explicit type name, e.g.

```
IntSeq       ==  LIST OF INT;
Pair         ==  TUPLE OF c: CHAR, i: IntSeq;

s:  IntSeq   ==  IntSeq[3,5,7,9];
p:  Pair     ==  Pair['A',2::s]
```

Since the scope of a constant also extends over the right hand side of its own definition, recursive constant definitions, such as

```
x: IntSeq == 1 :: x
```

are possible. This definition introduces a list `x` consisting of an infinite sequence of the value `1`. In order to allow mutually recursive constant definitions, EVE provides as a special form the *forward constant definition*. Its general form is

Ident : *SimpleType*

This definition makes public the name and the type of a new constant but not its value. From the point of this definition on, references to the constant are allowed. However, the actual constant definition has to take place within the same level of scope. Using this feature, mutually recursive constant definitions like

```
x2: IntSeq;
x1: IntSeq    ==   1 :: x2;
x2: IntSeq    ==   2 :: x1
```

are possible. These definitions introduce a list `x1` consisting of a infinite sequence of alternating values `1` and `2` starting with `1` and a list `x2` consisting of a infinite sequence of alternating values `2` and `1` starting with `2`.

A recursive constant definition only makes sense, if the expression on the right hand side of the definition contains *non-strict* operators or functions, e.g. the list constructor `CONS` (`::`). On the contrary, a definition like

```
undef: INT == undef+1
```

introduces a name `undef` for a value that will never be computed, since the strict operator `+` has to wait until all of its arguments have become available. For a detailed description of the strictness properties of the language, see Section 3.12.

Although the lists `x`, `x1` and `x2` defined above are logically inifinite, they can be represented in a finite (and actually very small) amount of computer memory. Altogether their construction only requires three applications of `CONS` (`::`), their infinite behaviour is a result of their (mutually) recursive definitions.

3.7 Conditional Expressions

A conditional expression is of the form

`IF` *Cond* `THEN` $Expr_1$ `ELSE` $Expr_2$

where *Cond* is a boolean expression, the *test condition*, and $Expr_1$ and $Expr_2$ are expression of equal type, e.g.

```
IF n = 0 THEN 1 ELSE n * Fib[n-1],
a + (IF a = 1 THEN b ELSE c) * d,
(IF 1 = 1 THEN 1 ELSE 0) > 0.
```

The if-then-else construct is *strict* in the first of its arguments, i.e. if its value is computed, first the boolean test condition is evaluated to `TRUE` or `FALSE`. Depending on the result, the expression of one branch of the construct *(but not the other one!)* is evaluated and returned as the result of the whole expression.

For a more detailed discussion on the strictness, respectively non-strictness of functions and data structures, see Section 3.12.

3.8 Case Expressions

The case construct is used to select the actual type of a union-typed value (see Section 3.5.4) and to extract its actual value. A *case expression* is an expression of the form

CASE *UnionExp*

$\{c_1\colon \mathit{Variant}_1\}$ Exp_1
$\{c_2\colon \mathit{Variant}_2\}$ Exp_2
...
$\{c_{n-1}\colon \mathit{Variant}_{n-1}\}$ Exp_{n-1}

ELSE Exp_n

where *UnionExp* is an expression that returns a union value, $\mathit{Variant}_1$ to $\mathit{Variant}_{n-1}$ represent different variants of this union and c_1 to c_n are identifiers used to define new constants. Exp_1 to Exp_n are expressions of equal type.

Each of the branches of the case construct tests the value returned by *UnionExp* for its actual variant (i.e. for the actual type of the union value). Each variant must not be tested by more than one branch, i.e. $\mathit{Variant}_1$ to $\mathit{Variant}_{n-1}$ have to be distinct. If $\mathit{Variant}_i$ matches the actual variant of the union value, the identifier c_i is bound to the value that the variant holds, i.e. c_i has type $\mathit{Variant}_i$. The scope of the constant name c_i is Exp_i, i.e. c_i may be freely used in Exp_i but is neither known in the other expressions Exp_j, with $i \neq j$, nor outside the case expression. The expression Exp_i is evaluated and returned as the result of the whole case expression.

If no $\mathit{Variant}_i$ matches the actual variant of the union value, the expression Exp_n in the else branch of the construct is evaluated and returned as the result of the case expression. If the variants tested by the case construct are *exhaustive*, i.e. if each of the variants of the union type is tested by one branch, the else branch of the construct has to be omitted.

E.g., using the definitions

```
-- Types
Data == UNION OF Num == INT, Letter = CHAR, Word == STR;
Node == UNION OF Flag == BOOL, Value == INT;

-- Constants
d: Data == Data$Word["Foo"];
n: Node == Node$Value[317]
```

the expression

```
CASE d

    {n: Num}    n*2
    {l: Letter} ORD[l]+1

    ELSE -1
```

returns the value `-1` and the expression

```
CASE n
```

```
{f: Flag}  ~f
{v: Value} v=0
```

returns the value `FALSE`. The second example shows a case expression with exhaustive variants, i.e. each of the variants of type `Node` is handled by one branch, therefore the else part has to be omitted.

There is one point that might be misleading: Every identifier c_i defined by

$$c_i \; : \; Variant_i$$

is of type $Variant_i$ which is a renaming of type $SimpleType_i$ in the definition of the underlying union type:

$$UnionType \texttt{ == UNION OF} \ldots Variant_i \texttt{ == } SimpleType_i \ldots$$

Therefore, in the first case expression shown above, `n` is a constant of type `Num`. However it may be freely used in integer expressions, since `Num` is merely a renaming of `INT`.

The case construct is strict in its first argument, i.e. in the expression *UnionExp*. Depending on the actual variant of the union value, one of the expressions Exp_i (*but none of the other ones!*) is evaluated and returned as the result of the whole expression.

A more detailed discussion on the strictness properties of EVE can be found in Section 3.12.

3.9 Block Expressions

A *block* is an expression of the form[12]

$$\begin{array}{l} \{ \\ \quad Definition_1 \; ; \\ \quad Definition_2 \; ; \\ \quad \quad \ldots \\ \quad Definition_n \\ \} \\ \quad Expression \end{array}$$

The value of a block is the value of the trailing expression that is evaluated in the global environment augmented (and possibly overridden) by the bindings established within the block, e.g.

```
{
    a:INT == 3;
    b:INT == a+2;
}
    (a + b) * 7
```

[12]The curly braces {} replace the more known `LET IN` construct of other functional languages.

evaluates to 56.

Every definition statement binds an identifier to a value and augments the environment by this new binding. The scope of the binding starts at the point of the definition statement and extends over the right hand side of this definition and all the following ones (up to the end of the block), as well as over the expression to be evaluated. Any binding to an identifier `x` outside the block is overridden by a redefinition of `x` within the block. The other way round, the binding of an identifer within a block is invisible outside the block.

Blocks may be nested and follow the *lexical scoping* discipline. The value of an identifier that has not been defined within a block is taken from the surrounding block. If it has not been introduced there, the value is taken from the block on the next outer level of nestings, and so on, up to the global level of definition statements.

In contrast to many other functional languages, in EVE the scope of an identifier starts at the point of its definition, previous occurrences of the identifier take their values from the surrounding environment. E.g. the nested block expression

```
{   a:INT == 3    }
{
    b:INT == a;
    a:INT == 5
}
    a + b
```

evaluates to 8 and *not* to 10, since the reference to `a` within the definition of `b` takes its value from the binding of the surrounding block.

3.10 Function Definitions

Functions are defined by statements of the form

Ident`[`*Ident*`, ...]` `:` *SimpleType*`[`*SimpleType*`, ...]` `==` *Expression*

The first *Ident* is the name of the function to be defined, the other *Idents* within the brackets define the names of the formal parameters. The first *SimpleType* denotes the type of the function's result value, the following ones the types of the parameters in the order of their appeareance within the brackets. The *Expression* specifies the result value of the function.

There is also a second form of function definitions for functions with *named type signatures*:

Ident`[`*Ident*`, ...]` `:` *SimpleType* `==` *Expression*

In this case, *SimpleType* denotes the name of a *function type* that specifies the types of the parameters as well as the type of the result. For further explanation on function types, see Section 3.11.

The following statements present some of the not very exciting but inevitable examples of function definitions how they can be found in any textbook on functional programming:

```
Fac[n]: INT[INT] ==  -- Factorial
  IF n = 0
    THEN 1
    ELSE n * Fac[n-1];

Fib[n]: INT[INT] ==  -- Fibonacci, Sledge-Hammer Version
  IF n <= 2
    THEN 1
    ELSE {
          a: INT == Fib[n-1];
          b: INT == Fib[n-2]
         }
          a + b;

Fibo[n]: INT[INT] == -- Fibonacci, Tail-Recursive Version
  {
   -- Local Auxiliary Function
   Iter[a,b,n]: INT[INT,INT,INT] ==
     IF n = 1
      THEN a
      ELSE Iter[b,a+b,n-1];
  }
   Iter[1,1,n]
```

As shown in these examples, the function *bodies* (i.e. the expressions on the right hand sides of function definitions) may include local definitions, also function definitions themselves. The formal parameters of a function can be interpreted as constant names that are locally defined, which overrides any binding of these names outside the function.

As usual, function definitions obey the lexical scoping discipline (see Section 3.9). Names that are used within the body of a function, but have neither been introduced as parameters nor by local bindings, take their values from the lexical enclosing environment. E.g. in the definitions

```
a: INT == 7;

F[m,n]: INT[INT,INT] ==
 {
  G[i,j]: INT[INT,INT] == i*j+n
 }
  G[m,a]
```

the reference to `a` within the body of `F` takes its value 7 from the global environment, while the reference to `n` within the body of `G` takes its value from the formal parameter of `F`.

As already mentioned several times, the scope of any name starts at the point of its definition, therefore functions have to be defined, before they can be used. In order to allow mutually recursive functions, EVE knows as a special form the *forward function definition*:

Ident`[`*Ident*`, ...] :` *SimpleType*`[`*SimpleType*`, ...]`

respectively

Ident`[`*Ident*`, ...] :` *SimpleType*

for functions with named type signature (see Section 3.11).

A forward definition statement makes public just the signature of a function, i.e. its head and its type, but not the body of the function itself. From the point of this statement on, references to this function are allowed but the actual function definition has to take place within the same level of scope.

E.g. in the definition sequence

```
G[x]: INT[INT];     -- Forward Definition of G

F[x]: INT[INT] ==   -- Definition of F
  IF ODD[x]
    THEN F[x-1]
    ELSE G[x/2];    -- Use of G

G[x]: INT[INT] ==   -- Actual Definition of G
  IF x = 0
    THEN 1
    ELSE F[x+2];    -- Use of F
```

the forward reference to `G` is resolved by the use of a forward definition statement.

3.11 Higher Order Functions

In EVE, functions are first order citizens like any other types of objects, i.e. they can be stored in data structures, passed to other functions as arguments, returned as results of functions, etc.

For this purpose, EVE provides as a fourth type constructor[13], the *function constructor*. Its general form is

SimpleType`[`*SimpleType*`, ...,` *SimpleType*`]`

This constructor creates the type that includes all those functions, whose signatures (i.e. the number and the types of parameters as well as the result type) equal the given signature (this type is known as *function domain* in denotational semantics). Examples for the definition of function types are

```
IntFun   ==  INT[INT];  -- Unary integer function, e.g. INEG
IntTest  ==  BOOL[INT]; -- Test integer function, e.g. ODD
CharInt  ==  CHAR[INT]; -- Conversion functions, e.g. CHR
```

The function constructor enables us to write higher order functions, i.e. functions that take other functions as arguments, e.g.

[13] The three other ones have been described in Section 3.3.

```
IntFun  == INT[INT];
IntList == LIST OF INT;

IMap[fun,list]: IntList[IntFun,IntList] ==
  IF NULL[list]
    THEN list
    ELSE fun[list.HEAD]::IMap[fun,list.TAIL];
```

This function `IMap` takes a function and a list as arguments and applies the function to every element of the list (it "maps the function to the list"). The generated result values are successively entered into a list which is returned as the result of the function.

Using the following function definition with a named type signature,

```
Twice[x]: IntFun == x+x
```

the expression

```
IMap[Twice,IntList[1,2,3,4,5]]
```

maps the function `Twice` to a list of integers and returns the list of the doubled values. But now, if we want to use `IMap` to apply the function

```
Square[x]: INT[INT] == x*x
```

to a list of integers, we have to take care. In contrast to our policy of giving all objects explicitely *named types*, `Square` has, like most functions, an *anonymous type* specified by its signature[14]. Our definition of type equality (see Section 3.3) forces us to convert a function with anonymous type signature to a named type signature. This may be done by the application of a *conversion function* that has the name of its associated type. E.g. the expression

```
IMap[IntFun[Square],IntList[1,2,3,4,5]]
```

maps the converted function `Square` to a list of integers and returns the list of the squared values.

Functions can also be generated as results of function calls, e.g. the statement

```
Mult[n]: IntFun[INT] ==
  {M[i]: INT[INT] == n*i}
  IntFun[M];
```

defines a function `Mult`, which takes an integer `n` as argument and returns a unary function, which multiplies a given integer `i` with `n`.

Functions can also be renamed or stored in data structures, e.g. the statements

[14] The reason for this is merely pragmatic, since it would be very tedious to define explicit type names for every kind of function signature.

```
FunPair == TUPLE OF a: IntFun, b: IntFun;

A: IntFun == IntFun[IADD]; -- Renaming of IADD
B: FunPair ==              -- Pair of ISUB and IDIV
  FunPair[IntFun[ISUB],
          IntFun[IDIV]];
```

define the name A as a *synonym* for IADD[15], and B as a tuple containing the functions ISUB and IDIV.

Now the existence of the prefix versions of the integer operators and comparison operators becomes clear. Their only purpose is to serve as the first order counterparts of the infix operators when using higher order functions.

However, there are several predefined functions that are *not* first-order objects and may not be given as arguments to higher-order functions:

1. The generic **list functions** CONS, HEAD, HEAD and TAIL are not first-order objects.
2. All user-defined **data constructors** (for tuples, unions and function types) are not first-order objects.
3. The generic **error test function** FAILED is not a first-order object.

This restriction imposed by the non-polymorphic type system of EVE is actually not very serious: if a well-typed user-defined function is put around a generic function, this function may be freely used as first-order object, e.g.

```
IntList == LIST OF INT;
ListFun == IntList[INT, IntList];

IntCons[int,list]: ListFun ==
  CONS[int,list];
IntApply[fun, int, list]: IntList[ListFun, INT, IntList] ==
  fun[int,list]
```

3.12 Strictness Properties

The strictness properties of functions and data structures may have strong impacts on the runtime behaviour of a program, especially on its termination and on its reaction on runtime errors.

In this section, we will concentrate on the non-termination interpretation of strictness i.e. $\perp$ will stand for the non-terminating value. However, if not explicitly stated in the next section, the propositions of this section also hold for the error interpretation.

In EVE there exist a few simple rules for the strictness of functions:

1. **User-defined functions** are *non-strict* in all of their arguments, unless the strictness in an argument is forced by the body of the function. This is the case, if the evaluation of the body requires calls of operators or functions that are strict in this argument.

[15] The term synonym already indicates that any reference to A is actually a reference to IADD. Function renamings do *not* create new functions!

2. **Predefined functions and operators** are *strict* in all of their arguments. This rule may be overridden by some of the following rules.
3. The **application of functions** to arguments is *strict* in the function to be applied.
4. The **boolean operators** & and | (and their associated functions `AND` and `OR`) are only *strict in their first argument*.
5. The infix **list construction operator** `::` and its prefix version `CONS` are *non-strict* in both of their arguments.
6. All **list, tuple, and union constructors** are *non-strict* in all of their arguments.
7. The **if-then-else construct** is only *strict in its first argument*, i.e. the boolean test condition.
8. The **case construct** is only *strict in its first argument*, i.e. the union expression.

Some examples may help to understand these rules: we define the non-terminating function

```
Forever[x]: INT[INT] == Forever[x]
```

and the following functions

```
Add0[x]: INT[INT] == 1+0;   -- Not strict in x

AddX[x]: INT[INT] == 1+0*x; -- Strict in x

Branch[x]: INT[INT] ==      -- Strict in x
  IF x=0                       -- = is strict in x
    THEN 1                     -- THEN-branch not strict
    ELSE Forever[x];           -- ELSE-branch strict in x

ConsL[x]: IntList[INT] ==   -- Not strict in x
  x :: IntList[0];             -- :: not strict in first arg

ConsR[x]: IntList[INT] ==   -- Not strict in x
  1 :: IntList[x];             -- :: not strict in secnd arg
```

Then the following propositions hold:

```
Forever[0]          =  ⊥
Add0[Forever[0]]    =  1
AddX[Forever[0]]    =  ⊥
Branch[0]           =  1
Branch[Forever[0]]  =  ⊥
ConsL[Forever[0]]   =  IntList[⊥,0]
ConsR[Forever[0]]   =  IntList[1,⊥]
```

where $\perp$ denotes a value that will never be computed.

However, although the expression

```
ConsL[Forever[0]]
```

Operator	*Error*	*Description*
`+`, `-`, `*`, `/`, `ABS`	Overflow	Result outside the range `MININT`–`MAXINT`
`/`, `%`	Zero divide	Division (resp. Remainder) by zero
`CHR`	No ASCII value	Argument outside the range 0–255
`HEAD`, `TAIL`	Empty list	Application to an empty list

Table 3.10: Runtime Errors

returns a result (namely the list with the first value undefined and the second value 0), the expression

```
ConsL[Forever[0]].HEAD
```

will not return a value, since the head of the constructed list will never be computed!

In EVE, all *compound data structures* (lists, tuples and also unions) are *non-strict* in the sense, that the structure as a whole may be used while components of the structure are still undefined.

The consequences of the strictness property on the error behaviour of a program will be discussed in the next section.

3.13 Error Behaviour

EVE handles runtime errors by the use of *error values*. Whenever some operation is not able to return a proper result, instead of the result the predefined constant `ERROR` is returned. `ERROR` is a generic constant, i.e. it is element of every data type that can be defined in EVE and therefore compatible with every type.

All strictness rules of the previous section also hold for error values, with two exceptions:

1. For all **predefined binary functions and operators** that are *strict* in both of their arguments, the "non-terminating $\perp$" has precedence over the "error $\perp$". If one argument is non-terminating and the other one is an error value, the computation of the function or operator does *not* terminate.
2. The **error test function** `FAILED` is only *strict* for the "non-terminating $\perp$" but not for the "error $\perp$". If applied to an error value, `FAILED` returns a valid boolean value.

`ERROR` may only be produced in those cases that are listed in Table 3.10.

If a function or operator is applied to `ERROR`, the result of the operation depends on the strictness, respectively non-strictness, of the function[16]:

- If the function is strict in the `ERROR` argument, the result will also be the `ERROR` value.
- If the function is not strict in this argument, it will compute some sensible result value.

The generic function `FAILED` is used to detect, whether the evaluation of some value value has been successful or whether the result is the `ERROR` constant, i.e.

[16] See Section 3.12 for a detailed discussion on this topic.

```
FAILED[ERROR] = TRUE.
```

However, FAILED is no first-order object and cannot be used as an argument to higher order functions (see Section 3.11).

The following examples demonstrate some cases, in which ERROR may be returned as result:

```
FAILED[1/0]                        =  TRUE
FAILED[IntList[1/0]]               =  FALSE
FAILED[CHR[1234]]                  =  TRUE
FAILED[IF 0=0 THEN 1 ELSE 1/0]     =  FALSE
FAILED[IF 1=0 THEN 1 ELSE ERROR]   =  TRUE
FAILED[(1/0)::IntList[1]]          =  FALSE
```

Take care, although 1/0 is the error value, IntList[1/0] returns a proper result, namely a list whose only element is the error value!

User-defined functions are basically non-strict, i.e. they may return proper results, although one of their arguments is invalid, e.g. using the definition

```
F[x,y]: INT[INT,INT] ==
  IF x=0
    THEN 1
    ELSE y;
```

the expression F[0,1/0] returns the result value 1, although the second argument to the function is the ERROR value.

3.14 A Programming Example

As an illustration of some of the facilities of EVE, we show the design of a function that performs symbolic differentiation of algebraic expressions. This example is taken from [Abelson and Sussman, 1985], where the program has been written in Scheme, a functional dialect of Lisp.

We want to write a function Derive that takes as arguments an algebraic expression and a variable and returns the derivative of the expression with respect to the variable. The algorithm itself is rather simple, but we want to direct our attention to the way EVE supports the solution of the problem.

First of all, how can we represent algebraic expressions? Algebraic expressions may be constants or variables, as well as sums or products of previously constructed expressions[17]. This description already gives us some hints on how to define expressions:

- Constants may be simply encoded as integers.
- Variables may be represented by character strings.
- Sums and Products are pairs, i.e. tuples, of expressions.
- Since an expression may be either a constant, a variable, a sum or a product, we will represent it as a union consisting of one of these four variants.

```
----------------------------------------------------------
--
-- Symbolic Differentiation (Abelson & Sussman, page 104)
--
----------------------------------------------------------

Constant == INT;      -- Constants

Variable == STR;      -- Variables

Exp;                  -- Forward Definition Exp

SumExp == TUPLE OF    -- Sum Expressions
  addend: Exp,        -- addend + augend
  augend: Exp;

ProdExp == TUPLE OF   -- Product Expressions
  multiplicand: Exp,  -- multiplcand * multiplier
  multiplier:   Exp;

Exp == UNION OF       -- Expressions
  Const == Constant,
  Var   == Variable,
  Sum   == SumExp,
  Prod  == ProdExp;
```

Figure 3.1: Symbolic Derivation (Type Definitions)

Figure 3.1 shows the several statements needed for these type definitions.

Let's take a look at the order of these statements: `SumExp` and `ProdExp` both refer to `Exp`, therefore `Exp` has to made public by a forward type definition (see Section 3.3). Vice versa, `Exp` refers to `SumExp` and `ProdExp` too (mutual recursion), but both of these types have been introduced before.

Now we have to introduce some functions doing auxiliary work in the process of computing the derivative of an expression.

- First, we need a function `VarEq` that compares two variables. This task is not very difficult, since variables are represented by character strings, which can be compared by the sequential comparison of their elements.
- Then we need two functions `MakeSum` and `MakeProd` constructing the sum and the product of two expressions. This task gets a little bit complicated, because we want these functions to perform some simplifications. E.g. the expression $0 + x$ should not remain a sum expression, but be simplified to x itself. Similar simplifications can be done for $0 * x$ and $1 * x$. In addition, if both parts of a sum or a product are constants, the result value should be computed directly without the construction of a compound expression.

Figure 3.2 shows, how these functions can be programmed.

An interesting point is the definition of the function `VarEq`. `VarEq` does not call the function `StrEq` that performs the actual comparison of strings. On the contrary, `VarEq` *is* the function `StrEq`, or in other words, it is a synonym of this function, so any reference to `VarEq` is actually a reference to `StrEq`. Renamings of functions are (like renamings of types) a very powerful tool for program and data abstraction, since any changes in the representation of an object will affect just some small parts of the program.

The actual derivation function, shown in figure 3.3, can now be written straightforward. Depending on the type of the expression, we perform one of the well-known differentiation rules for constants, variables, sums and products.

Our derivation program is still very simple with respect to the simplification of algebraic expressions. This is a very complex algorithmic problem, because, among other reasons, a form that is simplest for one purpose may not be for another. Therefore in "real" applications, combinations of different strategies have to be applied.

3.15 Syntax Definition

3.15.1 Notation

We define EVE's lexical structure and syntax using a form of *Extended Backus-Naur Notation*:

Terminal Symbols (i.e. the tokens of EVE) are written in `typewriter` style.

Nonterminal Symbols (i.e. syntactic entities defined by syntax rules) appear in *italic* style.

Meta symbols (i.e. symbols that structure syntax rules) appear in simple Roman style:

= seperates a nonterminal symbol that is being defined from its definition.

[17] We restrict ourselves to these kinds of expressions. It is rather simple to extend them to expressions that also contain subtractions, divisions, etc.

```
-- Test Strings for Equality
StrEq[s1,s2]: BOOL[STR,STR] ==
  IF NULL[s1]
    THEN NULL[s2]
  ELSE IF NULL[s2]
    THEN NULL[s1]
  ELSE
    (s1.HEAD = s2.HEAD) & StrEq[s1.TAIL,s2.TAIL];

-- Test Variables for Equality
BoolVar == BOOL[Variable,Variable];
VarEq: BoolVar == BoolVar[StrEq];

-- Construct the Sum of Two Expressions
MakeSum[exp1,exp2]: Exp[Exp,Exp] ==
  CASE exp1
    {c1: Const} CASE exp2
      {c2: Const} Exp$Const[c1+c2]   -- exp1 and exp2 constant
      ELSE IF c1 = 0                 -- only exp1 constant
        THEN exp2                    --   exp1 = 0
        ELSE Exp$Sum[exp1,exp2]      --   exp1 # 0
    ELSE CASE exp2
      {c2: Const} IF c2 = 0          -- only exp2 constant
        THEN exp1                    --   exp2 = 0
        ELSE Exp$Sum[exp1,exp2]      --   exp2 # 0
    ELSE
      Exp$Sum[exp1,exp2];            -- No constants

-- Construct the Product of Two Expresssions
MakeProd[exp1,exp2]: Exp[Exp,Exp] ==
  CASE exp1
    {c1: Const} CASE exp2
      {c2: Const} Exp$Const[c1,c2]   -- exp1 and exp2 constant
      ELSE IF c1 = 0 THEN
        Exp$Const[0]                 -- exp1 = 0
      ELSE IF c1 = 1 THEN
        exp2                         -- exp1 = 1
      ELSE
        Exp$Prod[exp1,exp2]          -- exp1 # 0,1
    ELSE CASE exp2
      {c2: Const} IF c2 = 0 THEN
        Exp$Const[0]                 -- exp2 = 0
      ELSE IF c2 = 1 THEN
        exp1                         -- exp2 = 1
      ELSE
        Exp$Prod[exp1,exp2]          -- exp2 # 0,1
      ELSE
        Exp$Prod[exp1,exp2];         -- No constants
```

Figure 3.2: Symbolic Derivation (Auxiliary Functions)

```
--------------------------------------------------------------
-- Derive an expressions exp with respect to the variable var
--------------------------------------------------------------

Derive[exp,var]: Exp[Exp,Var] ==

  CASE exp

    {c: Const}            -- Constants
      Exp$Const[0]

    {v: Var}              -- Variables
      IF VarEq[v,var]
        THEN Exp$Const[1]
        ELSE Exp$Const[0]

    {s: Sum}              -- Sums
     {
      first: Exp == Derive[s.addend,var];
      secnd: Exp == Derive[s.augend,var]
     }
      MakeSum[first,secnd]

    {p: Prod}             -- Products
     {
      first: Exp == p.multiplicand;
      secnd: Exp == p.multiplier;
      prod1: Exp == MakeProd[first,Derive[secnd,var]];
      prod2: Exp == MakeProd[Derive[first,var],secnd]
     }
      MakeSum[prod1,prod2];
```

Figure 3.3: Symbolic Derivation (Main Function)

( ) group syntactic items in order to form a larger item[18].

[ ] enclose syntactic items that are optional, i.e. that may appear once or not at all in that place[19].

{ } enclose syntactic items that may be repeated zero or more times in that place[20].

| seperates two alternative syntactic items, one of which must appear in that place.

. denotes the end of a syntax rule.

Each nonterminal symbol is defined by a **Syntax Rule** of the form

Nonterminal = *Definition* .

3.15.2 Input File

Input files may consist of all printable characters of the *US-ASCII character set* (see Table 3.11). In addition, the following invisible ASCII characters *(white space)* are allowed:

- ␣ (ASCII code 032) is the *space character*.
- ⟨*TAB*⟩ (ASCII code 009) is translated to a *single space* in all places it occurs (even in text constants).
- ⟨*LF*⟩ (ASCII code 010) denotes the *end of a line*.
- ⟨*FF*⟩ (ASCII code 012) is translated into the ⟨*LF*⟩ character in all places it occurs (also in text constants).
- ⟨*CR*⟩ (ASCII code 013) is translated into the ⟨*LF*⟩ character in all places it occurs (also in text constants).

All other characters are *illegal* (even within text constants or comments).

3.15.3 Lexical Structure

The following syntax rules define the lexical structure, i.e. the structure of EVE tokens:

Letter	=	`a` | `b` | ... | `z` | `A` | `B` | ... | `Z` .
Digit	=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` .
Ident	=	*Letter* { *Letter* | *Digit* }.
Integer	=	*Digit* { *Digit* }.
Number	=	*Integer* .
SpecialChar	=	*Any character of Table 3.6 except* \' *and* \" .
SingleChar	=	*Any printable character of the ASCII set including* ␣ *except* ' *and* \ | *SpecialChar* | " | \' .
StringChar	=	*Any printable character of the ASCII set including* ␣ *except* " *and* \ | *SpecialChar* | ' | \" .
Character	=	' SingleChar ' .
String	=	" { *StringChar* } " .

[18]Do not mix up the meta symbols ( ) with the symbols `( )` of the EVE language!

[19]Do not mix up the meta symbols [ ] with the symbols `[ ]` of the EVE language!

[20]Do not mix up the meta symbols { } with the symbols `{ }` of the EVE language!

	0	*16*	*32*	*48*	*64*	*80*	*96*	*112*
0	⟨*NUL*⟩	⟨*DLE*⟩	␣	`0`	`@`	`P`	`‘`	`p`
1	⟨*SOH*⟩	⟨*DC1*⟩	`!`	`1`	`A`	`Q`	`a`	`q`
2	⟨*STX*⟩	⟨*DC2*⟩	`"`	`2`	`B`	`R`	`b`	`r`
3	⟨*ETX*⟩	⟨*DC3*⟩	`#`	`3`	`C`	`S`	`c`	`s`
4	⟨*EOT*⟩	⟨*DC4*⟩	`$`	`4`	`D`	`T`	`d`	`t`
5	⟨*ENQ*⟩	⟨*NAK*⟩	`%`	`5`	`E`	`U`	`e`	`u`
6	⟨*ACK*⟩	⟨*SYN*⟩	`&`	`6`	`F`	`V`	`f`	`v`
7	⟨*BEL*⟩	⟨*ETB*⟩	`'`	`7`	`G`	`W`	`g`	`w`
8	⟨*BS*⟩	⟨*CAN*⟩	`(`	`8`	`H`	`X`	`h`	`x`
9	⟨*HT*⟩	⟨*EM*⟩	`)`	`9`	`I`	`Y`	`i`	`y`
10	⟨*LF*⟩	⟨*SUB*⟩	`*`	`:`	`J`	`Z`	`j`	`z`
11	⟨*VT*⟩	⟨*ESC*⟩	`+`	`;`	`K`	`[`	`k`	`{`
12	⟨*FF*⟩	⟨*FS*⟩	`,`	`<`	`L`	`\`	`l`	`|`
13	⟨*CR*⟩	⟨*GS*⟩	`-`	`=`	`M`	`]`	`m`	`}`
14	⟨*SO*⟩	⟨*RS*⟩	`.`	`>`	`N`	`^`	`n`	`~`
15	⟨*SI*⟩	⟨*US*⟩	`/`	`?`	`O`	`_`	`o`	⟨*DEL*⟩

Table 3.11: The US-ASCII Character Set

3.15.4 Context-Free Syntax

The following syntax rules define the context-free syntax of the Eve language in four sections:

- Types,
- Definitions,
- Expressions,
- Programs.

Types

IdentList = *Ident* { , *Ident* } .
SimpleType = *Ident* .
TypeList = *SimpleType* { , *SimpleType* } .
Field = *Ident* : *SimpleType* .
FieldList = *Field* { , *Field* } .
Variant = *Ident* `==` *SimpleType* .
VariantList = *Variant* { , *Variant* } .
ListType = `LIST OF` *SimpleType* .
TupleType = `TUPLE OF` *FieldList* .
UnionType = `UNION OF` *VariantList* .
FunctionType = *SimpleType* `[` *TypeList* `]` .
Type = *SimpleType* | *ListType* | *TupleType* | *UnionType* | *FunctionType* .

Definition Statements

```
TypeDefinition      = Ident [ == Type ].
ConstantDefinition  = Ident : SimpleType [ == Expression ].
FunctionDefinition  = Ident [ IdentList ] : ( FunctionType | SimpleType ) [ == Expression ].
Definition          = [TypeDefinition | ConstantDefinition | FunctionDefinition ].
DefinitionSequence  = Definition { ; Definition }.
```

Expressions

```
Expression    = ListExpr | { DefinitionSequence } Expression | IF Expression THEN Expression
                ELSE Expression | CASE Expression { Field } Expression { { Field } Expression
                } [ ELSE Expression ].
ListExpr      = CompoundExpr { :: Factor }.
CompoundExpr  = SimpleExpr [ Relation SimpleExpr ].
Relation      = = | # | < | <= | > | >= .
SimpleExpr    = [ + | - ] SimpleExpr { AddOp Term }.
AddOp         = + | - | | .
Term          = Term { MulOp Factor }.
MulOp         = * | / | % | &  .
Factor        = Number | Character | String | Designator { ArgumentList | Selector } | ( Ex-
                pression ) | ~ Factor .
Designator    = Ident [ $ Ident ].
ArgumentList  = [ [ Expression { , Expression } ] ] .
Selector      = . Ident .
```

Programs

```
Program = DefinitionSequence .
```

3.15.5 Keywords

There are eight keywords that are reserved and must not be redefined:

```
CASE  ELSE  IF     LIST
OF    THEN  TUPLE  UNION
```

3.16 Predefined Identifiers

Predefined identifiers may be redefined by the user, although it is strongly recommended not to do so. There are three classes of predefined identifers:

- Types,
- Constants,
- Functions.

3.16.1 Predefined Types

EVE has four predefined types thta are shown in Table 3.12.

Name	*Description*
INT	Integer Numbers
BOOL	Boolean Values
CHR	Characters
STR	Strings

Table 3.12: Predefined Types

Name	*Type*	*Description*
MININT	INT	Lower bound of integer numbers
MAXINT	INT	Upper bound of integer numbers
TRUE	BOOL	Boolean constant
FALSE	BOOL	Boolean constant
ERROR	*Any*	Error value

Table 3.13: Predefined Constants

3.16.2 Predefined Constants

Table 3.13 shows the predefined constants of EVE.

The constant ERROR is a generic constant, i.e. it is element of any type and therefore compatible with every type.

3.16.3 Predefined Functions

Functions on Atomic Types

Tables 3.14, 3.15 and 3.16 show the predefined functions working on integers, booleans and characters, respectively.

Name	*Type*	*Description*
ADD	INT[INT,INT]	Addition
SUB	INT[INT,INT]	Subtraction
MUL	INT[INT,INT]	Multiplication
DIV	INT[INT,INT]	Division
MOD	INT[INT,INT]	Remainder
NEG	INT[INT]	Negation
ABS	INT[INT]	Absolute Value
ODD	BOOL[INT]	Is number odd?

Table 3.14: Predefined Integer Functions

Name	*Type*	*Description*
AND	BOOL[BOOL,BOOL]	Conjunction
OR	BOOL[BOOL,BOOL]	Disjunction
NOT	BOOL[BOOL]	Negation

Table 3.15: Predefined Boolean Functions

Name	*Type*	*Description*
CAP	CHAR[CHAR]	Capitalize a letter
CHR	CHAR[INT]	Character from ASCII code
ORD	INT[CHAR]	ASCII code of a character

Table 3.16: Predefined Character Functions

Comparison Functions

Table 3.17 describes the functions that can be used to compare objects of atomic type (i.e. integers, booleans and characters). For each kind of comparison operator, there exist *three functions*, one for each atomic type. In order to get the actual function names, one has to replace the asterisk * in the names by `I (INT)`, `B (BOOL)`, or `C (CHAR)`.

List Functions

Table 3.18 refers to the generic type definition

AnyList `== LIST OF` *Any*.

However, all of these functions are *not* first-order objects and may not be used as arguments of higher-order functions (see Section 3.11).

Error Test Function

The generic error test function shown in Table 3.19 can be applied to expressions of *any* type.

Function	*Type*	*Description*
*LT	BOOL[*Atomic,Atomic*]	Less Than
*LE	BOOL[*Atomic,Atomic*]	Less or Equal
*EQ	BOOL[*Atomic,Atomic*]	Equal
*NE	BOOL[*Atomic,Atomic*]	Not Equal
*GE	BOOL[*Atomic,Atomic*]	Greater or Equal
*GT	BOOL[*Atomic,Atomic*]	Greater Than

Table 3.17: Predefined Comparison Functions

Function	*Type*	*Description*
CONS	*AnyList*[*Any*, *AnyList*]	Construct a list
HEAD	*Any*[*AnyList*]	Select the head of a list
TAIL	*AnyList*[*AnyList*]	Select the tail of a list
NULL	BOOL[*AnyList*]	Is a list empty?

Table 3.18: Predefined List Functions

Function	*Type*	*Description*
FAILED	BOOL[*Any*]	Test for ERROR

Table 3.19: Error Test Function

However, FAILED is *not* a first order object and may not be used as an argument of a higher-order function (see Section 3.11).

Chapter 4

The ADAM Abstract Dataflow Machine

This chapter describes the design of the ADAM Abstract Dataflow Machine. The ADAM architecture represents a hybrid concept combining features from conventional register architectures and from pure dataflow computers. After an introduction into the basic idea of our approach, we present the architecture in a top-down fashion: first the global structure of the machine is illustrated, then we describe the components of an ADAM module in more detail. In Appendix A, we present the operational behaviour of the machine by defining an interpreter that simulates the machine in operation. Finally, in Appendix B the semantics of the individual basic instructions is defined.

A general overview over the ADAM has been presented in [Schreiner, 1990] and [Schreiner, 1991].

4.1 Basic Idea

The basic idea of our approach is as follows. We take a mathematical expression, e.g.

$$\begin{array}{ll} (a+b)*(-y) & \\ \textbf{where} & a = -x \\ & b = x/y \end{array}$$

This expression can be viewed as the body of a function $f(x, y)$ with formal parameters x and y and is translated in a dataflow graph as depicted in Fig. 4.1.

In pure dataflow architectures, this dataflow graph is represented by a dataflow program of the following kind where each instruction consists of an operation code and the address of its successor instruction:

```
0:  COPY 2 3.l
1:  COPY 3.r 4
2:  NEG 5.l
3:  DIV 5.r
4:  NEG 6
5:  ADD 6
6:  MUL 7
```

Figure 4.1: A Dataflow Graph

Each node of the graph is translated into one instruction of the program where the arcs of the graph are represented by the instructions' successor addresses. If instructions are only allowed to have one successor, branches have to be translated into explicit `COPY` instructions duplicating their input values. If the successor is a binary instruction, the predecessor must also specify which of the "input ports" of the successor is referenced.

In most dataflow machines, values are transmitted between different instructions by pieces of information, called "tokens" that flow along the arcs of the dataflow graph. For instructions with more than one input arcs, special hardware mechanisms have to handle the synchronization of the input tokens, i.e. a "rendezvous" between the tokens has to be organized for a proper execution of the instruction.

In the ADAM model, the arcs of the dataflow graph denote registers and the nodes represent two-address instructions operating on these registers:

`0:`	`COPY`	x_1, x_2, 2
`1:`	`COPY`	y_1, y_2, 4
`2:`	`SPLIT`	3, 6
`3:`	`NEG`	x_1, x_1, 8
`4:`	`SPLIT`	5, 6
`5:`	`NEG`	y_2, y_2, 10
`6:`	`WAIT`	e_0, 7
`7:`	`DIV`	x_2, y_1, 8
`8:`	`WAIT`	e_1, 9
`9:`	`ADD`	x_1, x_2, 10
`10:`	`WAIT`	e_2, 11
`11:`	`DIV`	x_1, y_2, 12

Every instruction operates on two registers where the first register receives the result value. After an instruction has been executed, the control flow is routed by the address field of the instruction, i.e. each instruction performs a jump to its successor. `SPLIT` instructions fork the actual control flow into two parallel paths starting their execution at the specified successors. The execution of an instruction sequence may be unhinderedly continued as long as no `WAIT` instruction is encontered.

Each `WAIT` guards a binary *event* encoded in the first field of this instruction. Each event may have one of the values "occurred" or "not occurred", initially no event has occurred. If instruction execution reaches a `WAIT` whose event has not yet occurred, the instruction sequence is interrupted. However, the status of the referred event is set to "occurred". The next time, this event is tested during the execution of an instruction sequence, the sequence may unhinderedly pass the `WAIT`.

`SPLIT` and `WAIT` control the generation of parallel processes and the synchronization between them, they are the only instructions related to parallelism. However, they suffice to control the program execution in such a way that dataflow semantics is simulated.

We want to emphasize the essential difference between the pure dataflow model and our approach: In dataflow architectures, all data physically "flow" between instructions and, by this, control the computation. In the ADAM processor, sequences of instructions are executed without any data flowing between them. Communication between different instructions is done via shared registers, synchronization between parallel instruction sequences are explicitely handled by `WAIT` statements.

This approach has two essential advantages:

1. In dataflow machines, all data referenced by more than one instructions have to be **explicitly duplicated** and sent to the required instructions, even if they belong to the same activation of a function and reside on the same processor.

   In our model, parallel program paths may **share the data** held in common registers, an explicit duplication can be avoided in most cases. The compiler may optimize register allocation such that the overall number of registers required is reduced to an absolute minimum.

2. In dataflow architectures, the synchronization model is fixed in hardware. Parallel program paths have to be generated and finally synchronized even if the parallelism is not really utilized. This is true for most approaches, since the distribution of tasks to processors is usually handled on the level of whole functions, i.e. parallel paths within a function body are executed on the same processor.

   In our concept, a compiler may, by simple code analysis, detect in which cases parallel paths may be safely (i.e. without changes to the dataflow semantics) sequentially executed and then **serialize the code** by the use of `WAIT` statements. Sequentialization is not allowed for the bodies of non-strict functions, i.e. functions that may return values even if not all arguments have yet arrived. In general, a process called *strictness analysis* is necessary to detect to which extent function bodies may be safely serialized.

Using both optimizations, register sharing and serialization, we may optimize the above program in the following way:

0:	WAIT	e_0, 1
1:	COPY	x_2, x_1, 2
2:	DIV	x_2, y_1, 3
3:	NEG	x_1, x_1, 4
4:	ADD	x_1, x_2, 5
5:	NEG	y_1, y_1, 6
6:	MULT	x_1, y_1, 7

The `WAIT` statement is only responsible for synchronizing the parallel tasks delivering their arguments in registers x_1 and y_1. All other instructions may be sequentially executed requiring only one additional register x_2.

After this short introduction into the basic principles of our approach, the following sections present the ADAM architecture in detail.

4.2 Global Structure

The ADAM is a parallel machine with distributed memory. It consists of several equally constructed modules that are connected by a network of unbuffered uni-directional channels. Each module of the network has several input and output channels by which it can receive *tokens*, i.e. pieces of information, from other modules and send tokens to them. These channels represent the only communication medium of the ADAM, there is no global memory that the modules might have access to. One special module, the *master*, is connected to peripheral devices for input and output. The master handles all kinds of interactions with the outside world. In particular it is responsible for loading and distributing the program to be executed, for triggering the execution of the program and for delivering the result values of the computation.

Figure 4.2 shows a sample network of ADAM modules (each double-arrow represents two channels for communication in both directions).

The master module is built of the following components depicted in Figure 4.3:

1. **The Input/Output Interface**

   This interface is the master's connection to the outside world, i.e. to the host computer by that input and output is handled. The interface communicates by a special protocol with a server program that is sited on the host and is responsible for the access to required peripheral devices. The program to be executed is loaded to the ADAM by this interface while the results of the computation are printed on the screen and written into a file on the host.

2. **The Network Interface**

   This interface represents the master's connection to the network of worker modules. It is linked between two worker modules and is absolutely transparent with respect to network communication. Any token received from one side and not destined for the master is simply forwarded to the other side. However, the interface channels tokens from the master into the net and removes tokens destined for the master from the net.

3. **The Result Generator**

   This unit first triggers the execution of the program in the ADAM network and then demands the results of the computation from the network. Since the values received do not have any tags that indicate their types, their interpretation has to be directed by a *type descriptor file* that is loaded via the input/output interface. If this description e.g. indicates that the result type of a computation is a list of integer tuples, the result generator has to demand the delivery of all tuple references stored in the list and of all tuple elements, respectively. The received values are converted into printable form and then forwarded to the input/output interface.

4. **The Statistics Generator**

   This unit receives several types of statistics tokens that are generated by the worker modules. These messages give a rough impression on the overall load of the network and indicate which modules are more heavily loaded than others. The statistics tokens are converted into

Figure 4.2: The ADAM Network

Figure 4.3: An ADAM Master Module

Figure 4.4: An ADAM Worker Module

printable form and then forwarded to the input/output interface where they are written into a file. The load statistics may later be evaluated to achieve a better performance of the architecture by improving e.g. the load balancing among worker modules.

Each worker module of the ADAM consists of three components illustrated in Figure 4.4:

1. **The Arbiter**

   The *arbiter* is the only unit of a worker module with an interface to its neighbour modules. When the ADAM is booted, all arbiters are responsible for distributing the program to be executed over the network. Afterwards, they receive incoming tokens from the local units and from neighbour modules and route them to their correct destinations, either to one of the local units or to another module.

2. **The Processor Unit**

   The *processor unit* executes the given program and, thereby, performs the actual computation. It receives tokens (basically arguments for the local function activations) from and, vice versa, sends tokens (basically results of local function activations) to other processors in the network. Moreover, it communicates with its memory unit and those on other modules by request and result tokens.

3. **The Memory Unit**

Figure 4.5: The Arbiter

> The *memory unit* basically consists of a heap-organized store containing tuples of variable-size. These data structures may be allocated and referenced by incoming request tokens. A memory manager handles these requests and returns the result values (either integer values or tuple references). Tuples that are no more in use are reclaimed by a garbage collection mechanism.

The following sections provide an informal but detailed description of the various components of a worker module, i.e. of their structure, their behaviour and their interaction.

4.3 The Arbiter

The arbiter is that component of an ADAM module that is responsible for the communication between worker modules. Thus, it is the only unit on a worker module with an interface to the neighbour modules. Moreover, communication between the local units of a module is done via the arbiter, too. The construction of an arbiter is illustrated in Figure 4.5.

When the ADAM is booted, the first task of all arbiters is to distribute the program to be executed over all worker modules of the network. A local *boot table* determines along which paths of the network the program (split into several clusters) is forwarded. After all processor units have received the program, the main task of the arbiters is the correct routing of network tokens between the worker modules.

Each arbiter simultaneously listens on the external input channels that connect the module to its neighbours and on both of the internal input channels from the local processor and the local memory unit. If on one of these channels an input token is available, the arbiter receives the token and removes it from the channel. If more than one input channels are simultaneously ready for transmission, the arbiter non-derministically accepts one the tokens. The other input channels (i.e. the components on their opposite ends) are blocked until the arbiter receives the tokens they hold.

All tokens in the network have equal length and are constructed from the components

Address	*Type*	*Data*

that are interpreted as follows:

- *Address:*

  The address field contains the number of the network module that the token is destined for. This unique address is in the range

  $$0 \ldots \underline{\text{ModNum}}$$

  where <u>ModNum</u> is the number of modules in the network exclusively the master module. The master module always has the network address 0.

- *Type:*

  The type field describes how the piece of information contained in the *Data* field is to be interpreted and also determines whether to send the token to the processor or to the memory unit of a module. Basically, there are the following types of tokens:

  Value-Messages (short *Messages)* transmitting data values between processor units and memory units (see Section 4.4.5).

  Context-Messages transmitting to a processor the description of the instruction to that the result of some function activation is to be sent (see Section 4.4.5).

  Call-Requests transmitting the request for the activation of a new function instance to a processor unit (see Section 4.4.6).

  Allocation-Requests transmitting the request for the allocation of a new tuple to a memory unit (see Section 4.5.2).

  Read-Requests transmitting the request for reading the contents of some tuple field to a memory unit (see Section 4.5.2).

  Write-Requests transmitting the request for writing the contents of some tuple field to a memory unit (see Section 4.5.2).

  Submission-Requests transmitting the request for submitting the lifted arguments of a curried function whose description is stored in a memory unit (see Section 4.5.2).

  Reference-Messages reporting to a memory unit that a reference to some tuple has to be copied or has been deleted (see Section 4.5.4).

  Load-Reports transmitting to a processor or memory unit the report about the utilization of another processor or memory unit (see Sections A.3.8 and A.4.8).

Stop-Requests transmitting to all ADAM modules the signal that the program has delivered the desired result and that all units have to to terminate (see Sections A.3.7 and A.4.6).

- *Data:*

  The data field contains the actual piece of token information. For all types of tokens, the data field has equal length. However, the interpretation of its contents depends on the type tag of the token.

Incoming tokens are classified according to their type and destination adress and forwarded either to one of the local units on the module or to some neighbour module via one of the output channels that leave the module. Since the transmission of a token is not possible, before the receiver is ready to accept the token, the arbiter is connected to each of the leaving channels by a buffer store. This buffer is used to smooth differences in the communication speed of two neighboured modules. Thus, the sending arbiter is not blocked if the receiver is not yet ready to receive a token.

In order to route each incoming token to the correct output channel, each arbiter has a local routing table. This table contains one entry for each module of the whole network. It is indexed by the adress field of the token and provides the number of the output channel that the token has to be sent through. If the token carries the adress of the arbiter's own module, the arbiter sends the token according to its type either to the processor or to the memory unit of the module.

A more formal description of the operational behaviour of the arbiter is given in Appendix A.2.

4.4 The Processor Unit

The ADAM processor can be considered as a data-driven processor that efficiently supports internal parallelism and tolerates long memory latencies. However, there are essential differencess to the pure dataflow approach: The ADAM processor is actually a *control-driven* processor with several extensions that allow the efficient simulation of data-driven behaviour. Thus, the processor has a hybrid architecture combining features from conventional register-architectures and from true dataflow computers.

This design decision was made for two reasons:

1. We want to show how dataflow and controlflow are related amd that one model can be used to simulate the other. The dataflow paradigm may also be rather efficiently implemented on conventional parallel architectures.
2. This concept allows the more efficient execution of dataflow programs in general (see Section 4.1). Moreover, we may profit by the large amount of knowledge on compilation techniques for register architectures that has been gathered in the last decades.

For the same reasons, the idea of hybrid dataflow/von Neumann hardware architectures has most recently started to become a subject of international interest (see [Iannucci, 1988] and [Gao *et al.*, 1988]).

The ADAM processor basically consists of following components illustrated in Figure 4.6:

- **Execution Unit:**

  The execution unit is the heart of the processor performing the actual computations within the processor unit. Program execution is basically control-driven but constrained by special *synchronization instructions* that are used to express the data dependences within a program. This unit receives messages and requests from other modules and also sends messages and requests to processors and memories.

Figure 4.6: The Processor Unit

- **Program Store:**

  The program store is a contiguous area of memory that holds the machine code of the program. The code is partitioned into a sequence of *code blocks* each of which basically corresponds to one user-defined function.

- **Frame Store:**

  The frame store is a heap-organized store holding a set of *activation frames* each of which corresponding to one particular function activation. Each frame represents the actual state of its activation, i.e. it holds the contents of the regisers and the state of the events.

- **Context Store:**

  The context store is a table of activation descriptors, called *contexts*. Each context uniquely describes one function activation on the local processor unit by the base addresses of the activation's code block and of its register frame. These descriptors are used to keep references to activations as small as possible.

- **Trigger Store:**

  The trigger store contains a set of *instruction triggers* each of which is the description of a parallel task that is ready to be executed by has not yet been. Each time a new task is forked, a trigger for this task is put into this store that is the the source of internal parallelism on the processor.

- **Message Store:**

  The message store contains all *message tokens* that have been sent to the processor but have not yet been accepted by the execution unit. These tokens are used to transport pieces of information between processor and memory units. Each message may be handled without consuming any new processor resources.

- **Request Store:**

  The request store contains all *call requests* that have been sent to the processor element but have not yet been fulfilled by the execution unit. Each request asks the processor to create a new activation of a specific function, which requires the allocation of a new activation frame.

The following subsections provide a more detailed description of the processor components.

4.4.1 The Program Store

The program store (illustrated in Figure 4.7) is a linear adressed array that is filled with ADAM machine code[1] before program execution is started. All instructions have equal length and consist of the fields

OpCode	*OpWord*

where *OpCode* denotes the *operation code* of the instruction and *OpWord* represents its operand.

Instructions are grouped to so called *code blocks* each of which is basically the body of one user--defined functions. The first word in each code block is actually not an instruction but a *function descriptor* that consists of the fields

[1] Also some other values (e.g. constants) are deposited in the program store.

Figure 4.7: The Program Store

EventNum	*RegNum*

that are interpreted as follows:

- *EventNum*

  denotes the number of frame slots that are required to hold the *synchronization bits* of each activation of this code block. Each of these bits represents a synchronization event, i.e. a point of synchronization between two parallel instruction paths. Further details on this topic are discussed in Section 4.4.7.

- *RegNum*

  denotes the number of *registers* (i.e. slots in the activation frame) required by every activation of the function to store intermediate values. The slots are called registers since they roughly correspond to the registers of conventional von Neumann-processors. However, since function activations work in parallel, each instance needs its own register set, i.e. its own frame.

For a detailed description of the use of these descriptors see Section 5.4.

The first *BlockNum* words of the program store (where *BlockNum* is the number of code blocks in the store) do not contain machine instructions but the base addresses of all code blocks. These adresses are used to keep function references independent of the actual code addresses within the program store[2]

4.4.2 The Frame Store

The frame store is a heap-organized store that contains the *activation frames* of the processor. Each frame is a contiguous memory area that represents the local workspace for one particular function activation. Each activation has its own frame that it has exclusively access to. A frame is constructed of a sequence of slots of length *SlotLen* that are used as follows (see Figure 4.8):

- **Synchronization Bits:**

  Every activation of a function requires *EventNum* frame slots to hold the $EventNum * SlotLen$ bits that represent the synchronization events of the activation. The first slots of a frame are reserved for this purpose. Each of the bits within these slots may be tested and inverted by the synchronization instruction `WAIT` that is used to express the data dependencies within a program (see Section 4.4.7 for more details).

- **Result Address:**

  The next two slots contain the processor number *ProcNum*, the context *Context* and the offset *Offset* of the instruction to that the result of the activation is to be delivered. When the function activation has computed a result, this result value is transmitted by a message token to the instruction denoted by these fields (see Section 5.4 for more details).

- **Registers:**

  The following *RegNum* slots of the frame represent the registers that the function activation may freely use in order to store intermediate values. All instructions of the ADAM instruction set operate on these registers each of which is able to contain a fixed-size value, i.e. basically an integer or a reference to a tuple on the heap (see Section 4.4.7 for more details).

[2]For the moment, the ADAM does not take any advantage from the relocability of code blocks. However, this possibility is reserved for future architecture improvements.

Figure 4.8: An Activation Frame

Figure 4.9: The Context Store

The allocation of an activation frame is always the first action when invoking a function and its deallocation is always the last one before a function is terminated. The compiler is responsible for generating code that is synchronized in such a way that the frame is disposed if and only if no parallel task is active (i.e. in the trigger store) and no message is on the way to that frame any more (see Section 5).

4.4.3 The Context Store

The context store is a heap-organized store that contains one item (the *context*) for each activation of a function. Each context is a unique description of the function activation and consists of the fields

FrameBase	*CodeBase*

that are interpreted as follows (see Figure 4.9):

- *FrameBase:*

  This field contains the base adress of the activation frame, i.e. a pointer to the begin of the frame in the activation store. Actually it does not reference the first slot of the frame but the first slot that is used as a register (the slots before contain the synchronization bits and the result adress of the activation, see Section 4.4.2).

- *CodeBase:*

  This field contains the base adress of the activation's code block, i.e. a pointer to the begin of the block in the program store. Again, the adress does actually not reference the first word of the block but its first instruction (the first word of each block contains a description of the resources that are required by any activation of the function, see Section 4.4.1).

The context store is used to keep references to function activations as small as possible. This indirection table has been introduced since there are many more activation references than activations. Every activation reference contains a *context* field that indizes the actual activation description in the context store. Thus, the reference becomes much smaller than without this indirection step.

4.4.4 The Trigger Store

The trigger store contains a set of *instruction triggers*, i.e. descriptions of parallel tasks that are ready to execute but have not yet been triggered. The execution unit uses this store in order to remember all of those tasks it has temporarily put aside, although they might principially be executed. Each trigger consists of the fields

Context	*Offset*

that are interpreted as follows:

- *Context:*

  This field holds the index of an item in the context store. The referenced context contains the complete description of the function activation that the task belongs to (i.e. the base adresses of activation frame and code block).

- *Offset:*

  This field contains the offset of the task's first instruction, i.e. its position relative to the begin of of the code block denoted by *Context*.

Both fields uniquely determine a parallel task on the processor (see Figure 4.10). At any time, the execution unit is free use the information contained in a trigger in order to start the corresponding task and remove the trigger from the store.

4.4.5 The Message Store

The message store represents a buffer between arbiter and execution unit containing the message tokens that have been sent to the processor. Each message contains the fields

Type	*Data*

where *Type* indicates the interpretation of the value transmitted in *Data*. The following types of messages are possible:

- *Value-Messages* transmitting values between function activations (basically function arguments and result values),

Figure 4.10: The Trigger Store

- *Context-Messages* transmitting result contexts to function activations, i.e. addresses of the instructions that function results are to be sent to,
- *Load-Reports* transmitting workload descriptions between processor units.

Each time the execution unit accepts a token from this store, it is also given its actual load, i.e. the number of messages the store holds. If the store is empty, the load number is zero and the delivered message value is undefined. Otherwise the execution unit handles the message according to its type. How this is done in detail, is described in Section A.3.

4.4.6 The Request Store

The request store contains the call-request tokens that have been sent to the processor. Each call-request contains the fields

Type	*Context*	*Offset*	*ModAddr*	*FunDescr*

that are interpreted as follows:

- *Type:*

  gives the description how the request has to be handled. There exist two possibilites:

  Fast Call:
  In this case, the address of the instruction to that the context number of the new activation is to be sent also indicates the address of the instruction that waits for the result of the function call. No separate context message (see Section 4.4.5) is required and the function activation may *immediately start working*.

  Tail Call:
  The address of the instruction waiting for the function result will be sent in a *separate context message*. This function call mechanism enables the compiler to optimize tail-recursive functions: the last instance of a chain of tail-recursive function activations may return the result directly to the caller of the top-level instance without passing the intermediate chain. Moreover, each activation may terminate as soon as it has passed all arguments to its successor (see Section 5.5 for more details).

- *Context*, *Offset* and *ModAddr*

  uniquely determine the operation to which the context of the new function activation is to be sent: *ModAddr* denotes the address of a processor, *Context* describes some activation on that processor and *Offset* indicates some operation within this activation (see Section 4.4.4). The context index of the new activation is packed into a message token and sent to the denoted instruction.

- *FunDescr:*

  indicates the function to be activated. In detail, this field holds a pointer to one of the first *BlockNum* words in the code store containing the base addresses of all function code blocks.

For further description of the function call mechanisms, see Section 5.4.

4.4.7 The Execution Unit

The execution unit of the ADAM processor has an instruction set that is similar to that of conventional machines but is extended by several features that allow the simulation of dataflow-like behaviour. The ADAM instruction set shows the following characteristic features:

1. The instruction set basically consists of two-address instructions augmented by the address of the successor instruction. Thus, it is of the form

OpCode	*OpX*	*OpY*	*Succ*

   OpCode denotes the operation code of the instruction, *OpX* and *OpY* hold the numbers of the registers (i.e. of slots in the current activation frame) on which the operation is to be performed. Actually, these numbers are offsets to the begin of the register section of the frame, i.e. an operation field of 0 addresses the first register. *Succ* denotes the address of the instruction to be executed next in the form of an offset to the current instruction address. E.g. the instruction

   i: `ADD 1, 2, +4`

   at program address i causes the execution unit to add the contents of register 2 in the actual frame to the contents of register 1 and store the result in register 1. The next instruction to be executed has address $i + 4$ in the program store.

2. Sequences of instructions may be executed without any blocking as long as the execution unit does not encounter a `WAIT` instruction[3]. This instruction denotes one particular point of synchronization between two parallel tasks. It has the form

`WAIT`	*Event*	*Succ*

   Event denotes the number of a synchronization bit within the current activation frame. A `WAIT` instruction is executed by testing the denoted bit and then inverting it. If the tested bit holds a 0, the synchronization event has not yet occurred. In this case, the bit is set to 1 and the current instruction sequence is terminated. If tested bit already holds a 1, the synchronization event has been triggered by a parallel task with another execution of `WAIT`. The event bit is reset to 0[4] and the current instruction sequence is continued with the instruction denoted by the *Succ* field. E.g. if there are two parallel program paths with a wait instruction in each path such as

   i: `WAIT 4,` $+k - i$
   ...
   j: `WAIT 4,` $+k - j$
   ...
   k: `ADD 2, 3, +1`

   (with the instructions at addresses i and j being executed by parallel tasks), that path that first encounters its `WAIT` is interrupted. By this, synchronization bit 4 originally initialized with 0 is set to 1. The other task may now, without interruption, pass its `WAIT` and continue execution with the `ADD` instruction at address k. In this case, synchronization may have been necessary because the arguments for the `ADD` instruction have been computed by two parallel tasks and stored into registers 2 and 3, respectively.

[3] Actually, also the instructions `ALLOC` and `READ` referencing the heap store (see Section 5.7 and the function call instructions `FASTCALL` and `TAILCALL` (see Section 5.4) cause an instruction sequence to be interrupted.

[4] A compile time analysis of the source program is able to optimize the allocation of synchronization bits by determining in which cases already used bits may safely be reused.

3. Each of the instructions in the ADAM instruction set has (at most) one successor instruction denoted by the *Succ* operand field. The only exception of this rule is the `SPLIT` instruction that has the form

SPLIT	*Succ1*	*Succ2*

`SPLIT` forks the actual program path into two parallel tasks starting at instructions *Succ1* and *Succ2*. The description of the task denoted by *Succ1* is put into the trigger store. The processor is now free to execute this task at any time. However, the actual program execution is continued with the task denoted by *Succ2*. E.g., the instruction

i: `SPLIT +1, +2`

creates a trigger with the description of the parallel task starting at instruction $i+1$. Actual execution is continued at address $i+2$.

`SPLIT` and `WAIT` are the only ADAM instructions related to parallelism. However, they suffice for a perfect simulation of all features of dataflow behaviour.

The interaction between execution unit and the different processor stores is illustrated in Figure 4.11:

Each time the execution of an instruction sequence is started, the description of the actual activation (i.e. the base addresses of code block and register frame) is stored in two registers. Any register number encoded in an instruction is handled as an offset to the frame pointer, hence, every function activation has its private set of registers. All event numbers referenced by `WAIT` instructions are handled as *negative* offsets to the frame base address, hence, every function activation has its private sets of events each of which is represented by a bit in the frame.

If a `SPLIT` instruction is executed, the start address of one of the parallel branches is pushed on the trigger store while the other one is continued by the current instruction sequence.

If a `WAIT` instruction fails, the actual instruction sequence is terminated. In this case, the context is switched and a new activation is executed. This may be triggered by one of three events:

1. A **trigger** is popped from the trigger store and the task defined by this trigger is executed. In this case, both of the registers holding the base addresses of actual code block and activation frame are updated by the values that the trigger denotes and execution is started with the first instruction of the parallel task[5] .

2. A **message** is received from the message store. In this case, a new instruction sequence is started beginning with the instruction that represents the destination of the message. Such a message may either hold the argument for a function activation or it may hold the result value delivered by some function activation or it may also hold the result of a memory request (see the next section).

3. An **activation request** is received from the request store. In this case, some neighbour processor has sent a request to activate a new instance of some particular function. If the processor decides to fulfill this request, a new activation frame and a new context descriptor are allocated and a reference to the context descriptor is sent back to the caller.

[5] An important alternative is the concept to make context switches if and only if the trigger store is already empty. In this case, triggers only have to hold the address of the first instruction of the parallel task while the activation frame of the new task is the same as the actual one. Each trigger then represents a *thread* i.e. a lightweight process consisting only of a program counter but inheriting its environment from the father process.

Figure 4.11: Interaction between Processor Stores

The decision which of the two possibilities to take depends on the actual load of the processor. If it is heavily loaded, argument messages will be preferred, if the load is low, more activation requests will be fulfilled. The decision between these two alternatives is (besides the decision to which processor to send activation requests to) the basic tool for load balancing in the ADAM architecture.

In Appendix A.3, we define the operational behaviour of the ADAM processor by describing the algorithm for an interpreter that executes the instruction set of the machine. The semantics of all ADAM instructions is defined in Appendix B. In Chapter 5, we give a short description how several constructs of the high-level language EVE may be compiled into ADAM machine code.

4.5 The Memory Unit

Since the processor units are only able to handle atomic (integer) data, the management of compound data structres has been shifted to separate memory units. Each worker module contains one memory unit that is constructed of the following components (see Figure 4.12):

- **The Heap**
  holds a set of tuples i.e. data structures of arbitrary but fixed size. Each field of a tuple is either empty or holds an integer atom or a tuple reference. All heaps on all worker modules together represent a global store, i.e. all tuple references are unique within the network.
- **The Memory Manager**
  handles incoming requests to allocate a new tuple (`ALLOC`), to write a value into some tuple field (`WRITE`) or to return the contents of some tuple field (`READ`). Furthermore, it is responsible for garbage collection i.e. for reclaiming the space that is occupied by tuples that are referenced no more.
- **The "Expensive" Store and the "Cheap" Store**
  hold all incoming requests classified into two categories: the request may require the allocation of additional heap memory ("expensive") or it may not ("cheap"). The distinction between both kinds of requests helps to efficiently manage the heap store.

Although the ADAM heap is logically global, it is physically distributed among all worker modules. This has the following advantages:

- Requests to local memory units may be handled faster than requests to distant memory units and do not burden the network.
- Memory requests may be handled simultaneously by different modules.
- Network traffic is distributed over the network.

The later propositions rely on the assumption, that the workload is, at least roughly, balanced among memory units and that memory requests are spread over all modules. However, there is some evidence (see [Goldberg, 1988]) that for fine-grained parallelism this assumption can be expected to hold.

On the other hand, the distributed memory architecture requires a distributed model of garbage collection. This is accomplished by using a *reference counting* scheme with *weighted references* (see Section 4.5.4).

There are some essential details that distinguish the ADAM heap store from that of sequential architectures.

Figure 4.12: The Memory Unit

- The allocation of tuples is separated from writing their contents. Special synchronization mechanmisms enable to use tuple references before tuple fields are yet defined (*non-strict data structures*).
- Each tuple carries the information which of its fields have already been written and which not. If a read request demands some value that is not yet defined, the request is queued and answered as soon as the required value has arrived.
- The contents of a tuple field may be written at most once, i.e. the heap store obeys the *single assignment* rule. Any attempt to redefine a tuple field triggers a runtime error.

All these features together are essential for avoiding long memory latencies but preserving the determinism of parallel programs. They have been introduced by the concept of *I-structures* as described in [Arvind *et al.*, 1987].

The following sections provide an overview on the components of the memory unit. A more formal description of their operational behaviour can be found in Appendix A.4.

4.5.1 The Heap

Logically, the heap store is a set of tuples that consist of an arbitrary but fixed number of fields. Each field may contain an integer value or the reference to another tuple. This memory model is able to represent all of the data types in high-level (functional) programming languages, such as tuples, lists, algebraic types, etc. All storage instructions of the ADAM instruction set (see Appendix 5.7) make use of this heap model.

However, in order to keep storage mangagement simple, the physical construction of the heap is somewhat different. Actually, the heap only consists of fixed-size storage cells each of which is constructed two fields, the *head* and the *tail*, and is attached with an additional *tag*.

Internally, a logical tuple consisting of n fields, is mapped to $n-1$ physical heap cells in the following way (see Figure 4.13):

- The first $n-2$ tuple fields are mapped to $n-2$ heap cells. The head component of the i-th cell holds the value of the i-th tuple field while the tail component contains a pointer to the $(i+1)$-th cell. These cells are marked as *open* in their tag fields.
- Both of the last tuple fields are mapped to a single heap cell. The head of this cell holds the $(n-1)$-th field while the tail holds the n-th field. This node is marked as *closed* in its tag field.

As an important special case, *linked lists* are totally mapped to closed heap cell: each list cell contains exactly two fields that denote the first element of the list and a pointer to its rest, respectively.

In detail, a *heap cell* consists of the fields

Tag	*Head*	*Tail*

that are interpreted as follows:

- The *head* and the *tail* are that cell components that user-defined data may be stored into, i.e. each of them is able to hold an integer value or a tuple reference.

Figure 4.13: Mapping Tuples to Heap Cells

- The *tag* contains system-internal information required for a correct heap management. It is built of the following components:

Open	*RefCnt*	*PtrHd*	*AvHd*	*PtrTl*	*AvTl*

  – The *Open* field holds a bit that indicates the interpretation of the cell:
    * If *Open* is set, the tail field holds a pointer to the heap cell that represents next tuple field.
    * If *Open* is not set, the tail field holds the next (i.e. the last) field of the tuple.

  – The *RefCnt* field contains a measure for the number of pointers that reference the cell. Actually, this measure does not denote the number of the pointers itself but the sum of their *weights*. However, if that number decreases to zero, the cell can not be accessed any more and may be reclaimed. For further details on this topic, see Section 4.5.4.

  – The fields *PtrHd* and *PtrTl* indicate whether the contents of the cell's head respectively tail have to be interpreted as integer values or as tuple references. This information is neccessary for the correct the garbage collection mechanism used (also see Section 4.5.4).

  – The fields *AvHd* and *AvHd* indicate whether the contents of the cell's head respectively tail have already been written or not. Depending on this fact, incoming read requests may be fulfilled or have to be deferred (see Section 4.5.2).

4.5.2 The Request Stores

The memory unit receives two kind of requests that are distributed among different kind of request stores:

1. **Expensive Requests:**

   A request is called *expensive*, if handling this request may require the allocation of additional heap cells. Obviously, *allocation requests* but also some other kinds of requests belong to this category.

2. **Cheap Requests**

   All other requests are called *cheap*, since they differ from the first kind of requests in some essential sense: the memory manager may always handle these requests without consuming additional memory resources.

This distinction helps to manage the resources of the memory unit as efficient as possible (see Section 4.5.3). The various kinds of requests are now described in detail.

Allocation Requests

Each allocation request asks the memory manager to reserve space for a tuple of some arbitrary but fixed length. If the manager accepts this request, it allocates the required space and returns the address of the tuple to the sender of the request. If the manager is not willing to accept the request (e.g. due to lack of unused space) it forwards the request to a memory unit that is sited on one of the neighbour modules (or puts it back into its request store again).

An allocation request token is constructed as follows:

Num	*Cont*	*Off*	*Mod*

The fields of the token are interpreted in the following way:

- *Num* denotes the required number of tuple fields.
- *Cont*, *Off* and *Mod* are the context, the offset and the module number of the instruction to that the new tuple reference is to be sent.

If the memory manager receives an allocation request, there are two cases:

- There is enough memory left to fulfill the request, i.e. there are enough heap cells available to represent the desired tuple. In this case, the required cells are allocated and properly initialized (i.e. linked together) to represent the tuple. The address of the first cell is returned as the address of the tuple.
- There are less free heap cells available than required to fulfill the request. In this case, the manager forwards the request as a whole to that memory unit that it thinks to be least loaded among its neighbours.

As an alternative to this scheme, the memory manager might also decide to fulfill the request partially, i.e. to allocate enough memory to represent some part of a tuple and to send a request for the rest of the memory to another module. However, allocating the components of a tuple on different processors has some essential drawback: since the reference to a tuple is defined as the reference to the cell that holds its first component, read requests may have to be forwarded to other modules, which leads to an increased network traffic. In order to minimize communication overhead, this possibility is therefore not investigated furthermore.

Read Requests

A read request token consists of the fields

Num	*Cont*	*Off*	*Mod*	*Tuple*

where the various fields have the following interpretation:

- *Tuple* and *Num* denote the address of a tuple (i.e. the address of the first heap cell representing the tuple) and the number of a field within this tuple.
- *Cont*, *Off* and *Mod* denote the context, offset and module address of the instruction to that the contents of the tuple field are to be sent.

There are three cases that determine how a read request has to be handled:

1. The desired tuple field does not exist, i.e. the referenced tuple consists of fewer fields than required. This case indicates a severe compilation error, the generated code is in some way invalid. Hence, the memory manager terminates execution reporting the kind of error occurred.
2. The tuple component is still empty, i.e. its contents have not yet arrived in the memory unit. In this case, the tuple field actually holds the reference to a list of all those instructions that have up to now requested the contents of this field. A new heap cell holding the return address of the new read request is allocated and linked to this list.
3. The tuple component holds a valid value (integer or tuple reference). In this case, the value is packed into a message token and sent to the denoted instruction.

The second case is the reason for read requests being **expensive requests**. It may be neccessary to allocate a heap cell in order to store the read request in the memory unit.

Submission Requests

A submission request is the demand to interpret some storage tuple as the description of a *curried function* and to to submit the lifted arguments stored in the tuple to some specific activation of that function. Basically, such a request is constructed of the fields

Context	*Module*	*Tuple*

with the fields being interpreted as follows:

- *Context* and *Module* denote the address of a function activation, i.e. the address of its context and its module, respectively.
- *Tuple* denotes the address of the tuple representing the description of the curried function.

The description of the curried function stored in the tuple is constructed as follows:

$\langle f, n\rangle$	arg_1	arg_2	$\ldots$	arg_i

- The first tuple field consists of a pair $\langle f, n\rangle$ where f is the address of the function and n is the parameter number of the first of its i lifted arguments.
- The remaining i field hold the lambda-lifted arguments of the function, i.e. all those arguments that represent the global variables referenced in the original function body.

The memory manager handles this request by packing the lifted arguments into message tokens and sending the tokens to the denoted activation. In detail, the j^{th} argument is sent to the $(n+j)^{th}$ instruction of the function activation. If some of the arguments have not yet defined, read requests with the appropriate result addresses are constructed and queued into the concerned fields (see Subection 4.5.2). As soon as the arguments arrive, the corresponding requests are fulfilled.

Since (parts of) the request may have to be stored in the memory unit, submission requests are **expensive requests**. For more details on the handling of curried functions, see Section 5.6.

Write Requests

A write request token is constructed of the components

Type	*Num*	*Value*	*Tuple*

with the various fields being interpreted as follows:

- *Type* denotes the kind of value to be written, i.e. whether it is an integer value or a tuple pointer.
- *Tuple* and *Num* denote the address of some tuple and the number of an empty field within this tuple, respectively.
- *Value* denotes the (integer or pointer) value that is to be written into this component.

How the write request is handled, depends on several conditions:

- If the referenced tuple field does not exist, this indicates a severe program error. In this case, the memory manager terminates execution and reports the kind of error occurred.
- If the tuple field already holds a valid datum i.e. if another write request has already been sent to this field, the *single assignment rule* (see Section A.4) is violated. Since this also indicates a severe program error, execution is terminated, too.
- Otherwise the tuple field has not yet been written, i.e. it contains a reference to a list of instruction addresses to that the datum is to be sent. Before the memory manager writes the datum into the tuple field, it packs it into a message token and sends copies of this token to all the denoted instructions. At last, the datum is written into the tuple field and the available bit is set.

Reference Messages

A reference message informs the memory manger that a reference to some tuple has been deleted or has to be duplicated and that therefore the reference count field of that cell has to be updated. There are three kind of reference messages

1. Increment messages,
2. Decrement messages,
3. Copy messages.

Increment messages and decrement messages are basically constructed from the fields

Type	*Context*	*Offset*	*Module*	*Tuple*

that are interpreted as follows:

- *Type* denotes the kind of reference request the token represents:
  - IncRef tells the manager that the pointer *Tuple* is to be duplicated.
  - DecRef tells the manager that the pointer *Tuple* is to be deleted.
- *Tuple* denotes a reference to some tuple whose reference count has to be updated as a consequence of the message.
- *Module*, *Context* and *Offset* denote the module address, the context and the offset of the instruction that has sent this request. This kind of information is only required in the case of increment requests, otherwise it is ignored.

Decrement messages have to be sent only if tuple references are explicitly deleted (which is very seldom the case). Moreover, since the garbage collection scheme uses weighted references, *increment messages* are only rarely necessary. Actually, this kind of message has to be only sent if a reference with weight is to be duplicated.

Copy messages are constructed from the fields

CopRef	*Field*	*Reference*	*Module*	*Tuple*

that are interpreted as follows:

- *Tuple* denotes a reference to some tuple whose reference count has to be incremented.
- *Field*, *Reference* and *Module* denote some field of a particular tuple on some memory unit to that the result of the request is to be sent.

Copy messages have to be submitted if some read request demands the value of a tuple field that itself holds a tuple reference with minimal weight. In this case, the field must not be copied but a copy message has to be sent to the memory unit that holds the referenced tuple demanding a new reference with greater weight.

For a detailled description of the garbage collection mechanism, see Section 4.5.4.

Load Messages

A *load message* informs the memory manager that the load of some neighbour memory unit has significantly changed. It basically consists of the fields

Mod	*MemLoad*	*RequLoad*

with following interpretation:

- *Mod* is the module address of the memory unit that sent this token.
- *MemLoad* is the number of heap cells allocated in that unit.
- *RequLoad* is the number of expensive requests that wait for service in that unit.

The memory manager uses this piece of information to update an internal table that holds the load of all neighboured memory units. Depending on whether the load of this unit has increased or decreased, in the future the manager will forward less respectively more allocation requests to that unit.

4.5.3 The Memory Manager

The memory manager is responsible for the managemenot of all incoming memory requests. According to some particular strategy, it first decides which kind of request (i.e. an expensive or a cheap request) to accept and then provides the desired service.

The manager holds a local table of its load and that of all memory units sited on the neighbour modules. As soon as the load of a memory unit has significantly changed, the corresponding manager informs its "colleagues" about the new situation. This piece of information basically consists of the number of heap cells that are currently in use and the number of expensive requests waiting on that unit.

If the manager is not willing to fulfill some particular allocation request (e.g. due to lack of unused memory or if some neighbour is significantly less loaded) it uses the information stored in its load table to forward that request to one of its neighbours.

The formal definition of the memory manager's operational semantics is given in Section A.4.

4.5.4 Garbage Collection

The ADAM garbage collection mechanism uses a simple distributed reference counting algorithm that basically relies on the idea of [Bevan, 1987]. For the sake of simplicity, improvements of this algorithm as suggested in [Rudalics, 1988a] and [Rudalics, 1988b] have not yet been considered.

The major features of the algorithm are as follows:

- It is *relatively simple*.
- It is *inherently parallel* and especially suitable for *distributed memory* architectures.
- It is *true real-time* and does not require any separate garbage collection phase.
- It requires *few additional messages* and *no synchronization* between messages.

- It is relatively *space efficient* using only a few bits per reference.

The basic idea of this algorithm is to associate to each reference a *weight* and to each tuple a *reference count*. The algorithm always maintains the invariant

> **The reference count of a tuple is equal to the sum of the weights of all references to it.**

This ensures that the reference count of an object is zero if and only if it is not referenced any more. In this case, the heap cells that represent the tuple may be reclaimed for further use.

In detail, the algorithm works as follows (illustrated in Figure 4.14):

- **Allocating Tuples:**

  If a new tuple is allocated, the tuple is given some arbitrary reference count which has to be a positive integer power of two. The binary logarithm of this value is stored as a weight in the new reference that is returned as result of the allocation request.

- **Deleting References:**

  If a storage reference is explicitly deleted, a *decrement message* is sent to the memory unit that holds the referenced tuple. This message contains the reference to be deleted consisting of the address of a tuple and a weight. The reference count field of the denoted tuple is decreased by the weight (i.e. by the stored value taken to the power of two). If the reference count field decreases to zero, the tuple is not referenced any more and may be reclaimed.

- **Duplicating References:**

  If a storage reference is duplicated, both of the new references get half of the weight of the original reference. Since reference weights are always given as binary logarithms, this means to decrease the weight field by one.

  If the reference weight is one, the reference must not be immediately copied but an *increment request* has to be sent to the memory unit containing the referenced tuple. The reference count field of the tuple is increased by the maximum weight minus one (considering the weight of the old reference) and a new reference to the cell is created with its weight field set to its maximum. This reference is sent to the instruction that wishes to duplicate the reference, which may now be done as previously described.

  If the reference count field of the tuple already holds the maximum value that can be stored in that field, this value is considered as "infinity". All further *increment* and *decrement requests* do not have any more effect one the reference count, i.e. the tuple will never be reclaimed.

Basically, tuple references have to be duplicated (i.e. their weight fields have to be decreased) in the following cases:

1. The tuple reference is forwarded to a parallel task but the actual process will also use the reference. In this case, the reference has to be duplicated *before* the reference is made accessible to the other task. This is necessary since both tasks may now make use of the reference as described in the next paragraph.

2. A `READ`, `SUBMIT` or `WRITE` request is to be sent to a tuple but the tuple reference has still to be used for further activities. In this case, the reference has to be duplicated *before* the request token is submitted. This is necessary since the reference count of the tuple has to be decreased by the weight of the reference stored in the request token. Otherwise, a tuple might be reclaimed although requests are still on the way to the tuple.

Figure 4.14: Garbage Collection

A tuple reference has to be explicitly deleted (and a decrease message to be sent to the tuple) only if a process receives the reference and does neither forward it to a parallel process nor use it for any memory request. However, this is very rarely the case.

The basic difference between this approach and the original idea is the fact, that it does not make use of *indirection cells* as suggested in [Bevan, 1987]. Instead of this, we allow the reference count of a tuple to be increased up to some upper bound. The advantage of this method is that the allocation of additional heap cells and chains of indirection cells (slowing down tuple references) can be avoided.

The only drawback is the fact that, once the reference count of a tuple has reached its maximum, the cell will never be reclaimed any more. However, there is much statistical evidence ([Rudalics, 1988b]), that also for very small reference count fields (3 bit) more than 99% of all cells will be reclaimed.

Another disadvantage of our scheme (as of all reference counting algorithms) is the fact that *circular structures* are not reclaimed, since their reference counts will never decrease to zero any more. However, we are of the opinion that the simplicity of this algorithm together with its suitability for the distributed memory model justifies to ignore this particular (and not very serious) insufficiency.

Chapter 5

Compiling the EVE Language into ADAM Code

In this chapter, we give a general introduction into the concepts used for translating the functional language EVE into machine code for the ADAM architecture. The goal of this overview is not to provide an exact definition of the compilation process, but to give some intuition into its basic principles as opposed to the compilation of imperative languages for conventional architectures. A detailled description of a compiler for the EVE programming language will be published in the frame of another diploma thesis [Loidl, 1992].

5.1 Basic Compilation Steps

A compiler translating the EVE language into ADAM machine code may handle the compilation process in four passes as illustrated in Figure 5.1. Several parts of this process are strongly related to the compilation of sequential programming languages as described in [Aho *et al.*, 1986].

1. **Lexical Analysis and Syntax Analysis**

   The first compiler pass handles the *lexical analysis* of an EVE source program i.e. the construction of syntax tokens (terminal symbols) from the characters of the source file. A top-down parser receives the tokens delivered from the lexical analyzer in order to perform a *syntax analysis* of the program and to construct the corresponding syntax tree. The context-free grammar of EVE can be easily transformed into an equivalent LL(1) grammar, thus the parser may be implemented using the method of recursive descent.

   Besides the syntax of the program, the parser has (supported by an attributed EVE grammar) to check its correct semantics, in particular it is responsible for *type checking* the program. Since in EVE the definition of an object must always precede its use, the correct typing of all definitions and expressions can be easily checked in a single pass.

   This first compilation pass is already able to catch all syntactic and semantic programming errors and to reject invalid programs. The result of this step is the program translated into a form of **enriched lambda calculus** as described in [Peyton Jones, 1987]. Basically, the extensions added to the pure lambda calculus consist of the "let-in" construct for the (local) definition of named constants and functions. Moreover, for a correct delivery of the result values by the ADAM master module (see Section 3.2), a **type descriptor file** has to be generated indicating the data type of the result.

Figure 5.1: The Eve Compilation Process

2. **Lambda-Lifting**

   In Eve like in most other functional programming languages, the definition of local functions are allowed that may refer to global constants defined in the enclosing environment. In particular, a local function may refer to the parameters of the outer function even if these parameters have not been passed as explicit arguments (lexical scoping).

   However, in the Adam machine architecture there is no global environment. The only values that a function has access to are those that have been submitted as arguments. Before Adam code may be generated from an Eve program, the program has to be "flattened" in the sense that all local functions are lifted to the top level and all global variables used within them are passed as additional arguments.

   This transformation is performed by the *lambda lifting* algorithm ([Johnson, 1985]) transforming lexically scoped functional programs into sets of **super-combinators** (i.e. functions without free variables). This super-combinator form of a program is a special case of the enriched lambda-calculus and may now serve as the basis for code generation.

3. **Code Generation and Optimization**

   In this step, the super-combinator form of the program is translated into Adam assembly code. Basically, from the body of each super-combinator a dataflow program may be generated (as described in [Traub, 1986]) and this dataflow program may then be translated one-to-one into Adam assembly code.

   However, in order to utilize the full power of the Adam facilities, a number of optimizations should be done:

   (a) A *strictness analysis* of each super-combinator determines which parts of the function body depend on which function arguments. Those parts that successfully compute a result if and only if (several or) all arguments have become available may safely wait for the arrival of these arguments and can then be sequentially executed. However, special care has to be taken to preserve the non-strictness semantics of the Eve language: the body of a function that may return results independently of some argument must not be synchronized in advance with that particular argument.

   (b) *Register allocation and assignment* may be optimized by avoiding those value duplications whose only purpose is to give several tasks simultaneous access to the duplicated values. If none of the tasks changes the contents of the register storing the value, the duplication is superfluous and can be removed. Moreover, registers that are no more used by one task may in certain cases be reused by other tasks. However, a correct synchronization is of essential importance for this kind of optimizations.

   (c) Several other optimizations known from conventional architectures (like *peephole optimization*) may applied to the Adam architecture, too.

   The result of this pass is a program in Adam **assembly code**. The code generator does not directly generate object code in order to make the result of the compilation readable for human beings such that inefficiencies can be detected and removed by more optimizations.

4. **Assembly**

   Finally, the last pass of the compiler translates the **assembly code** generated by the previous compilation phase into **object code** that may be executed on the Adam architecture.

The following sections describe in more detail which code may be generated by the compilation of several Eve language constructs.

Figure 5.2: A Dataflow Graph

5.2 Arithmetical Expressions

Any arithmetical expression can be translated in a directed acyclic graph. The nodes of this graph represent the operations performed and its edges represent the data dependencees between the operations. An edge e from operation op_1 to op_2 indicates that the result value of op_1 is an input argument to op_2. For example, the expression

$$x_0 * y_0 + x_1 / y_1$$

can be translated in the *dataflow graph* shown in Figure 5.2.

Obviously, the nodes of a dataflow graph have to be translated into ADAM instructions. But how shall the data dependences be translated? In the ADAM architecture, the only communication medium between two instructions is the register set of the actual activation frame. The dependency edge from $*$ to $+$ can therefore be expressed by a register r_0 which takes the result value of $*$ and is input to $+$. Obviously, $*$ is a successor instruction of $+$, therefore the control flow has to be routed from instruction $*$ to instruction $+$.

In the same way, the dependency edge from $/$ to $+$ can be translated by a register r_1 that receives the result of the $/$ operation. The control flow has also to be routed from $/$ to $+$.

Since $+$ has more than one input arguments, i.e. the result values of the operations $*$ and $/$, synchronization between $*$ and $/$ is necessary. This can be expressed by inserting a `WAIT` instruction testing a synchronization event e_0. But also for the synchronization of the input arguments x_0, y_0 for the $*$ operation, respectively x_1, y_1 for the $/$ operation, two `WAIT` instructions with synchronization events e_1 and e_2 are required.

Compiling the expression $x_1 * y_1 + x_2 / y_2$ into ADAM code may therefore yield the following result (the instructions 0:`WAIT` and 1:`WAIT` are the possible entry points to this piece of code, i.e. both of them have no predecessors in the code block):

0:	`WAIT`	e_1, +2	– Synchronization between x_0, y_0
1:	`WAIT`	e_2, +2	– Synchronization between x_1, y_1
2:	`MUL`	x_0, y_0, +2	– $x_0 := x_0 * y_0$
3:	`DIV`	x_1, y_1, +1	– $x_1 := x_1/y_1$
4:	`WAIT`	e_0, +1	– Synchronization between 2: and 3:
5:	`ADD`	x_0, x_1, +1	– $x_0 := x_0 + x_1$

Since the Adam machine code consists of two-address instructions, the first of the input registers also gets the result value of the operation.

However, this is not the only possibility of synchronization. The expression might also be compiled into following code block (with again instructions 0: and 1: being the entry points):

0:	`WAIT`	e_1, +1	– Synchronization between x_0, y_0
1:	`WAIT`	e_2, +2	– Synchronization between x_1, y_1
2:	`WAIT`	e_3, +1	– Synchronization between 0: and 1:
3:	`MUL`	x_0, y_0, +1	– $x_0 := x_0 * y_0$
4:	`DIV`	x_1, y_1, +1	– $x_1 := x_1/y_1$
5:	`ADD`	x_0, x_1, +1	– $x_0 := x_0 + x_1$

This kind of synchronization is also perfectly valid and generates a piece of code that is sequentially executed. This small example already shows one of the big advantages of the architecture: By seperating synchronization instructions from computational instructions we are given any freedom in choosing the optimal kind of synchronization (with respect to the tradeoff between parallelism and memory workload).

However, the use of registers and seperate synchronization instructions may also help to optimize code generation in the case that the same variable occurs in an expression more than once. E.g. a clever compiler may generate from the expression

$$(x + y) * (x - y)$$

the following piece of code:

0:	`WAIT`	e_0, +1	– Synchronization between x, y
1:	`COPY`	z, x, +1	– $z := x$
2:	`ADD`	x, y, +1	– $x := x + y$
3:	`SUB`	z, y, +1	– $z := z - y$
4:	`MUL`	x, z, +1	– $x := x * z$

saving two superfluous `WAIT` instructions! In pure dataflow architectures, the expression above would result in two `COPY` operations for each of the values x and y. However, thies only yields two parallel program paths executed on the same processor and then again makes two synchronization operations necessary.

5.3 Conditional Expressions

The correct treatment of conditional expressions is a little bit difficult in the sense that each branch has to be correctly synchronized with all parallel paths of the surrounding expression. Take, for example, the expression

$$\textbf{if } a = 0 \textbf{ then } 1 \textbf{ else } a + b$$

with free variables a and b and let this expression be the body of a function $f(a,b)$. A translation of this expression into ADAM code may look as follows (in order to illustrate the control flow, statements have been attached with symbolic labels instead of numerical addresses):

```
          ...         ..., r0, a              – a has arrived in r0
          ...         ..., r1, b              – b has arrived in r1

a:        COPY        r2, r0, +1              – r2 := a
          EQUALC      0, r2, +1               – r2 := (a = 0)
          SPLIT       b, +1                   – Signal r2 available
          SWITCH      r2, true1, false1       – Conditional jump

error1:   RETURN      r2, +1                  – Return ERROR
          WAIT        e2, term                – Wait for b and terminate

b:        WAIT        e1, +1                  – Wait for r2
          SWITCH      r2, true2, false2       – Conditional jump
error2:   WAIT        e2, term                – Synchronize with error1
true2:    WAIT        e3, term                – Synchronize with true1
false2:   WAIT        e4, wait                – Synchronize with false1

true1:    RETURNC     1, +1                   – Return 1
          WAIT        e3, term                – Wait for b and terminate

false1:   WAIT        e4, wait                – Wait for b
wait:     ADD         r0, r1, +1              – r0 := a + b
          RETURN      r0, term                – Return r0 and terminate

term:     TERMINATE
```

As soon as a has become available, the processor can decide which branch of the conditional to execute. The boolean value indicating the result of this decision is stored in r_2. However, before the branch is actually entered, a parallel path starting with b:`WAIT` is forked that indicates that the comparison result in r_2 is available.

When b has arrived, b:`WAIT` synchronizes with the parallel path that has been forked after the comparison operation. Since the comparison result of available, the actual path can decide with which branch a synchronization is required.

In contrast to the usual conditional statements, `SWITCH` has *three* exits, one exit for each of the possible results 'true', 'false' and 'error', respectively.

- If a has been the error value, r_2 also contains the error value. If this is the case, error$_1$:`RETURN` is executed delivering ERROR as its result. However, before the whole activation may terminate, the actual path has to wait for the arrival of b that is signalled by the event e_3. As soon as this synchronization has been performed, the function activation may terminate.
- If $a = 0$, true$_1$:`RETURNC` is executed returning the constant 1 as its result. Again, before the activation may terminate, the actual path has to wait for the arrival of b signalled by the event e_3.
- If $a \neq 0$, false$_1$:`WAIT` performs a synchronization with the arrival of b. Afterwards, $a + b$ is computed and the sum is returned as the result of the activation. The function activation may then be immediately terminated.

As shown above, each branch of the conditional statement has to be synchronized with *each* of the arguments required by *any* branch of the construct[1]. While a result may be delivered by one branch even if some global arguments are missing, the whole activation must *not* be terminated before all missing arguments have become available.

5.4 Function Calls and Function Definitions

The compilation of a **function call** requires the generation of code for the following tasks:

1. Sending a request for the creation of a new function activaton,
2. Sending the function arguments to the new activation,
3. Receiving the result value of the activation.

For example, a function call $f(a, b)$ with arguments a and b may be compiled into the following piece of code (with entry points at the addresses 0, 1 and 2):

0:	...	$\ldots, +3$	– Trigger 3 :
1:	...	$r_0, \ldots, +6$	– a in r_0
2:	...	$r_1, \ldots, +8$	– b in r_1
3:	FASTCALL	$f, +1$	– Request activation of f
4:	COPY	$r_3, r_3, +2$	– Activation address in r_3
5:	...	$\ldots, r_4, +8$	– Result in r_4
6:	SPLIT	$+1, +4$	– Parallel paths 7 : and 10 :
7:	WAIT	$e_0, +1$	– Synchronize 1: and 6:
8:	SEND	$r_0, r_3, 0$	– Send a to f as argument 0
9:	JUMP	$+3$	– Successor 12 ::
10:	WAIT	$e_1, +2$	– Synchronize 2: and 6:
11:	SEND	$r_1, r_3, 1$	– Send b to f as argument 1
12:	WAIT	$e_2, +1$	– Synchronize 9: and 11:

There are **three parallel paths** by that this piece of code may be simultaneously entered:

- The instruction at address 0 may be executed as soon as it becomes certain that f is to be activated. Its successor instruction is 3:`FASTCALL` that submits a call request token into the network.
- The instruction at address 1 may be executed as soon as the value a of the first argument of f has become available. Its successor instruction is 7:`WAIT` that synchronizes the path with the task calling the function f.
- The instruction at address 2 may be executed as soon as the value b of the second argument of f has become available. Its successor instruction is 10:`WAIT` that synchronizes the path with the task calling the function f.

The submitted call request token carries the request to activate a new instance of f and informs the processor fulfilling this request that the address of the newly created activation has to be sent to 4:`COPY`. In this case, the only purpose of the instruction is to tell the processor to store the activation address received into r_3 and jump to successor 6:`SPLIT`. The result value of the activation

[1] However, it is not necessary to synchronize each path with a different event as shown in the example.

will be sent to the instruction immediately next in the program store, i.e. to the instruction at address 5.

After the activation address has been stored into r_3, 6:`SPLIT` forks program execution into two parallel paths starting at 7:`WAIT` and 10:`WAIT`. Each `WAIT` synchronizes with one of the tasks that have computed arguments a and b, respectively. As soon as any of these arguments has become available, it is sent to the new function activation. After both arguments have been submitted, the corresponding parallel tasks are joined by 12:`WAIT`.

The code block depicted above has two parallel exit paths. The one path starting at address 5 is triggered as soon as the result value of the function activation has become available. The other path is the result of joining both paths that have submitted arguments a and b, respectively. It can be used to signal to the program surrounding the code block that all instructions of the code block have been executed. This is necessary in order to determine when a function activation may be terminated.

Compiling a **function definition** requires the generation of code for

- Receiving the function arguments,
- Executing the body of the function,
- Returning the function's result value,
- Terminating the function activation.

For example, if we want to compile the function definition $f(a, b)$ with formal parameters a and b, we may generate the following piece of code:

0:	...	$\ldots, +i$	– Trigger Instruction
1:	...	$\ldots, r_0, +j$	– Receive a into r_0
2:	...	$\ldots, r_1, +k$	– Receive b into r_1
	...		– Body Computes Result into r_2
$n-3$:	`WAIT`	$e_1, +1$	– Synchronize with Trigger
$n-2$:	`RETURN`	$r_2, +1$	– Send Result in r_2 to Caller
$n-1$:	`WAIT`	$e_2, +1$	– Synchronize $n-2$: with Rest of Body
n:	`TERMINATE`		– Deallocate Activation Frame

The first instruction at address 0 is triggered as soon as the new activation is created, yet before any of the function arguments has arrived. This "trigger instruction" starts the execution of the code block, since it is possible that instructions of the body may be executed before all (or even any) of the arguments have arrived. Moreover, the trigger indicates that the "return address" has been delivered to the current activation, i.e. the address of the instruction to that the result of the function activation is to be sent. No `RETURN` statement must be executed before the result address has arrived (it is possible that function arguments arrive before the result address does, see the next section).

$n-3$:`WAIT` synchronizes the body that has computed the result with the trigger path that indicates that the result address has arrived. $n-2$:`RETURN` delivers the computed result value stored in r_2 to the denoted instruction. Afterwards, program execution has to be synchronized with all those instructions for that it is not guaranteed that they have been executed even if a result has been computed. At last, if it is sure that no instruction in the function body is under execution any more, the activation may be terminated and its corresponding frame be deallocated.

5.5 Tail-Recursive Functions

A function is called *tail-recursive*, if depending on the values of the input arguments, the function computes its result either directly (without any recursive call) or returns the result of a recursive call without any change as its own result value.

Tail-recursive functions may be optimized such that the last instance of the chain of recursive function activations delivers its result directly to the caller of the top-level activation without passing the value through the intermediate chain. Besides the speedup achieved, this method has the essential advantage that each function activation may terminate and deallocate its frame, as soon as it has passed all of the arguments to the next activation.

However, since the function call mechanism is *non-strict* (i.e. functions may be called before their arguments have become available), it is not possible to use only one activation frame for the whole chain of recursive activations. A function instance may activate the next one, even if the arguments of this instance are still under computation. As a consequence, only those function instances are active and occupy resources, for which still useful work is to do. This feature leads to an essentially more efficient memory utilization.

Compiling a tail-recursive function call $f(a, b)$ may yield the following code block:

0:	...	$\ldots, \ldots, +3$	– Trigger 3:
1:	...	$r_0, \ldots, +6$	– Receive Argument 0 as r_0
2:	...	$r_1, \ldots, +8$	– Receive Argument 1 as r_1
3:	`WAIT`	e_5, +1	– Wait for Result Context
4:	`TAILCALL`	f, +1	– Request Activation of f
5:	`CONTEXT`	r_3, +1	– Submit own Result Context
6:	`SPLIT`	+1, +4	– Parallel paths at 7: and 11:
7:	`WAIT`	e_1, +1	– Synchronize 1: and 6:
8:	`SEND`	r_4, r_3, 0	– Send r_4 as a
9:	`JUMP`	+3	– Jump to Synchronization
10:	`WAIT`	e_2, +2	– Synchronize 2: and 6:
11:	`SEND`	r_5, r_3, 1	– Send r_5 as b
12:	`WAIT`	e_3, +1	– Synchronize 9: and 11:

Similiar to standard function calls presented in the previous section, this code block has three entry paths indicating the arrival of arguments a and b and triggering the function call, respectively.

The essential difference is denoted by the instructions 4:`TAILCALL` and 5:`CONTEXT`. In contrast to `FASTCALL`, `TAILCALL` submits a call request without telling the new activation where to return the result value to. `CONTEXT` receives the address of the new activation and submits the result address of the actual activation to the new one. By this, the result value will be directly sent to the caller of the actual activation, i.e. the actual function does not receive any result value! The previous synchronization of 4:`WAIT` with the actual result address guarantees that the submitted result address is well-defined.

The optimization described in this section is not restricted to tail-recursive functions. It may be applied in all cases in that the result of a function call is, without any change, the result of another function call. Therefore, it may in many cases help to generate more efficient programs with respect to time and, in particular, space.

5.6 Curried Functions

The ADAM architecture also supports the use of *curried functions*, i.e. functions that are passed as function arguments or returned as function values. A curried function is a $i + 1$ tuple on the heap consisting of the fields

$\langle f, n \rangle$	arg_1	arg_2	...	arg_i

that have the following interpretation:

- The first field consists of a pair $\langle f, n \rangle$, where f is the address of the function (i.e. of its code block) and n is the parameter number of the first of the i fixed arguments.
- The remaining i fields hold the lifted arguments of the function, i.e. all those arguments that are submitted independent of the actual function call.

The curried function is now represented by a pointer that references this tuple. This pointer may be handled lie any atomic value, in particular it may be passed as argument or returned as result value of a function. If a function call $f(a, b)$ with a curried function f and arguments a and b is compiled, the following piece of code may be generated:

0:	...	..., r_0, +3	– Tuple reference in r_0
1:	...	..., r_1, +7	– Argument a in r_1
2:	...	..., r_2, +9	– Argument b in r_2
3:	`READ`	r_0, 0, +1	– Read First Tuple Field $\langle f, n \rangle$
4:	`APPLY`	r_3, +1	– ⟨f, n⟩ in r_3, Request Activation
5:	`SUBMIT`	r_0, r_4, +2	– Submit Lifted Arguments
6:	...	..., r_5, +6	– Result value in r_5
7:	`SPLIT`	+1, +3	– Successors 8: and 10:
8:	`WAIT`	e_0, +1	– Synchronization 1: and 7:
9:	`SEND`	r_0, r_3, 0	– Send a as Argument 0
10:	`JUMP`		– Jump to Successor Instruction
11:	`WAIT`	e_1, +2	– Synchronization 2: and 7:
12:	`SEND`	r_1, r_3, 1	– Send b as Argument 1

Before the actual activation of f can be done, 3:`READ` has to fetch the code block address of f (stored in the first field of the referenced tuple) from the heap. Then 4:`APPLY` requests the allocation of a new activation frame for f. As soon as the address of the new activation has arrived, 5:`SUBMIT` triggers the submission of the fixed arguments stored in the referenced tuple on the heap. At last, 9:`SEND` and 12:`SEND` submit the actual function arguments.

The activation of curried functions only requires three more network messages than the activation of functions that are already fixed at compile time: two for demanding and receiving the description of the function from the heap (`READ`) and one for triggering the submission of the lambda-lifted arguments stored on the heap (`SUBMIT`). In all other matters, the use of curried functions is not essentially different.

5.7 Operations on Data Structures

Basically, there are two operations on compound data structures:

1. Construction of a Structure,
2. Selection of a Structure Component.

In EVE, there are several compound data types (lists, strings, tuples, unions). However, if is not difficult to realize how these types can be mapped to the tuple concept available on the level of the ADAM architecture.

Basically, the selection of a tuple component by a `READ` statement can be done without any problems. However, one has consider that `READ` is a blocking operation and must be done in a separate parallel process in order to preserve the non-strictness of the EVE language.

While the selection of a tuple field is rather simple, the construction of a tuple is more complicated. In EVE a data structure can only be constructed as a whole, i.e. all components of the structure must be defined. Consider e.g. the expression

$$a :: b$$

constructing a list cell with head a and tail b and let this expression be the body of a function $f(a, b)$. While in the high-level language EVE the allocation of a list cell and the definition of its components are indivisible, they are strongly separated on the level of the ADAM architecture:

	...	..., alloc	– Trigger allocation
	...	..., x0, wrHd	– a arrived in r_0
	...	..., x1, wrTl	– b arrived in r_1
alloc:	`ALLOC`	2, +1	– Allocate list cell
	`COPYP`	r_2, r_2, +1	– Reference in r_2, half weight
	`RETURN`	x2, +1	– Return cell reference
	`COPYP`	x2, x2, +1	– Half weight of r_2
	`SPLIT`	wrHd, wrTl	– Write head and tail in parallel
wrHd:	`WAIT`	e_1, +1	– Wait for a and cell
	`WRITEI`	r_0, r_2, 0	– Head $r_2 := r_0$
	`WAIT`	e_3, term	– Join with tail path
wrTl:	`WAIT`	e_2, +1	– Wait for b and cell
	`WRITEP`	r_1, x2, 1	– Tail $r_2 := r_1$
	`WAIT`	e_3, term	– Join with head path
term:	`TERMINATE`		– Terminate activation

As soon as the body of the function is activated (i.e. the result context has arrived), the new list cell may be allocated and its reference be written into register r_2. However, before this reference may be delivered as the result of the activation, its weight has to be divided by 2, which is done by the `COPYP` instruction (for details on the garbage collection process, see Section 4.5.4). The reference with the updated weight is returned to the caller of the activation. Since the reference will be used by two independent `WRITE` operations, its weight has again to be subdivided. Afterwards, the actual task is split into two parallel paths, each of which is to be synchronized with the arrival of the contents of the list cell's head and tail, respectively.

Each of the pathes triggered by the arrival of the values a and b has to be synchronized with the allocation of the list cell. Then the values (an integer atom and a list pointer) are simultaneously written into the corresponding field (head or tail) of the cell. Afterwards, both pathes join together and the activation may terminate.

For the preservation of the EVE non-strictness properties, it is absolutely necessary that the head and tail of a list cell are written in parallel. If the contents of one field is already available but the other field has not yet been defined, a process referring the already defined field must not be blocked. However, in this case any sequentialization of the writing processes might will prevent one field to be written because the contents of the other field have not yet arrived!

Chapter 6

The ADAM Implementation and Evaluation

This chapter contains a general overview on the prototype implementation of the ADAM on a conventional parallel machine. The basic features and restrictions of this implementation are sketched and the performance and speedup of several parallel benchmark programs are presented. The full source code of this prototype implementation is given in Appendix C.

6.1 The ADAM Implementation

The ADAM abstract machine has been especially designed for an efficient simulation on conventional parallel computers. A prototype implementation of an ADAM interpreter has been written for the IMPULS 2400, a transputer-based multiprocessor installed at RISC-Linz. The machine consists of 16 nodes each of which is equipped with an INMOS T800 32-bit processor with a performance of about 10 Mips ([Inmos, 1988b]) and 4 MB of local RAM. Communication between processors is provided by bidirectional serial links at a speed of about 20 Mbit/s. The topology of the machine may be dynamically configured before the start of program execution.

The ADAM **interpreter** is totally written in OCCAM 2 ([Inmos, 1988a]) and is organized in such a way that each transputer simulates one ADAM module. The major components of a module (arbiter, processor unit and memory unit) are implemented by parallel processes sited on the corresponding transputer. The channels between ADAM modules correspond to links between transputers where each bidirectional link is able to represent two unidirectional channels. Since each transputer has four links, only ADAM topologies are possible where each module is connected to at most four neighbours.

The only part of the EVE **compiler** that has been implemented up to now is the final pass generating ADAM object code from ADAM assembly language. This part has been (and the whole compiler will be) written in TOPSPEED MODULA-2 ([Jensen, 1988]) and is operational on a DOS based PC/AT with an 80386 processor. With the help of this assembly program, the user may now manually create ADAM assembly code that is translated into object code and executed by the ADAM interpreter.

Currently, the ADAM&EVE system can be used as illustrated in Figure 6.1 (the following figures describe each step with the help of a sample program for fibonacci number generation):

1. **Program**

Figure 6.1: Current State of the ADAM&EVE System

A text file is written holding the ADAM assembly code for each user-defined function of a program (see Figure 6.2). The assembly code may contain symbolic function names, operation codes and jump labels, comments and other syntactic sugar for a simplified program development. The assembly program processes this file, resolves the symbolic references and creates an ADAM object file, the **program file**.

2. **Expression**

   A text file is written holding the ADAM assembly code for the expression to be evaluated within the scope of a program (see Figure 6.3). This assembly code may refer to the functions defined in the program file described in the previous paragraph. The assembly program processes this file, resolves the symbolic references and creates an ADAM object file, the **expression file**. The seperation between program file and expression file allows to submit several expressions for evaluation without each time reloading the basically same program to the transputer network.

3. **Type**

   A text file is written describing the type of the result computed by the expression described above (see Figure 6.4). This **type descriptor file** allows in a simple way the (recursive) definition of any type that may be expressed in an EVE program. It is used to direct the result generator during the execution of the program as described below.

4. **Execution**

   Program file, expression file and type descriptor file are submitted to the ADAM interpreter running on the transputer system. This is done, according to an interpreter option, either in interactive or in batch mode. The interpreter distributes program file and expression file over all transputers in the network and submits the type descriptor file to the result generator on the master module. The generator triggers the start of the execution and, directed by the type descriptor, demands the components of the result from the network. The result is, simultaneously with program execution, printed to the screen and written into a log file (see Figure 6.5).

5. **Statistics**

   Depending on an interpreter option, some kind of runtime statistics may be generated and written into a file (see Figure 6.6). This statistics illustrates the utilization of each processor unit and each memory unit of the network. It may be used in order to learn about the dynamic behaviour of the program and to detect insufficiencies and bottlenecks of the implementation.

In the final system, the user will of course be freed from these low-level details. He will submit an EVE source file to the compiler that will generate the ADAM program file. An dialogue shell will interactively allow the user to enter expressions to be evaluated in the scope of this program. From these expressions, the compiler will generate expression files and type descriptor files that will be successively submitted to the ADAM interpreter that will evaluate them and print the results to the screen.

The following subsections provide some internal details of the implementation, in particular the construction of tokens, stores, etc.

6.1.1 Tokens

Each kind of token in the ADAM has equal length of 11 bytes. The first byte is reserved for the token's address field (an ADAM network may therefore consist of up to 256 modules), the following one for its type field. The remaining 9 bytes contain the piece of information the token is carrying and are structured as shown in Figure 6.7.

```
PROGRAM Fibonacci

EXPORT Fib                      ; function Fib

; ---------------------------------------------------------------------------
; Fib[n]: INT[INT] ==
;   IF n <= 1
;     THEN 1
;     ELSE Fib[n-1] + Fib[n-2]
; ---------------------------------------------------------------------------
FUNCTION Fib(3,4)               ; maximum event and register number

        JUMP      start          ; start trigger path
        COPYI     1, 2, start    ; receive n in r1 and r2

start:  WAIT      0, +1          ; synchronize trigger with n
        COPYI     3, 2, +1       ; r3 := n
        LESSEQC   1, 3, +1       ; r3 := n <= 1
        SWITCH    3, true, false ; if n <= 1 then true else false

error:  RETURN    3, term        ; return ERROR and terminate

true:   RETURNC   1, term        ; return 1 and terminate

false:  SPLIT     fib1, fib2     ; compute Fib[n-1] and Fib[n-2] in parallel

fib1:   SUBC      1, 1, +1       ; r1 := r1-1
        FASTCALL  Fib, +1        ; call Fib
        COPYI     3, 3, send1    ; r3 := address Fib
        COPYI     1, 1, add      ; r1 := result Fib
send1:  SEND      1, 3, 1        ; send r1 to r3 as n
        WAIT      1, end         ; synchronize with send2

fib2:   SUBC      2, 2, +1       ; r2 := r2-2
        FASTCALL  Fib, +1        ; call Fib
        COPYI     4, 4, send2    ; r4 := address Fib
        COPYI     2, 2, add      ; r2 := result Fib
send2:  SEND      2, 4, 1        ; send r2 to r4 as n
        WAIT      1, end         ; synchronize with send1

add:    WAIT      2, +1          ; wait for r1 and r2
        ADD       1, 2, +1       ; r1 := r1 + r2
        RETURN    1, end         ; return r1

end:    WAIT      3, term        ; synchronize add with send1/2
term:   TERMINATE                ; terminate activation

END Fib

END Fibonacci
```

Figure 6.2: Fibonacci Program Source File

```
; --------------------------------------------------------------------------
;
; Fib21.ASE                                               Wolfgang Schreiner
;                                                         (c) RISC-Linz 1990
; Compute the 21st Fibonacci Number
;
; --------------------------------------------------------------------------
EXPRESSION Fib21

IMPORT Fib              ; function Fib

BEGIN (2,3)             ; maximum event and register number
    TAILCALL  FIB, +1   ; tail-recursive call of Fib
    CONTEXT   0, +1;    ; address of Fib in r0, submit own context
    SENDC     21, 0, 1  ; submit 21 to Fib
    TERMINATE           ; terminate activation
END Fib21
```

Figure 6.3: Fibonacci Expression Source File

```
; --------------------------------------------------------------------------
;
; Fib21.ADT                                               Wolfgang Schreiner
;                                                         (c) RISC-Linz 1990
; Type Description for Fibonacci Number Generator
;
; --------------------------------------------------------------------------

; root line
#0: 1       ; activate function 0, main type is labeled with 1

; main type
 1: I       ; main type is INT
```

Figure 6.4: Fibonacci Expression Type File

```
This is ADAM V1.0 (16 processors used).
Enter name of program file (.ADM): FIBONACC
Enter name of expression file (.ADE): FIB21
(\USR\SCHREINER\ADTEST\FIBONACC.ADM)(\USR\SCHREINER\ADTEST\FIB21.ADE)(\USR\SC
HREINER\ADTEST\FIB21.ADT)
Program started.
17711
Result received (9.8 s).
Program terminated (9.8 s).
Another Expression (y/n)? n
Another Program (y/n)? n
Bye.
```

Figure 6.5: Fibonacci Execution Log File

6.1.2 Program Store

The program store consists of 32-bit words each of which may be interpreted as an

- Instruction,
- Code Block Address,
- Resource Descriptor,
- Long Constant.

These types of program words are now described in greater detail (see Figure 6.8):

- **Instructions:**

  The most significant byte of the word holds the operation code of the instruction, the operand word of the instruction is sited in the remaining three bytes. The various fields of an operand word (see Section A.3.2) are always aligned to byte boundaries, which has the following consequences:

  - *Register Descriptors* occupy 1 byte, i.e. a function activation may address at most 256 registers.
  - *Instruction Offsets* occupy 1 byte, i.e. the distance between an instruction and its successor may be at most 256 words[1].
  - *Small Constants* occupy 1 byte, i.e. they may represent values between -128 to $+127$.
  - *Event Descriptors* occupy 2 bytes, i.e. a function activation may use at most 65535 synchronization events.
  - *Function Descriptors* occupy 2 bytes, i.e. a program may consist of maximal 65535 code blocks.

- **Code Block Addresses:**

  These addresses occupy a whole 32-bit word. Hence, programs may consist of up to $2^{32}-1$ instruction words.

[1] Longer Jumps can be done by the `JUMP` instruction (see Section B.2.4).

```
Proc  Frame Store Utilization            Request Store Utilization
----------------------------------------------------------------------------
P8     4.4 *                             5E-02
P8     5.4 **                              0.2
P8     6.7 **                            5E-02
P8     7.5 **                              0.1
P2     3.5 *                             1E-01
P2     3.9 *                             2E-02
P3    15.8 *****                          10.7 ***
P8     8.4 ***                           2E-02
P8    10.3 ***                           1E-01
P8    11.5 ***                             0.9
P8    12.8 ****                            1.6
P8    14.2 ****                            2.6 *
P8    15.8 *****                           4.0 *
P4     2.3 *                              12.3 ****
P8    17.6 *****                           5.0 **
P8    19.5 ******                          6.0 **
P8    21.7 *******                         7.0 **
P8    24.1 *******                         9.0 ***
P2     4.4 *                             2E-02
P4     2.6 *                              14.5 ****
P4     2.9 *                              18.3 *****
P4     3.2 *                              21.0 ******
P4     3.5 *                              23.1 *******
P4     3.9 *                              24.5 *******
P2     3.9 *                             2E-02
P2     3.5 *                               0.5
P5     9.3 ***                           2E-02
P2     3.9 *                               6.8 **
P2     4.4 *                               8.4 ***
P4     4.4 *                              23.9 *******
P2     4.9 *                               9.7 ***
P2     5.4 **                              8.1 **
P2     6.1 **                              6.6 **
P6     4.4 *                               7.9 **
P2     6.7 **                              4.1 *
P2     7.5 **                              0.8
P4     4.9 *                              30.0 *********
P4     5.4 **                             31.1 *********
P7     5.4 **                              2.1 *
P7     6.1 **                              3.0 *
P7     6.7 **                              3.7 *
P4     6.1 **                             35.1 ***********
P7     7.5 **                              1.1
P4     6.7 **                             33.3 **********
P7     8.4 ***                           2E-02
P4     7.5 **                             31.3 *********
P4     8.4 ***                            29.6 *********
P7     9.3 ***                           2E-02
```

Figure 6.6: Fibonacci Execution Statistics (Excerpt)

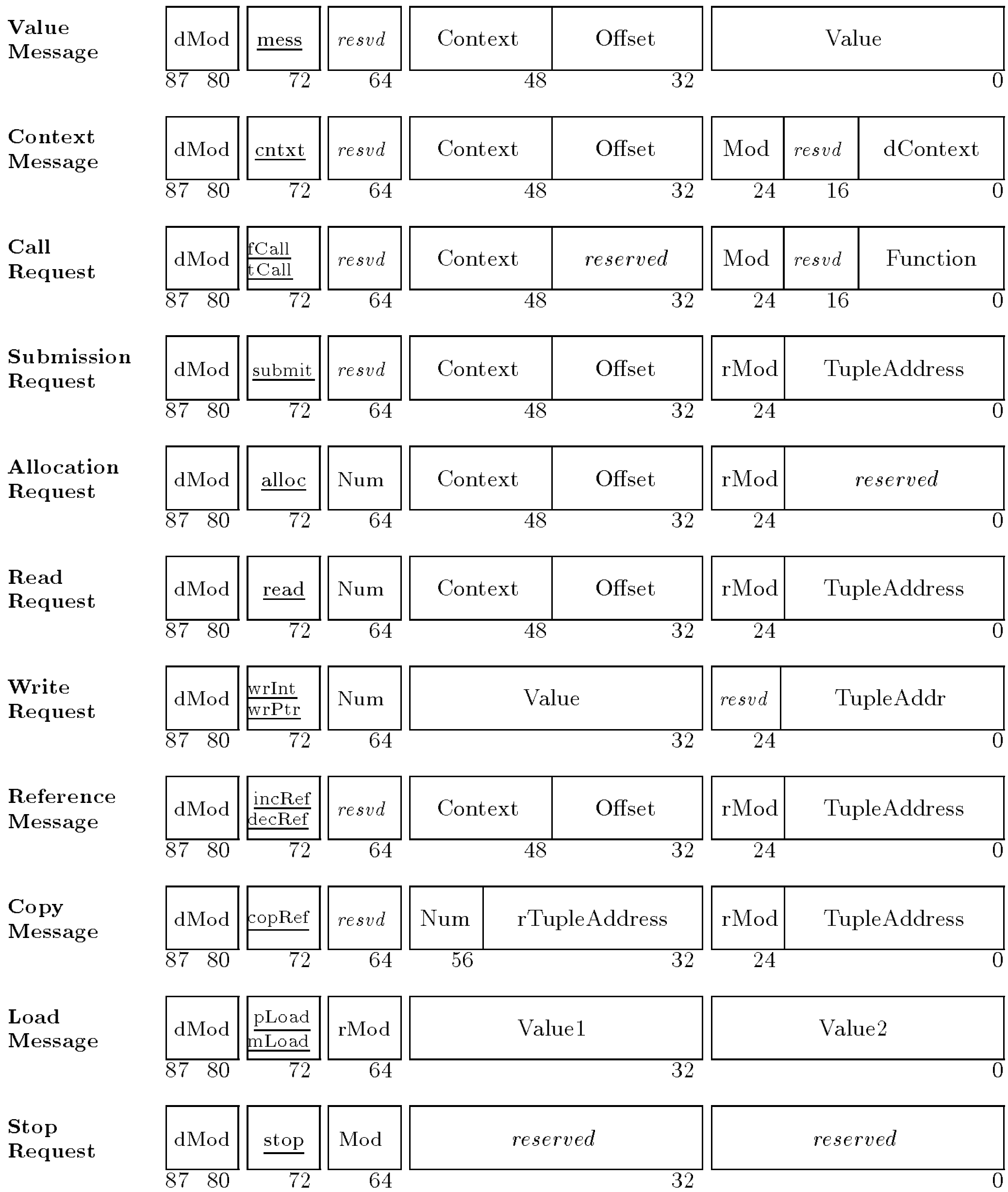

Figure 6.7: Token Formats

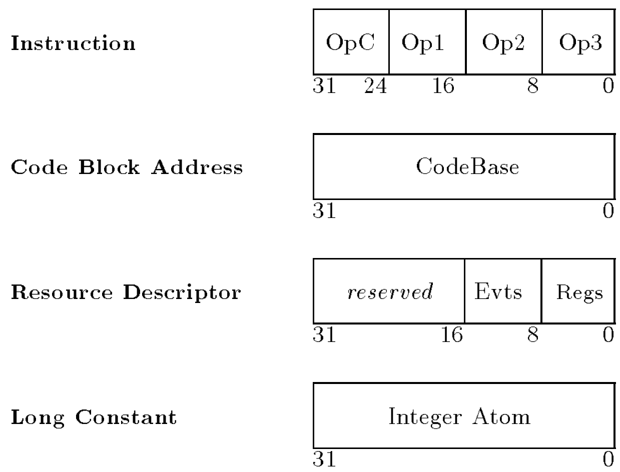

Figure 6.8: Program Store Format

- **Resource Descriptors:**

  The least significant byte of a resource descriptor holds the number of registers that each activation of this function requires. The next byte represents the number of of words required for each activation frame to represent the synchronization events used.

- **Long Constants:**

  A long constant occupies a full word and can therefore represent an *integer atom* (see Section 6.1.6).

6.1.3 Frame Store

The frame store holds a number of *activation frames* that consist of 32-bit words, where each word is able to represent a

- Register,
- Result Address,
- Result Module,
- Set of Synchronization Bits.

These types of frame word are now described in detail (see Figure 6.10):

- **Register:**

  A register is able to hold an atomic value, i.e. basically an integer atom or a tuple reference (see Section 6.1.6).

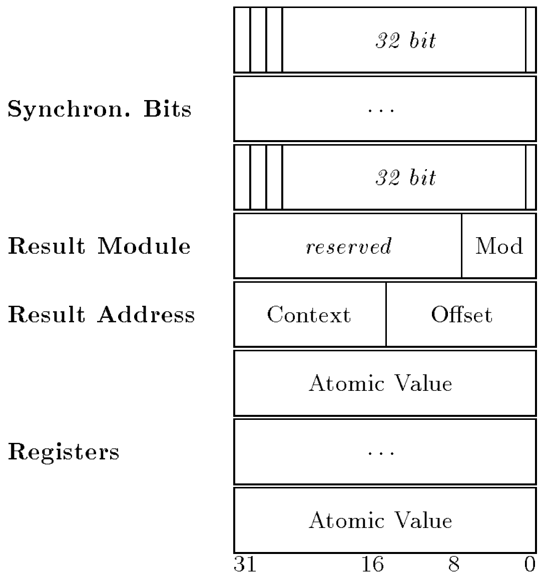

Figure 6.9: Activation Frame Format

- **Result Address:**

  The more significant halfword of the result address holds a reference to the context that describes the activation to which the result value of the actual activation is to be delivered. The less significant halfword hold the address of the instruction within that activation that shall receive the result.

- **Result Module:**

  The address of the module to which the result value is to be sent is stored in the least significant byte of the word.

- **Synchronization Bits:**

  Each word is able to hold 32 synchronization bits each of which may be tested and inverted with the `WAIT` instruction (see Section B.2.5).

6.1.4 Context Store

The context store consists of 64-bit double words each of which represents a *context*, i.e. the description of a particular function activation (see Figure 6.10):

The less significant 32-bit word of the context holds the base adress of the function activation's code block, the more significant word holds the base address of its frame. The frame store as well as the program store may therefore consist of maximal $2^{32} - 1$ words.

FrameBase	CodeBase
63 32	0

Figure 6.10: Context Format

Context	Offset
31 16	0

Figure 6.11: Trigger Format

6.1.5 Trigger Store

The trigger store consists of 32-bit words each of which represents a *trigger*, i.e. the description of an executable task (see Figure 6.11):

The more signifcant 16-bit halfword of the trigger holds the number of the context describing the activation in which the task may be executed. There may therefore exist up to $2^1 6$ concurrent activations on a processor unit.

The less significant halfword of the trigger holds the address of the instruction within the code block of the denoted activation. A code block may therefore consist of maximal $2^1 6$ instructions.

6.1.6 Atomic Values

An atomic value occupies a 32-bit word where the least significant bit always holds an error flag indicating whether the contents of the word are valid or not. The remaining 31-bit may be interpreted as an

- Integer Atom,
- Tuple Reference,
- Activation Address,
- Curried Function.

These types of atomic values are constructed as follows (see Figure 6.12):

- **Integer Atom:**

  The 31-bit data part of an atomic value is able to represent integer values in the range from -2^{30} to $+2^{30} - 1$.

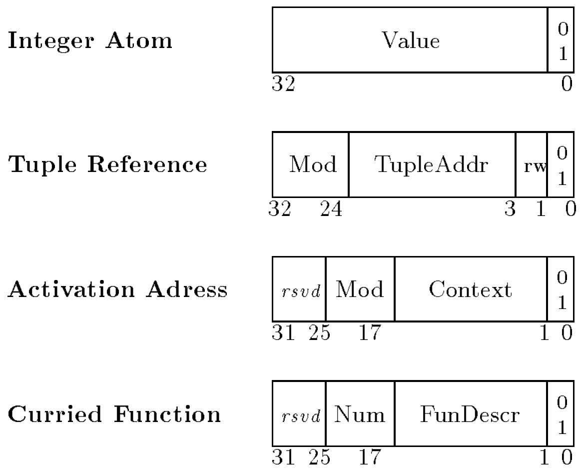

Figure 6.12: Atomic Values Format

- **Tuple Reference:**

  Both of the least significant bits of the data part hold the weight of the reference. The weight is therefore in the range from $2^0 = 1$ to $2^4 = 16$. The following 21 bits hold the actual address of the referenced tuple. The local heap of each memory unit may therefore contain up to 2^{21} cells. The most significant byte of the word holds the address of the module on which the referenced tuple is is sited.

- **Activation Address:**

  The least significant halfword of the data field holds the context number of the activation. The following byte holds the address of the module on which the activation has been created.

- **Curried Function:**

  The less significant halfword of the data field holds the description of the curried function. The following byte holds the number of the function parameter that the first of the curried arguments corresponds to.

6.1.7 Heap Cells

The heap store of a memory unit consists of 80-bit cells. Figure 6.13 depicts the construction of such a cell:

- Both of the two less significant 32-bit words represent the *head* and *tail* data fields.
- The most significant 16-bit halfword represents the *tag* of the cell carrying system-internal information:
  - The more significant byte of the halfword holds the reference count of the tuple. Hence, a maximal reference count of 256 can be represented.

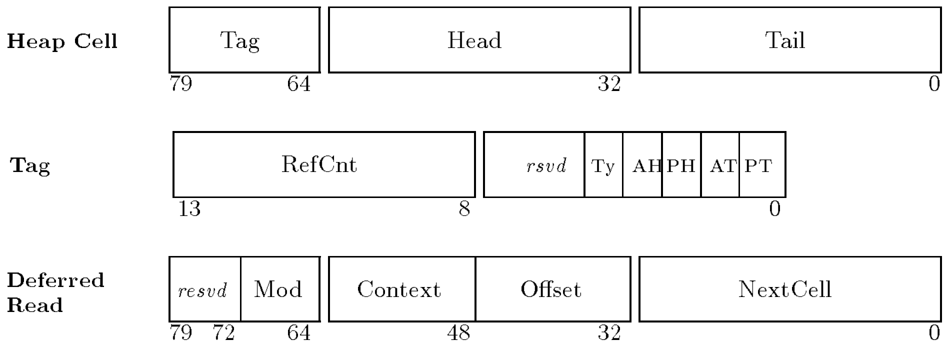

Figure 6.13: Heap Cell Format

- The least significant two bits of the tag hold system information on the tail field, i.e. the *available flag* and the *pointer flag* of the tail.
- The following two bits hold system information on the head field, i.e. the *available flag* and the *pointer flag* of the head.
- The next bit holds the *type flag* of the cell. This bit indicates whether the cell is open (i.e. has a successor reference stored in the tail) or closed (i.e. has some user datum stored in the tail).

Deferred read requests are stored in a heap cell as illustrated in Figure 6.13:

- The module address of the instruction to that the demanded value is to be delivered is stored in the less significant byte of the tag field.
- The context and the offset of the instruction are stored in two consecutive halfwords of the head field.
- The pointer to the next of the list of deferred read requests is stored in the tail field of the heap cell.

6.1.8 Summary of Implementation Restrictions

Table 6.14 gives a summary of several restrictions imposed by the current implementation of ADAM on our transputer network.

6.2 The ADAM Evaluation

A variety of ADAM assembly programs has been written in order to investigate the following criteria of our prototype implementation:

1. the correctness of the interpreter,
2. its absolute performance,

Number of Modules	≤ 256
Length of Program Store	$\leq 2^{32}$
Length of Activation Store	$\leq 2^{32}$
Length of Context Store	≤ 65536
Number of Code Blocks	≤ 65536
Length of Code Blocks	≤ 65536
Integer Values i	$-2^{30} \leq i \leq +2^{30} - 1$
Small Constants c	$-128 \leq c \leq 127$
Distance to Successor	≤ 256
Registers per Activation	≤ 256
Events per Activation	≤ 65536
Heap Cells per Module	$\leq 2^{21}$
Pointer Reference Weight	$\leq 2^{4}$
Tuple Reference Count	≤ 256

Figure 6.14: Implementation Restrictions

3. the relative speedups achieved.

1. **Correctness**

   We have been able to debug the prototype of our interpreter in a relatively short time. It is now such stable that new errors can only very rarely be detected. However, many more (and also larger) programs have to be written in order to test all features of the implementation more thoroughly. One major weakness known but not yet debugged is the fact that under certain circumstances the interpreter may, after the delivery of the program result, run into a deadlock when it tries to terminate all processors of the network.

2. **Performance**

   Our main concern in the implementation of the interpreter has been a simple and clear program design suitable for further debugging and maintenance. Hence, we have attached importance only to the optimization of a few critical parts of the program, e.g. the execution unit of the ADAM processor. However, the overall performance of the system is surprisingly high:

   At RISC-Linz, we have also installed a commercial transputer implementation of STRAND88, a parallel language which includes several principles of dataflow programming ([Foster and Taylor, 1989]). First comparisons with this implementation show that the average execution time of a STRAND88 program is about 50% of a comparable ADAM program. However, in the case of an extensive use of non-strict data structures (like the difference lists mentioned below), ADAM beats STRAND88 with a factor of more than 10!

3. **Speedup**

   Several parallel benchmark programs have been written and show rather promising speedups. Among them, there are the programs

   - FIBO: The doubly recursive fibonacci number generator presented in Section 6.1.
   - PRIMES0: A prime number generator using a lot of intermediate lists (described in Section 2.4.1).
   - PRIMES1: A prime number generator using difference lists (described in Section 2.4.2).

Program	Input	1T / 2W Ring	2T / 2W Ring	4T / 4W Full	8T / 8W Chordal Ring	16T / 16W Torus
FIBO	20	54(1.0)	28(1.9)	16(3.3)	10(5.4)	7(7.7)
PRIMES0	2000	302(1.0)	163(1.8)	93(3.2)	59(5.1)	41(7.4)
PRIMES1	10000	237(1.0)	127(1.9)	88(2.7)	65(3.7)	51(4.6)

Table 6.1: ADAM Benchmarks

Some of the results are given in Table 6.1. Each column of the table represents one particular processor configuration with the number of transputers (T) used, the number of worker modules (W) simulated and the topology of the transputer network listed above. Each item in the table denotes the elapsed time in seconds, the values in parentheses show the relative speedups achieved.

The efficiency (speedup/processor ratio) ranges in both of the first test programs from more than 90% to about 50% in the worst case. However, these results have been gained with varying topologies. They may by somewhat misleading since we do not yet know which topology shows the best properties for the ADAM architecture.

The third test program, the prime number generator PRIMES1, is in absolute terms much faster than the PRIMES0 program. This is not very surprising since PRIMES0 is extremely slowed down by allocating a lot of superfluous auxilary lists. However, the relative speedups achieved by PRIMES1 are signifcantly lower than for PRIMES1. One of the main reasons may perhaps be the fact that all processors access the same list of primes. Those processors on that the first five or six elements of this list are stored, have to fulfill many more read requests and are therefore much more loaded than the other processors. However, we still have to investigate this effect more thoroughly.

First investigations have shown that an effective load balancing is currently one of the main problems of the ADAM architecture. The tools used up to now are too rough to keep the overall load of the network on an equal level. Many inpredictible ups and downs in the workload of each processor cause a lot of superfluous network traffic since too many activation requests are sent to neighbour processors (instead of handled by the actual processor). This property often results in extreme deviations in the runtime of the same program (up to 100%!). As an extreme example, the computation of the 20^{th} fibonacci number once took more time than the computation of the 21^{th} one! Thus, there remains a lot of work to be done in order to achieve better balancing strategies. However, the difficulty of load balancing is a general feature of all true dataflow architectures, too.

In the near future, we will exhaustively test the ADAM concept using various processor configurations in order to learn more about the dynamical behaviour of the machine (which is many aspects still rather mysterious to us). We will investigate several methods for load balancing and will try to improve the overall performance of the system by removing some existing bottlenecks.

The project ADAM&EVE is continued by another diploma thesis ([Loidl, 1992]) dealing with the implemenation of a compiler for the EVE language. In the frame of this work, we will also experiment with several other parallelelization and synchronization strategies than the pure dataflow principle. Such experiments are possible because parallelization and synchronization are handled by the code generation phase of the compiler and are not fixed in the ADAM architecture itself. We are confidential that a lot of interesting results will be gained in the future.

Appendix A

The ADAM Operational Behaviour

We define the operational semantics of the ADAM machine by an algorithm that describes the behaviour of each component of an ADAM worker module. In particular, the semantics of a processor unit is defined by an interpreter that executes ADAM machine code. We believe that this form of presentation may be adequate for two reasons:

1. While the main concern of the semantics definition of an abstract machine is of course its static behaviour (i.e. the mapping of programs to the computed result values), we feel that also the dynamical behaviour of the machine is of particular interest. However, this runtime behaviour is more adequately described by an algorithm than by some form of static definition (as in denotational semantics).
2. The algorithm may serve as the basis for an implementation of the machine, either in hardware or in software. Actually, it already defines the main parts of an interpreter simulating the ADAM machine on some kind of conventional computer. Since this is one of our preliminary goals, the algorithmic definition may help to save time and effort and, more important, to preserve correctness of the implementation.

Actually, we have *first* defined the operational behaviour of the machine by the algorithms presented in the following subsections and *then* implemented a prototype of ADAM by the use of this algorithmic definition. Even if some details have been implemented in a slightly different (i.e. more efficient but semantically equivalent) way, this description has proved to be a valuable tool.

A.1 Notation

Before we start to describe the operational behaviour of the machine, we will give a short introduction into the notation that we use to describe the algorithms in the following sections.

The algorithms are presented in a PASCAL-like syntax where following typesetting conventions have been obeyed:

- **Keywords** are written in boldface style in all lower case (e.g. **case**).

- **Functions** appear in roman style with upper case initials (e.g. 'Execute'). If the function is not refined any more but handled as a "black box", then it is written in italic style (e.g. *AllocFrame*).
- **Constants** appear in underlined style, enumeration type constants in all lower case (e.g. incRef), other constants with upper case initials (e.g. LinkNum).
- **Types** appear in Roman Style with upper case initials (e.g. 'Word').
- **Variables** appear in Roman Style in all lower case (e.g. 'offset').

These conventions will also be used in the ordinary text explaining the algorithms. However, in this situtation, functions, types and variables are enclosed by single quotes to distinguish the "meta-level" identifers from "object-level" text (e.g. the variable 'offset' as opposed to an instruction offset).

Besides the usual PASCAL-like constructs the following additional language constructs will be used:

- **Record Types** are construed by

  $$Type_1 \times \ldots \times Type_n$$

  with $Type_i$ being the type of the i^{th} component of the record.

- **Record Values** are constructed by

  $$\langle Value_1, \ldots, Value_n \rangle$$

  with $Value_i$ being the i^{th} component of the record.

- **Union Types** are constructed by

  $$Type_1 \mid \ldots \mid Type_n$$

  with $Type_i$ being the type of the i^{th} variant of the union.

- **Type Coercions** are denoted by

  $$Value\colon Type$$

  with the meaning that the object *Value* is in some sensible way[1] converted into an object of type *Type*.

- The **Token** _ is used for two purposes:
  - On the left hand side of assignment statements, such as in

    $$\langle \mathrm{var}_1, \mathrm{var}_2, \_ \rangle := \langle \mathrm{val}_1, \mathrm{val}_2, \mathrm{val}_3 \rangle$$

    it indicates that the value of the corresponding tuple field (in this case 'val_3') is **not required**.
  - On the right hand side of assignment statements, such as in

    $$\langle \mathrm{var}_1, \mathrm{var}_2, \mathrm{var}_3 \rangle := \langle \mathrm{val}_1, \mathrm{val}_2, \_ \rangle$$

    it indicates that the value of the corresponding tuple field (in this case 'var_3') is **not changed**.

type

Number = *Unbounded Integer Number*
Boolean = (true, false); (* Logical Values *)
Bit = (zero, one); (* Basic Storage Unit *)
Bits(n) = **array** [0 ... n−1] **of** Bit; (* BitField *)

Figure A.1: Basic Types

Figure A.1 shows the basic types that are used in the following sections:

- 'Number' denotes the set of all (positive and negative) integer numbers. These numbers are not restricted by any bound. If they are used in numerical computations, no overflow may occur.
- 'Boolean' denotes the set of the logical truth values true and false. The usual logical connectives are defined on that type.
- 'Bit' denotes the basic unit of storage. In contrast to 'Boolean', no operations are predefined on that type.
- 'Bit(n)' defines a generic type that depending on the value of the number 'n' denotes a storage field of 'n' Bits. All storage classes in the ADAM architecture will be derived from this type.

A.2 The Arbiter

Network Quantities

Figure A.2 shows the definition of some network quantities that are globally used in an ADAM module. They describe the interface between various modules and between the units of a module.

- ModNum is the total number of all modules in the network (exclusively the master module), i.e. all valid module addresses are in the range 0 to ModNum (the master has address 0). LnkNum denotes the number of modules that the local module is directly connected to (i.e. the number of the neighbour modules). ActMod is the address of the actual worker module considered. All of these "constants" have to be initially defined during the setup of the network.
- NetWdLen and ModWdLen define the bit length of each word transmitted by 'external" links (i.e. by links between different modules) and by "internal" links (i.e. by links from the arbiter to the local processor unit and memory unit, respectively).

The following type definitions use these constants to define some corresponding address and link types as well as the structure of tokens transmitted by these links:

[1] How these coercions are handled is of little importance for the logical behaviour of the machine and for the moment neglected. Chapter 6 describes how these low-level facilities are handled in our prototype implementation.

const

```
LinkNum   = Number of Modules the Actual Module is Connected to;
ModNum    = Number of Modules in the Network (exclusively Master);
ActMod    = Address of the Actual Module Considered;

NetWdLen  = Length of Words Transmitted by Network;
ModWdLen  = Length of Words Transmitted between Units within a Module;
```

type

```
NetLink  = Bits(NetWdLen); (* Link between Modules *)
ModLink = Bits(ModWdLen); (* Link to Processor/Memory Unit *)

ModAddr = [0..ModNum]; (* Module Addresses *)
LinkAddr = [1..LinkNum]; (* Links leaving the Module *)

(* Network Tokens *)
Token      = ModAddr × TokenType × TokenInfo;
TokenType = ProcType | MemType;
ProcType  = (mess, cont, fCall, tCall, pLoad, stop);
MemType   = (alloc, read, wrInt, wrPtr, submit, mLoad, incRef, decRef, copRef, stop);
TokenInfo  = ProcToken | MemElement; (* Token Info Field *)
```

var

```
(* External and Internal Links *)
inpLink, outLink: array LinkAddr of NetLink; (* External Links *)
inpProc, inpMem: NetLink; (* From Processor and Memory *)
outProc, outMem:ModLink; (* To Processor and Memory *)
```

Figure A.2: Network Quantities

- A 'Token' is the high-level interpretation of a word transmitted by the network. It consists of an address field 'ModAddr', a type field 'TokenType' and the field 'TokenInfo' carrying the actual information of the token. Since tokens may be destined for processor units and memory units, 'TokenType' is the union of 'ProcType' and 'MemType' as well as 'TokenInfo' is the union of 'ProcInfo' and 'MemInfo'. The detailled interpretation of these fields can be found in Section A.3.2 and Section A.4.2, respectively.

The variable definitions demand some further explanation, too:

- 'inpLink' and 'outLink' are both arrays that describe the communication links that enter respectively leave the module. Assignment statements of the form

  outLink[i] := word
  word := inpLink[j]

  will be interpreted as acts of communication with the neighbours of the actual module.

- 'inpProc' and 'inpMem' are the links by that the arbiter receives network tokens from the processor unit and the memory unit, respectively. These 'internal' links are constructed in exactly the same way as the 'external' links that establish the connection between different modules.

- 'outProc' and 'outMem' denote the links through that the arbiter may send tokens to the processor unit and the memory unit, respectively. In contrast to the other links, tokens on these links do not carry module addressses any more, since they have already found their destination module. Therefore, these links are defined in a different way.

A.2.1 The Arbiter Operation

Figure A.3 describes the operational behaviour of the arbiter:

- The arbiter holds a local routing table 'route' that provides a mapping from module addresses to link numbers. Indexed with the address of a module, it delivers the number of the link through that tokens destined for this module have to be forwarded. This table has to be initialized first.

- Before program execution may begin, the whole program code has to be loaded into the program store of each memory unit. The arbiters are responsible for the correct distribution of that program code among all worker modules. We leave the details of this system setup undefined.

- The procedure *ReadyLink* listens on all of the (internal and external) input channels of the arbiter and returns the word 'netWd' that has been received by one of these channels. We do not define the strategy that this procedure uses for the serialization of "simultaneous" transmission requests, but some kind of priority among internal and external links will probably have to be obeyed.

- The destination address 'mod' of 'netWd' is examined in order to determine whether the token has already arrived in the destined module. In this case, the field 'type' determines whether the token has to be forwarded to the module's processor unit or to its memory unit. In either of these cases, the module address has become superfluous and is not transmitted with the rest of the token. If the token is on its way to another module, the arbiter has to

Arbiter

```
var
    route: array ModAddr of LinkAddr;
    netWd: NetLink;
    mod: ModAddr;
    type: TokenType;
    info: TokenInfo;
    out: LinkAddr;
    terminate: Boolean;

Initialize route;
Transfer Program into Processor Unit and Forward it to Other Units;

terminate := false;
repeat
    netWd := ReadyLink(inpLink, inpProc, inpMem);
    ⟨mod, type, info⟩ := netWd:Token;
    if mod = ActMod then
        case type of
            mess, cont, fCall, sCall, pLoad:
                outProc := ⟨type, info⟩:ModLink;
            alloc, read, wrInt, wrPtr, submit, incRef, decRef, mLoad:
                outMem := ⟨type, info⟩:ModLink;
            stop:
                outProc := ⟨type, _⟩:ModLink;
                outMem := ⟨type, _⟩:ModLink;
                terminate := true
        endcase
    else
        out := route[mod];
        outLink[out] := netWd
    endif
until terminate;

Route all stop tokens destined for other modules
```

Figure A.3: Arbiter Operation

forward it through one of the output links ‘outLink’. This communication is only possible if the arbiter on the other side of the link is ready to receive the token. Otherwise the actual arbiter is blocked[2].

- The special token type ‘stop’ signals that the master module has received its result data and that all modules in the network may terminate execution. In this case, the arbiter forwards copies of the tokens to processor and memory unit. It then leaves the transfer loop and handles the transmission of the stop tokens destined for other worker modules. If all worker modules have received their stop tokens, the arbiter may stop its execution[3].

A.3 The Processor Unit

The following sections describe the operational behaviour of the processor unit by an algorithm that interprets ADAM machine code and performs the desired “microcode” actions. Actually for sake of lucidity, the description is split into two parts:

- The following subections define the overall behaviour of the processor unit regardless of the instruction set of the machine.
- Appendix B provides the exact definition of the “microcode” of each instruction.

A.3.1 Global Constants

Figure A.4 shows the constants that are globally known throughout the the following algorithms and subalgorithms.

- **Integer Bounds:**

  The first constants define the upper and lower bounds for the fixed-size types ‘Integer’ and ‘Constant’ that will be defined in the next subsection.

- **Lengths of Storages:**

  The next constants define the lengths of the various types of storages that the processor has access to, i.e. the number of words every storage consists of.

- **Length of Storage Words:**

  Within each type of storage, all words have equal length. However, different storages generally have different word lengths that are fixed by these constants.

- **Restrictions:**

  These constants denote restrictions that arise from the limited size of tokens and storage words. Among them, there are the maximal number of addressable registers per code block (‘MaxReg’) or the maximal address distance from one instruction to its successor instruction (‘MaxSucc’).

```
const

    (* Integer Bounds *)
    MinInt   = Minimal Representable Integer Number;
    MaxInt   = Maximal Representable Integer Number;
    MinCon   = Minimal Representable Small Constant;
    MaxCon   = Maximal Representable Small Constant;

    (* Lengths of Storages *)
    ProgLen   = Length of Program Store;
    FrameLen  = Length of Frame Store;
    ContLen   = Length of Context Store;
    TrigLen   = Length of Trigger Store;
    MessLen   = Length of Message Store;
    RequLen   = Length of Request Store;
    HeapLen   = Length of Memory Unit Heap;

    (* Lengths of Storage Words *)
    ProgWdLen   = Length of Program Store Word;
    FrameWdLen  = Length of Frame Store Word;
    ContWdLen   = Length of Context Store Word;
    TrigWdLen   = Length of Trigger Store Word;
    MessWdLen   = Length of Message Store Word;
    RequWdLen   = Length of Request Store Word;

    (* Restrictions *)
    MaxReg    = Number of Addressable Registers per Activation;
    MaxEvnt   = Number of Addressable Events per Activation;
    MaxSucc   = Maximal Distance to Successor Instruction;
    MaxOff    = Maximal Length of CodeBlock;
    MaxBlock  = Maximal Number of Code Blocks per Processor;
    MaxWght   = Maximal Weight of Pointer;
```

Figure A.4: Global Processor Constants

```
type

    (* Bounded Numbers *)
    Integer  = [MinInt..MaxInt]; (* Representable Integer Number *)
    Constant = [MinCon..MaxCon]; (* Small Integer Numbers *)

    (* Storage Words *)
    ProgWord  = Bits(ProgWdLen); (* Word of Program Store *)
    FrameWord = Bits(FrameWdLen); (* Word of Frame Store *)
    ContWord  = Bits(ContWdLen); (* Word of Context Store *)
    TrigWord  = Bits(TrigWdLen); (* Word of Trigger Store *)
    MessWord  = Bits(MessWdLen); (* Word of Message Store *)
    RequWord  = Bits(RequWdLen); (* Word of Request Store *)

    (* Storage Indizes *)
    FramePtr = [0..FrameLen−1]; (* Pointer into Frame Store *)
    ContPtr  = [0..ContLen−1]; (* Pointer into Context Store *)
    ProgPtr  = [0..ProgLen−1]; (* Pointer into Program Store *)

    (* Components of the Operand Part of an Instruction *)
    Register = [0..MaxReg−1]; (* Addressable Registers *)
    Event    = [0..MaxEvnt-1]; (* Addressable Events *)
    InstOff  = [0..MaxSucc-1]; (* Addressable Successor Instructions *)
    FunDescr = [0..MaxBlock−1]; (* Descriptor of Code Block *)
    JumpOff  = [−BlockOff..+BlockOff]; (* Jump Offsets within Code Blocks *)

    (* Operand Formats *)
    OpReReIn  = Register × Register × InstOff;
    OpCoReIn  = Constant × Register × InstOff;
    OpReInIn  = Register × InstOff × InstOff;
    OpReReCo  = Register × Register × Constant;
    OpCoReCo  = Constant × Register × Constant;
    OpReIn    = Register × InstOff;
    OpCoIn    = Constant × InstOff;
    OpInIn    = InstOff × InstOff;
    OpFuIn    = FunDescr × InstOff;
    OpEvIn    = Event × InstOff;
    OpJu      = JumpOff;

    (* Instruction Formats *)
    OpCode    = All Possible Operation Codes (Appendix B);
    OpWord    = OpReReIn | OpReCoIn | OpReInIn | OpReReCo | OpReIn
                | OpCoIn | OpInIn | OpEvIn | OpJu;
    Instruction = OpCode × OpWord;

    (* Contents of Trigger and Context Store *)
    BlockOff = [0..BlockLen−1]; (* Code Block Offset *)
    Trigger  = ContPtr × BlockOff; (* Operation Trigger *)
    Context  = FramePtr × ProgPtr; (* Activation Descriptor *)
```

Figure A.5: Global Processor Types

type

```
(* Function Descriptors *)
CurryFun  = FunDescr × Constant; (* Curried Function Descriptor *)
ActivAddr = ModAddr × ContPtr; (* Activation Address *)

(* Heap References *)
TupleAddr = [0..HeapLen−1]; (* Address of Heap Tuples *)
RefWght   = [0..MaxWght−1]; (* Reference Weight *)
TupleRef  = TupleAddr × RefWght; (* Local Tuple Reference *)
Pointer   = ModAddr × TupleRef; (* Global Tuple Reference *)

(* Message Tokens *)
OpArg    = ContPtr × BlockOff × FrameWord; (* Value Message *)
ConArg   = ContPtr × BlockOff × ModAddr × ContPtr; (* Context Message *)
Argument = OpArg | ConArg; (* Argument Message *)
LoadRep  = LinkAddr × ProcLoad; (* Load Report *)
Message  = Argument × LoadRep; (* Message Token *)

(* Request Tokens *)
Request = ContPtr × BlockOff × ModAddr × FunDescr;

(* Token Sent to Processor Unit *)
ProcToken = Message | Request;

(* Atomic Data *)
Info  = Integer | Pointer | ActivAddr | CurryFun; (* Atomic Information *)
Data = Info × Boolean; (* Information plus Error Flag *)

(* Resource and Load Measures *)
Resource   = Event × Register; (* Activation Resources *)
ContLoad   = [0..ContLen]; (* Load of Context Store *)
TrigLoad   = [0..MessLen]; (* Load of Trigger Store *)
MessLoad   = [0..MessLen]; (* Load of Message Store *)
FrameLoad = [0..FrameLen]; (* Load of Frame Store *)
RequLoad   = [0..RequLen]; (* Load of Request Store *)
LoadDescr  = FrameLoad × RequLoad; (* Load Descriptor *)

(* Processor Actions *)
ProcAction = (execute, receive, request, report);
```

Figure A.6: Global Processor Types (Continued)

A.3.2 Global Types

Figures A.5 and A.6 show the types that are used throughout the following algorithms.

- **Bounded Numbers:**

  'Integer' and 'Constant' denote finite subsets of the infinite type 'Number' both of which bounded by some constant upper and lower bounds. 'Integer' is the basic kind of information supported by the ADAM architecture, while 'Constant' represents a rather small subsect of 'Integer' that is used to optimize many of the integer operations occurring in programs.

- **Storage Words:**

  These types define the basic entities that the various classes of storages are constructed from.

- **Storage Indizes:**

  These types denote the various kinds of addresses by that each word in the program store, the frame store or the context store may be directly referenced. The other storage classes are organized in a different way (see Section A.3.3).

- **Components of the Operand Part of an Instruction:**

  These types define the components that the operand part of every instruction (i.e. the instruction word without the operation code) is constructed from. The components may contain constants ('Constant'), slots within the actual activation frame ('Register'), addresses of synchronization points within the frame ('Event'), offsets to successor instructions ('InstOff'), function descriptors ('FunDescr') instruction offsets for large jumps ('JumpOff').

- **Operand Formats and Instruction Formats:**

  These classes of types describe the possible interpretations of an instruction, i.e. the semantics of its various fields. The operation code field is always the same but the interpretation of the operand word (i.e. the rest of the instruction word) depends on this code. A definition of all operation codes can be found in Appendix B.

- **Contents of Trigger and Context Store;**

  'Trigger' and 'Context' define the interpretations of the words in the trigger and in the context store, respectively.

- **Function Descriptors:**

  'FunDescr' uniquely describes a function by pointing to a word in the program store that holds the base address of the denoted code block. 'CurryFun' is the description of a curried function by holding a field 'FunDescr' and a field 'Constant' that indicates the parameter number of the first lambda-lifted arguments (for details, see Section 5.6). 'ActivAddr' uniquely desribes a particular function activation in the network by context pointer and module address.

- **Heap References:**

  These types define the construction of heap references, i.e. references to tuples stored in some memory unit. Each pointer consists of a module address (indicating the address of the memory unit), a tuple address (indicating a particular tuple within the unit) and a reference weight (used for garbage collection). For more details, see Section A.4.

---

[2] Actually, it will be sensible to insert buffer queues between arbiter and output link in order to smooth different communication speeds and keep communication latencies as small as possible.

[3] In a concrete implementation, some communication protocol has to be introduced that organizes a correct "shutdown" of the network and inhibits deadlocks. Moreover, after the network has terminated, the arbiter will reset itself to its initial state in order to receive a new program to be executed

- **Argument Messages, Load Reports, Message Tokens:**

  The following classes of types define the structure of message tokens, i.e. tokens that transmit pieces of information between processor and/or memory units. Basically, there are two types of messages, argument messages and load reports. The former type again may be subdivided into value messages and context messages.

- **Request Tokens:**

  These types define the construction of request tokens, i.e. tokens that transmit call requests from one processor to another.

- **Atomic Data:**

  These types define the atomic data that are stored in registers and that all instructions work on. These values may be fixed-size numbers ('Integer'), references into memory units ('Pointer'), references to function activations ('ActivAddr') or descriptions of curried functions ('CurryFun'). Additionally, each value is augmented with a flag that indicates whether the value has to be considered as valid or not (error value).

- **Processor Action:**

  This type denotes the possible kinds of actions that the processor may be engaged in (see Section A.3.4).

- **Resource and Load Measures:**

  The following types define various measures that are used to compare the resource utilization of different processors and to estimate the influence that the creation of a new function activation will have on this utilization.

A.3.3 Global Variables

Figure A.7 declares the global variables that are referenced in the following algorithms and subalgorithms.

- **Basic Stores:**

  The variables '_Store' denote the various storage classes that the processor has access to. The program store 'progStore', the frame store 'frameStore' and the context store 'contStore' are arrays that can be directly referenced by the corresponding types of indizes. On the other hand, the trigger store 'trigStore', the message store 'messStore' and the request store 'requStore' are defined by the pseudo-constructor **collection of**. We use this terminology to leave the actual organization of each of these stores undefined. The algorithms uses the pseudo-function *Fetch* to get a value from one of these stores and the pseudo-procedure *Put into* to store a value into them[4]. However, the order in which values are returned by these stores is not defined.

- **Load Descriptors:**

  The following variables hold all values that characterize the utilization of the processor i.e. the number of words occupied in each storage. These values serve as the basis of all "stratetic" decisions of the processor, in particular which kind of action to perform next, to accept or to forward a foreign call request and to determine the processor to which a call request is sent.

[4]This instruction is only used in combination with the trigger store, as the processor unit does not write values into the message store and the request store.

```
var

    (* Basic Stores *)
    progStore:  array [0..ProgLen-1] of ProgWord;
    frameStore: array [0..FrameLen-1] of FrameWord;
    contStore:  array [0..ContLen-1] of ContWord;
    trigStore:  collection of TrigWord;
    messStore:  collection of MessWord;
    requStore:  collection of RequWord;

    (* Load Descriptors *)
    frameLoad: FrameLoad;
    contLoad:  ContLoad;
    trigLoad:  TrigLoad;
    requLoad:  RequLoad;
    messLoad:  MessLoad;

    (* Descriptors of Connected Modules *)
    procLoad: array LinkAddr of LoadDescr; (* Load of Neighbour Modules *)
    modAddr: array LinkAddr of ModAddr; (* Address of Neighbour Modules *)
    linkAddr: array LinkAddr of LinkAddr; (* 'Reverse' Link Address *)

    (* Termination Flag *)
    terminate: Boolean;
```

Figure A.7: Global Processor Variables

```
ExecutionUnit

    var
        action: Action; (* Actual Processor Action *)

    Load Program into Program Store;
    Initialize modAddr and linkAddr;

    procLoad[1..LinkNum] := ⟨0, 0⟩:ProcLoad;
    ⟨frameLoad, contLoad, trigLoad, requLoad, messLoad⟩ :=
        ⟨0:FrameLoad, 0:ContLoad, 0:TrigLoad, 0:RequLoad, 0:MessLoad⟩;

    terminate := false;
    repeat (* Interpreter Loop *)
        action := ProcAction(procLoad,
            frameLoad, contLoad, trigLoad, requLoad, messLoad);
        case action of
            execute: Instruction (* Execute Instruction *)
            receive: Reception (* Receive Message *)
            request: Request (* Fulfill Request *)
            report:  Report (* Report Processor Load *)
        endcase
    until terminate
```

Figure A.8: Execution Unit

- **Descriptors of Connected Modules:**

  The table 'procLoad' contains the knowledge that the processor has on the utilization of each of its neighbour processors. These data consist of the load of the neighbour's frame store and the load of its request store. Although this amount of knowledge is very limited and the processor therefore has a verry narrow view of the whole network, it should serve as a sufficient basis for balancing the load in the network.

  The table 'modAddr' holds the network addresses of all neighbour modules. The array 'linkAddr' contains the index of the entry into that the load of the actual module is to be written on each neighbour module: If 'linkAddr[i]' = j, then the neighbour processor with number 'modAddr[i]' will store the load of the actual module into the entry procLoad[j].

- **Termination Flag:**

  This flag signals that a stop request has been sent to the processor and that it therefore has to terminate execution. This stop request is sent by the master module to each module of the network indicating that the program has successfully delivered a result. The module therefore has to reset itself to its initial state in order to load a new program for execution.

A.3.4 The Execution Unit

At the start of the execution, the ADAM machine program is loaded into the program store of the module (i.e. each processor holds the same contents in its program store). Then the arrays 'modAddr' and 'linkAddr' are initialized to hold the addresses of the neighbour modules and to give its own load report some position in the 'procLoad' arrays of the neighbours. 'procLoad' holds the information that the processor has about the load of its neighbours. Its values are initally set to zero as well as the variables that hold the processor's own load measures. The termination flag is reset to false and the interpretation loop is entered.

Depending on the processor's load values, the subalgorithm *ProcAction* decides which kind of action is to be performed next. We let the strategy undefined that the processor uses for its decision, hence, this algorithm is not specified in more detail.

There are four possible kinds of actions the processor may perform (illustrated in Figure A.8):

1. **Execute Instruction Sequence:**

   In this case, the processor fetches a trigger from the trigger store, determines the task that is described by the trigger and starts the execution of the specified program path.

2. **Receive a Message:**

   If the processor performs this kind of task, it fetches a token from the message store and stores the value it carries into the denoted slot of some activation frame. Basically, there are two kinds of messages, *argument messages* for the communication between user-created function activations and *load reports* transmitting system-internal reports between processors and memories.

3. **Fulfill Requests:**

   If the processor decides to accept a function request, it creates a new activation of a specific function. Basically, this requires the allocation of a new activation frame in the frame store and the reservation of a a context descriptor in the context store. If this is not possible (due to lack of free space), the processor forwards the request to that processor among its neighbours that it thinks to be least loaded.

4. **Report Processor Load:**

   This action is performed, if the processor load has significantly changed so that the neighbour processors should be informed about the new situation. In this case, the processor sends *load report messages* to all of its neighbours modules and to the master module.

If the termination flag is set to true[5] the top level loop is left and the processor becomes ready for "meta-communication" (basically loading a new program). However, we do not describe these actions in detail.

A.3.5 Executing an Instruction Sequence

The execution of a sequence of instructions (described by the algorithm in Figure A.9) is initiated by an instruction trigger fetched from the trigger store. Since this store holds descriptions of all that tasks that may be executed without further preliminaries, the processor is free in its decision which trigger to take. Therefore we may leave the organization of the trigger store undefined.

Each trigger consists of two fields, the *context* field and the *offset* field. The context is a reference to some specific entry in the context store. This entry again consists of two components, a *frame base* and a *code base*. The frame base is a reference to the first register in the frame of some function activation, while the code base is a reference to the first instruction of the function's code block. These components are extracted and saved in some variables for further use. The trigger's instruction offset is added to the context's code base which yields the address of the instruction to be executed.

[5] As we will see in Section A.3.7, this is only the case if a stop request has been received.

Instruction

```
var
    context: ContPtr;
    codeBase, instruction: ProgPtr;
    offset: BlockOff;
    frameBase: FramePtr;
    opC: OpCode;
    opWd: OpWord;

⟨context, offset⟩ := Fetch(trigStore):Trigger;
trigLoad := trigLoad − 1;
⟨frameBase, codeBase⟩ := ContextStore[context]:Context;
instruction := codeBase + offset:ProgPtr;

break := false;
repeat
    ⟨opC, opWd⟩ := ProgramStore[instruction]:Operation;
    Execute Instruction opC
until break
```

Figure A.9: Instruction Execution

The program word referenced by this address is fetched and split into the components *operation code* and *operand word*. Depending on the operation code, a specific procedure for the particular instruction is called. These procedures (representing the processor‘s "micro-code") are described in Appendix B in detail.

After having executed the instruction, a break flag (initally set to true) is tested. If the test still yields true, the processor assumes that the instruction address has been updated with the address of the instruction's successor and continues the execution. Otherwise some synchronization event has caused an interruption of the instruction sequence and the loop is left.

A.3.6 Receiving a Message

Receiving a Message requires the following tasks (depicted in Figure A.10):

1. Fetching a message token from the message store,
2. Identifying the message type,
3. Extracting the value that the message carries and the address of its destination
4. Routing the message value to its destination with respect to the type of the message and its address.

In detail, the processor performs the following actions:

- The processor fetches a token from the message store together with the actual load of this store. If the load is zero, the store is empty and the contents of the token are undefined. Otherwise the token's type and value fields are extracted.

Reception

```
var
    type: TokenType;
    mData: Message;
    data: Data;
    rOff, offset: BlockOff;
    rMod: ModAddr;
    rCont, context: ContPtr;
    op: Register;
    link: LinkAddr;
    load: ProcLoad;

⟨⟨type, mData⟩, messLoad⟩ := Fetch(messStore):⟨ProcToken, messLoad⟩;

if messLoad > 0 then (* Message Received *)
    case type of

        mess: (* Message Destined for Operation *)
            ⟨context, offset, data⟩ := mData:OpArg;
            ⟨frameBase, codeBase⟩ := ContextStore[context]:Context;
            ⟨_, ⟨_, op, _⟩⟩ := progStore[codeBase+offset:ProgPtr]:OpReReIn;
            frameStore[frameBase+op:FramePtr] := data:FrameWord;
            Put ⟨context, offset⟩:TrigWord into trigStore;
            trigLoad := trigLoad+1:TrigLoad

        context: (* Message Carries Result Context *)
            ⟨rCont, rOff, rMod, context⟩ := mData:ResArg;
            ⟨frameBase, codeBase⟩ := contStore[context]:Context;
            frameStore[frameBase−1:FramePtr] := ⟨rCont, rIns⟩:FrameWord;
            frameStore[frameBase−2:FramePtr] := rMod:FrameWord;
            Put ⟨context, 0:BlockOff⟩:TrigWord into TriggerStore;

        stop: (* Stop Token Encountered *)
            Empty Request Store
            terminate := true

        pLoad: (* Message Carries Processor Load *)
            ⟨link, load⟩ := mData:LoadRep;
            procLoad[link] := load;

    endcase
endif
```

Figure A.10: Receiving a Message

- If the type field has the value mess, the message value is argument to an instruction within some function activation[6]. In this case, the data field of the message contains a pointer to the context defining the function activation and an offset value denoting a particular instruction of this activation. These pieces of information are used to fetch the indicated instruction and to determine the address of its right operand, i.e. the number of the a register within the actual frame. The message value is stored in the referenced register and a trigger for the instruction is put into the trigger store.

- If the type field has the value context, the message carries a result address to some function activation, i.e. the information to which instruction in which activation on which processor the function has to deliver the result value of its computation. This result address is stored in the activation frame of the referenced function and the first instruction in the associated code block is triggered.

- If the type field of the token is stop, the master has signalled all processors that the desired result value has been delivered and that execution is to be terminated. In this case, the request store is emptied i.e. all requests are fetched and discarded until the stop token has encountered there, too. Then the termination variable is set.

- If the message token's type field has the value pLoad, the message is not used for program communication between functions but for "meta-communication" between processors. In this case, the message carries the report about the load some neighbour processor, i.e. the load of its frame store and its request store. The actual processor stores these pieces of information in the variable 'procLoad' and continues execution.

A.3.7 Fulfilling a Request

The request queue carries the messages that require the processor to activate a new instance of a specific function, i.e. to allocate an activation frame of appropriate size and a context that describes the new activation. The behaviour of the processor when receiving a call request is illustrated in Figure A.11.

- First, the processor fetches a token from the request store together with the actual load of this store. If the load is zero, the store is empty and the contents of the token are undefined. Otherwise the type of the token, the description of the function to activate and the address of the instruction waiting for the request acknowledge are exctracted.

- If the request type is stop, the token does actually not carry a request for activating a function but for terminating the execution of the program. This "stop request" is sent by the master module to all processors in the network and signals that the program has successfully delivered a result. In this case, the message store is emptied by fetching and discarding all messages until the stop token is received there, too. Then the top-level loop is left by setting the termination flag.

- Otherwise, the token carries the description of a function to activate. This description consists of a pointer to the base address of the function's code block in the program store. The first word of the code block contains a description of the function's resource requirements, i.e. the number of slots needed for synchronization bits and registers, respectively. If a block of the appropriate size is free in the frame store and if there is at least one word left in the context store, the resources are allocated and the new context is initialized. If there is not enough memory left, the request message is forwarded to that module that the processor thinks to be least loaded.

[6]By convention, messages are always sent to the right (i.e. the second) input argument of an instruction.

Request

```
var
    type: ProcType;
    rData: Request;
    events: Event;
    regs: Register;
    frame: FramePtr;
    off: BlockOff;
    cont, context: ContPtr;
    mod, min: ModAddr;
    fun: FunDescr;
    code: ProgPtr;

⟨⟨type, rData⟩, requestLoad⟩ := Fetch(requStore):⟨ProcToken, RequLoad⟩;
if requestLoad > 0 then
    (* Call Request Received *)
    if type = stop then
        Empty Message Store
        terminate := true
    else
        (* Activate Function *)
        ⟨type, cont, off, mod, fun⟩ := rData:ResArg;
        code := progStore[fun:ProgPtr]:ProgPtr;
        ⟨events, regs⟩ := progStore[code-1:ProgPtr]: Resource;
        if AvailContext ∧ AvailFrame(events, regs) then
            (* Allocate Context and Frame *)
            ⟨context, contLoad⟩ := AllocContext;
            ⟨frame, frameLoad⟩ := AllocFrame(events, regs);
            contStore[context] := ⟨frame, code⟩:ContWord;
            if type = fCall
                (* Enter Result Context and Release Trigger *)
                frameStore[frame−2:FramePtr] := mod:FrameWord;
                frameStore[frame−1:FramePtr] :=
                    ⟨con, off+1:ProgPtr⟩:FrameWord;
                Put ⟨newContext, 0:BlockOff⟩:TrigWord into trigStore
            endif
            (* Return New Context *)
            inpProc := ⟨mod, mess,
                ⟨cont, off, ⟨⟨ActMod, newContext⟩, false⟩⟩⟩:NetLink
        else
            (* Forward Call Request to Minimal Loaded Processor *)
            min := MinLoadedProc(procLoad,
                trigLoad, frameLoad, contLoad, messLoad, requLoad);
            inpProc := ⟨min, type, rData⟩:NetLink
        endif
    endif
endif
```

Figure A.11: Function Calls

Report

```
var
    i: LinkAddr;

(* Report Processor Load to Master *)
inpProc := ⟨0:ModAddr, pLoad,
    ActMod, ⟨frameLoad:Integer, requLoad:Integer⟩⟩⟩:NetLink

(* Report Processor Load to Neighbour Modules *)
for i := 1 to LinkNum do
    inpProc := ⟨procAddr[i], pLoad,
        ⟨linkAddr[i], ⟨frameLoad:Integer, requLoad:Integer⟩⟩⟩:NetLink
endfor
```

Figure A.12: Load Reports

- The request token also holds the address of the operation that the description of the function activation is to be sent to. If the type field of the token is fCall, this address additionally denotes the instruction to that the result of the function activation is to be sent: it is in the same activation but with a instruction address incremented by one. In this case, the result address is computed and stored in the new activation frame. The first instruction of the function is put into the trigger store and the function by this already activated. However, if the type field contains tCall, the result address will be sent in a seperate "context message" and the function is not yet activated. Finally, a message token is sent to the caller transmitting the module address and the address of the context describing the new function activation.

A.3.8 Reporting the Load

If the processor load has significantly changed, this information is reported to the master and to all the modules that the actual module is connected to. For this purpose, the processor looks up its table 'procAddr' to determine the neighbours' module addresses and sends a load report message (carrying the actual load of the frame store and of the request store) to each of them. These pieces of information roughly characterize the current state of the processor's load and the state expected for the near future. They therefore represent a satisfying tool to balance the workload among the worker modules.

Figure A.12 shows the algorithm in detail.

A.4 The Memory Unit

The following sections define the operational behaviour of a memory unit by an algorithm that describes the activities of its memory manager.

A.4.1 Global Constants

const

ExpLen = *Number of Tokens in "Expensive" Request Store*;
CheapLen = *Number of Tokens in "Cheap" Request Store*;
StackLen = *Number of Tuple References in Deallocation Stack*

HeapWdLen = *Length of Heap Words in Bits*;
CheapWdLen = *Length of Cheap Store Words in Bits*;
ExpWdLen = *Length of Expensive Store Words in Bits*;

MaxRef = *Maximal Reference Count (= "Infinity")*;

Figure A.13: Global Memory Constants

Figure A.13 describes the global constants that the memary manager references (in addition to the constants defined in Section A.2 and Section A.3.1):

- 'ExpLen' and 'CheapLen' define the number of words that can be stored in the "cheap", respectively "expensive" request Store. 'StackLen' is the maximal depth of the deallocation stack used during the garbage collection process.
- 'HeapWdLen', 'CheapWdLen' and 'ExpWdLen' define the bit lengths of the words stored in the heap and the request stores, respectively.
- 'MaxRef' defines the maximal reference count that can be represented in each word of the heap store. Since a reference count value of 'MaxRef' cannot be increased any more, this constant represents "infinity" and will be never again decreased to zero.

A.4.2 Global Types

Figure A.14 defines the global types that will be used in the following algorithms (in addition to the types defined in Section A.2 and Section A.3.2):

- The first statements define the types of words stored in the heap ('HeapWord') and the request stores ('CheapWord' and 'ExpWord').
- 'MemInfo' denotes the contents of tokens sent to the memory unit. This type is subdivided into cheap requests ('CheapRequ') and expensive requests ('ExpRequ') both of which again are subdived into various subrequests.
- 'Cell' is the high-level interpretation of the storage word 'HeapWord'. It consists of a 'Tag' field and two 'Data' fields.
- The next types define various kind of load measures ('HeapLoad', 'ExpLoad' and 'CheapLoad') and the information that each memory unit has about the load of its neighbours ('MemLoad').
- 'StackPtr' denotes the values that the top of the deallocation stack may receive during the garbage collection process. 'MemAction' denotes the kind of actions the memory manager may perform, i.e. its "degrees of freedom".

```
type

    (* Storage Words *)
    HeapWord = Bits(HeapWdLen);
    CheapWord= Bits(CheapWdLen);
    ExpWord  = Bits(ExpWdLen);

    (* Contents of Memory Tokens *)
    MemInfo = ExpRequest | CheapRequest;

    (* Expensive Memory Requests *)
    ExpRequest = ExpType × ExpInfo;
    ExpType    = (alloc, read, submit, stop);
    ExpInfo    = Constant × ContPtr × Offset × ModAddr × TupleData;
    TupleData  = TupleRef × Boolean;

    (* Cheap Memory Requests *)
    CheapRequest = CheapType × CheapInfo;
    CheapType    = (writeInt, writePtr, incRef, decRef, copyRef, mLoad);
    CheapInfo    = WriteInfo | RefInfo | CopyInfo | LoadInfo;
    WriteInfo    = Constant × Data × TupleData;
    RefInfo      = ContPtr × Offset × ModAddr × TupleData;
    CopyInfo     = Constant × TupleRef × ModAddr × TupleRef;
    LoadInfo     = ModAddr × MemLoad;

    (* Heap Cells *)
    Cell      = Tag × Data × Data;
    Tag       = TagInfo | ModAddr;
    TagInfo   = Boolean × RefCnt × FieldInfo × FieldInfo;
    FieldInfo = Boolean × Boolean;

    (* Memory Unit Load Parameters *)
    MemLoad   = HeapLoad × ExpLoad;
    HeapLoad  = [0..HeapLen];
    ExpLoad   = [0..ExpLen];
    CheapLoad = [0..CheapLen];

    (* Miscellaneous *)
    StackPtr = [1..StackLen];
    MemAction = (exp, cheap, report);
```

Figure A.14: Global Memory Types

```
var

    (* Storages *)
    heap:       array [1..HeapLen] of HeapWord;
    expStore:   collection of ExpWord;
    cheapStore: collection of CheapWord;
    stack:      array [1..StackLen] of TupleRef;

    (* Load Parameters *)
    heapLoad:   HeapLoad;
    expLoad:    ExpLoad;
    cheapLoad:  CheapLoad;
    memLoad:    array [1..LinkMod] of Load;

    (* Network Parameters *)
    modAddr:array LinkAddr of ModAddr;
    linkAddr: array LinkAddr of LinkAddr;

    (* Miscellaneous *)
    free:   TupleAddr; (* List of Free Heap cells *)
    action: MemAction; (* Next Action to Execute *)
    top:    StackPtr; (* Top of Deallocation Stack *)
```

Figure A.15: Global Memory Variables

A.4.3 Global Variables

Figure A.15 shows the definition of the global variables referenced by the memory manager (in addition to that ones that have been defined in Section A.2).

- 'heap', 'expStore' and 'cheapStore' are the storages that the memory manager operates on. While the former one is directly referenced by addresses, access to the later ones is provided by the pseudo-instructions *Fetch* and *Put into*. As in Section A.3.3 we leave the concrete representation of these storages undefined. 'stack' is used for the recursive deallocation of tuples during the garbage collection process.
- Various load parameters '_Load' describe the utilization of each kind of storage, i.e. the number of words occupied. 'memLoad' holds the information that the memory manager has about the load of each neighbour memory.
- 'modAddr' holds the addresses of all neighbour modules, 'linkAddr' holds for each neighbour module the position on that the load of the actual module is stored in the neighbour's local table 'memLoad'.
- 'free' holds a pointer to a list of all unreferenced heap words, i.e. all that storage cells that may be delivered as results of allocation requests. 'action' denotes the kind of action the memory manager executes next. 'top' is the actual top of the deallocation stack used during the garbage collection process.

MemoryManager

```
var
    i: LinkAddr;
    action: MemAction;

Initialize linkMod and ownMod;

memLoad[1..LnkMod] := ⟨0, 0⟩: MemLoad;
⟨heapLoad, expLoad, cheapLoad⟩ := ⟨0:HeapLoad, 0:ExpLoad, 0:CheapLoad⟩;

free := 1:TupleAddr;
for i:=1 to HeapLen do
    heap[i] := ⟨_, _, (i+1):Value⟩: HeapWord;
endfor;
heap[HeapLen] := ⟨_, _, 0:Value⟩: HeapWord;

terminate := false;
repeat (* Manager Loop *)
    action := MemAction(heapLoad, expLoad, cheapLoad, memLoad);
    case action of
        exp:    ExpRequest;
        cheap:  CheapRequest;
        report: LoadReport
    endcase
until terminate;
```

Figure A.16: Memory Manager

A.4.4 The Memory Manager

Figure A.16 describes the behaviour of the memory manager:

- Having started its execution, the memory manager does a lot of initalization work: the tables 'linkMod' and 'ownMod' have to be defined in order to describe the memory manager's view of the actual network configuration. Furthermore, all load parameters have to be set to 0. Then the heap words have to be linked together in order to form the list 'free' of all unused words in the heap. The termination flag 'terminate' is set to false and the main loop is entered.
- Having entered the main loop, the memory manager uses the function *MemAction* to determine the kind of action to be executed next. Its decision basically depends on the load of the various storages and the load of its neighbour modules. Having decided to perform a particular kind of action, the manager calls one of the procedures 'CheapRequ', 'ExpRequ' or 'LoadReport'.
- The main loop is left, as soon as 'terminate' has been set to true. As we will see in Section A.4.6, this is only the case if a *stop request* has arrived.

A.4.5 Auxiliary Procedures

Figures A.17, A.18, A.19 and A.20 depict some auxiliary procedures that will be used by the algorithms presented in the following sections:

- **Availability of Heap Space:**
  Figure A.17 shows the subalgorithm 'AvailTuple' that checks whether enough heap cells are available for the allocation of a new tuple of particular size. For a tuple with 'n' fields, 'n'−1 heap heaps have to be allocated. The availability of these cells is determined by examining the value of 'heapLoad'.
- **Allocation of a Tuple:**
  Figure A.17 depicts the allocation of a tuple by the subalgorithm 'AllocTuple'. If a tuple with 'n' fields is to be allocated, 'n' has to be larger than 1, i.e. tuples that consist of a single field are not allowed (but also not necessary). The first 'n'−1 cells of the 'free' list are unlinked from this list and properly initalized:
  - The available-bits 'avHd' are set to false in order to signalize, that the head fields of the cell have not yet been written. 'avTl' of the 'n'-th cell is also set to false, since the corresponding tail field will contain the last field of the tuple. The data fields that correspond these bits are set to zero, since no instruction has requested their contents yet.
  - For the first 'n'−1 cells of the chain, 'open' is set to true, for the last cell, it is set to false.
  - The reference count of the first cell of the chain (that is also the reference count of the whole tuple) is set to MaxWght, which reflects the weight of the new cell reference created.
  - Finally, the 'free' list and the measure 'heapLoad' are updated.

```
AvailTuple(n: Constant): Boolean

    return (heapLoad + n:HeapLoad − 1:HeapLoad ≤ HeapLen:HeapLoad)

AllocTuple(n: Constant): TupleAddr

    var
        new: TupleAddr;
        tail: Data;

    if n ≤ 1 then ErrorHalt endif;
    new := free;
    ⟨_, _, tail⟩ := heap[new]: cell;

    while n > 2 do
        heap[free] :=
            ⟨⟨true, MaxWght, ⟨false, _⟩ ⟨false, _⟩⟩, 0:Data, tail⟩: HeapWord;
        free := tail:TupleAddr;
        n := n−1
    endwhile

    heap[free] := ⟨⟨false, _, ⟨false, _⟩ ⟨false, _⟩⟩, 0:Data, 0:Data⟩: HeapWord;
    free:= tail:TupleAddr;
    heapLoad := heapLoad + n:HeapLoad − 1:HeapLoad;

    return new
```

Figure A.17: Availability of Tuple Space and Its Allocation

GarbageCollection(tuple: TupleAddr, wght: RefWght)

var
    count, weight: RefCnt;

⟨top, stack[0]⟩ := ⟨1, ⟨tuple, wght⟩⟩;

while top > 0 **do**

    top := top − 1:StackPtr;
    ⟨tuple, wght⟩ := stack[top];
    weight := 2^{wght}: RefCount;
    ⟨⟨_, count, _, _⟩, _, _⟩ := heap[tuple]:Cell;

    if count < weight **then**
        ErrorHalt
    elsif count = MaxRef **then**
        NoOp
    elseif count = weight **then**
        DeallocTuple(tuple)
    else
        heap[tuple] := ⟨⟨_, count−weight, _, _⟩, _, _⟩;
    endif

endwhile

Figure A.18: The Garbage Collection Mechanism

DeallocTuple(tuple: TupleAddr)

```
var
    free0, addr: TupleAddr;
    open, open0, avHd, ptrHd, avTl, ptrTl, errHd, errTl: Boolean;
    head, tail: Info;
    mod: ModAddr;
    ref: TupleRef;
    wght: RefWght;

free0 := free; free := tuple;
⟨⟨open0, _, ⟨avHd, ptrHd⟩, ⟨avTl, ptrTl⟩⟩, ⟨head, errHd⟩, ⟨tail, errTl⟩⟩ := heap[cell]:cell;

open := true;
while open do
    if ¬avHd then ErrorHalt endif;
    if ptrHd & ¬errHd then
        ⟨mod, ref⟩ := head:Pointer;
        if (mod ≠ ActMod) then
            inpMem := ⟨mod, decRef, ⟨ref, errHd⟩⟩:NetLink
        else (* push reference on deallocation stack *)
            if top > MaxStack then ErrorHalt endif;
            stack[top] := ref;
            top := top + 1:StackPtr
        endif
    endif;
    heapLoad := heapLoad − 1:HeapLoad;
    open := open0;
    if open then tuple := tail;
    ⟨⟨open0, _, ⟨avHd, ptrHd⟩, ⟨avTl, ptrTl⟩⟩, ⟨head, errHd⟩, ⟨tail, errTl⟩⟩ := heap[cell]:cell;
endwhile;

if ¬AvTl then ErrorHalt endif;
if ptrTl & ¬errTl then
    ⟨mod, ref⟩ := tail:Pointer;
    if (mod ≠ ActMod) then
        inpMem := ⟨mod, decRef, ⟨ref, errTl⟩⟩:NetLink
    else (* push reference on deallocation stack *)
        stack[top] := ref;
        top := top + 1:StackPtr
    endif
endif;

heap[tuple] := ⟨_, _, free0:Data⟩:HeapWord
```

Figure A.19: Deallocation of a Tuple

Copy(tuple: Data, done: Boolean)

```
var
    addr: TupleAddr;
    wght: RefWght;
    error: Boolean;

⟨⟨_, ⟨addr, wght⟩⟩, error⟩ := data;
if data = 0:NodeAddr then (* nil pointer *)
    done := true
elsif error then (* error pointer *)
    done := true
elsif wght = 0:RefWght then (* minimum weight *)
    done := false
else
    done := true;
    data := ⟨⟨_, ⟨_, wght−1:RefWght⟩⟩, _⟩
endif
```

Figure A.20: Duplicating a Tuple Reference

- **The Garbage Collection Mechanism**

  Figure A.18 shows how the garbage collection mechanism is handled by the subalgorithm 'GarbageCollection'. If some tuple reference has been deleted, the reference count of the corresponding tuple has to be updated. If the count decreases to zero, the tuple may be reclaimed. In this case, all the tuple references stored in the tuple are deleted, the reference counts of the denoted tuples have to be updated and the garbage collection mechanism may recursively proceed.

  The procedure 'GarbageCollection' uses a global deallocation stack for recalling all those tuple references that have been deleted. Initially, the original reference is pushed on this stack. Iteratively, a reference is popped from the stack, its weight is extracted and the reference count of the denoted tuple is fetched. If the count is less than the weight, a severe program error has occurred and execution is terminated. If the count has already reached its maximum ("infinity"), the count is not changed any more i.e. the tuple will never be reclaimed. Otherwise, the reference count field is decreased by the weight of the deleted reference. If it becomes zero, the deallocation routine 'DeallocTuple' is called that reclaims the tuple cells and pushes all references stored in the tuple on the deallocation stack. The process proceeds until the deallocation stack is empty i.e. no more tuple references have been deleted.

- **Deallocation of a Tuple:**

  How the heap space of a tuple is reclaimed by subalgorithm 'DeallocTuple', is illustrated in Figure A.18:

  - The memory manager follows the 'tail' links of the tuple's cell chain and investigates the contents of the 'head' fields: If some 'avHd' bit is not yet set to true, some fatal error has occured, since in this case a write request may still be on the way to the tuple field. In this case, execution is terminated with an error message. The same is true for the 'tail' field of the last cell of the chain.

ExpRequest

```
var
    expRequ: ExpRequ;
    type: ExpType;
    info: ExpInfo;

⟨expLoad, expRequ⟩ := expStore:⟨ExpLoad, ExpRequest⟩;
if ExpLoad > 0:ExpLoad then
    ⟨type, info⟩ := expRequ:ExpRequest;
    case type of
        alloc: AllocRequest(info);
        read: ReadRequest(info);
        submit: SubRequest(info);
        stop: Empty cheap request store;
            terminate := true
    endcase
endif
```

Figure A.21: Expensive Memory Requests

- If some of these fields holds a pointer, the memory manager checks whether the referenced tuple is sited on the actual memory unit. If so, the reference is pushed on the deallocation stack such that the reference count of the denoted tuple can be updated. Otherwise, a decrease message is sent to the memory module with the referenced tuple in order to trigger the garbage collection there.

All cells of the tuple are linked to the 'free' list and 'heapLoad' is updated.

- **Duplicating a Tuple Reference**

  The subalgorithm 'Copy' depicted in Figure A.20 handles the duplication of tuple references. If a tuple reference has to be duplicated, both copies receive half of the weight of the original reference. However, if the reference is nil or error then the copy operation does not change the original pointer. If the reference weight has already reached its minimum ($1 = 2^0$), then the reference may not be copied any more and 'done' is set to false. Otherwise, the weight field of the reference is decreased by 1, which means to divide the weight by 2, since all weights are represented by their binary logarithms.

A.4.6 Expensive Requests

Figure A.21 shows the memory manager handling an "expensive" request:

- First, a request token 'expRequ' is fetched from the store 'expStore' together with the actual load 'expLoad' of the store. If 'expLoad' is zero, the store is empty and the contents of 'expRequ' are undefined.
- Otherwise, the 'type' field and the 'info' field of the token are extracted. Depending on the type of the request, one of the procedures 'AllocRequest', 'ReadRequest' or 'SubRequest' is called with the argument 'info'. If 'type' is stop, the token denotes a stop request. In this case, the cheap request store is emptied (by reading all requests until a stop token encounters there, too) and the flag 'terminate' is set.

```
AllocRequest(info: ExpInfo)

    var
        n: Constant;
        context: ContPtr;
        offset: BlockOff;
        mod, mem: ModAddr;
        tuple: TupleAddr;
        ptr: Pointer;

    ⟨n, context, offset, mod, _⟩ := info;
    if AvailTuple(n) then
        tuple := AllocTuple(n);
        ptr := ⟨⟨ActMod, ⟨tuple, MaxWght⟩⟩, false⟩:Pointer;
        inpMem := ⟨mod, mess, ⟨context, offset, ptr⟩⟩:NetLink;
    else
        mem := MinLoadedMem(heapLoad, expLoad, cheapLoad, memLoad);
        inpMem := ⟨mem, alloc, info⟩:NetLink;
    endif
```

Figure A.22: Allocation Requests

Allocation Requests

Figure A.22 shows the activities of the memory manager after having received an *allocation request*, i.e. the request to allocate space for a tuple of certain size:

- The 'info' field of the request is split into its components, where 'n' denotes the number of tuple fields required and 'context', 'offset' and 'mod' represent the network address of the instruction to that the result value is to be delivered.
- If there is enough memory left, the tuple is allocated and 'cell' is set to the address of this tuple. Then the actual tuple reference 'ptr' is constructed by combining 'cell' with the module address <u>ActMod</u>, the reference weight <u>MaxWght</u> and the error flag <u>false</u>. The reference is sent by a message token to the denoted instruction.
- If there is not enough memory left to fulfill the allocation request, the memory manager determines the memory unit it thinks to be loaded least among (the module itself and) its neighbours and forwards the request to that particular memory unit.

Read Requests

Figures A.23 and A.24 show the behaviour of the memory manager when handling a *read request*, i.e. the request to deliver the contents of some particular tuple field:

- First, the read request is split into its components, where 'n' is the number of the field to be read, 'context', 'offset' and 'mod' denote the address of the instruction to that the result is to be delivered, 'cell' represents the address of the tuple that holds the field, 'wght' is the weight of the reference and 'error' its error flag.

```
ReadRequest(info: ExpInfo)

    var
        n, num: Constant;
        context: ContPtr;
        offset: BlockOff;
        mod, tmod: ModAddr;
        tptr: TupleRef;
        tuple, tuple0, new: TupleAddr;
        wght: RefWght;
        error, open, avHd, avTl, ptrHd, ptrTl, gc: Boolean;
        refCnt: RefCnt;
        head, tail: Data;

    ⟨n, context, offset, mod, ⟨⟨tuple, wght⟩, error⟩⟩ := info;
    ⟨n, tuple0⟩ := ⟨num, tuple⟩;

    while open & n > 0:Constant do (* Find Desired Tuple Field *)
        ⟨⟨open, refCnt, ⟨avHd, ptrHd⟩, ⟨avTl, ptrTl⟩, head, tail⟩ := heap[tuple]:Cell;
        tuple := tail:TupleAddr;
        n := n − 1:Const
    endwhile;

    if n > 1 then ErrorHalt endif;
    gc := true;

    Continued in Figure A.24
```

Continued in Figure A.24

Figure A.23: Read Requests

Continued from Figure A.23

```
if n = 0 then (* Value Stored in Head Field *)

    if avHd & ¬ptrHd then (* Head Available and Not Pointer *)
        inpMem := ⟨mod, mess, ⟨context, offset, head⟩⟩:NetLink
    elsif avHd & ptrHd then (* Head Available and Pointer *)
        Copy(head, done);
        if done then (* Copy Head Pointer *)
            heap[tuple] := ⟨_, head, _⟩:HeapWord;
            inpMem := ⟨mod, mess, ⟨context, offset, head⟩⟩:NetLink;
            GarbageCollection(tuple0, wght)
        else (* Send Copy Request *)
            ⟨tmod, tptr⟩ := head;
            inpMem := ⟨tmod, copRef, ⟨num, ⟨⟨tuple0, wght⟩, error⟩, ActMod, tptr⟩;
            heap[tuple] := ⟨⟨_, _, ⟨false, _⟩, _⟩, 0:Data, _⟩:HeapWord;
            ⟨gc, avHd⟩ := ⟨false, false⟩
        endif
    endif
    if ¬avHd then (* Delay Read Request *)
        new := AllocTuple(1:Constant);
        heap[new] := ⟨mod, ⟨context, offset⟩:Data, head⟩:HeapWord;
        heap[tuple] := ⟨_, new:Data, _⟩:HeapWord;
        if gc then GarbageCollection(tuple0, wght) endif
    endif

else (* Value Stored in Tail Field *)

    if avTl & ¬ptrTl then (* Tail Available and Not Pointer *)
        inpMem := ⟨mod, mess, ⟨context, offset, tail⟩⟩:NetLink
    elsif avTl & ptrTL then (* Tail Available and Pointer *)
        Copy(tail, done);
        if done then (* Copy Tail Pointer *)
            heap[tuple] := ⟨_, _, tail⟩:HeapWord;
            inpMem := ⟨mod, mess, ⟨context, offset, tail⟩⟩:NetLink;
            GarbageCollection(tuple0, wght)
        else (* Send Copy Request *)
            ⟨tmod, tptr⟩ := tail;
            inpMem := ⟨tmod, copRef, ⟨num, ⟨⟨tuple0, wght⟩, error⟩, ActMod, tptr⟩;
            heap[tuple] := ⟨⟨_, _, _, ⟨false, _⟩⟩, _, 0:Data⟩:HeapWord;
            ⟨gc, avTl⟩ := ⟨false, false⟩
        endif
    endif
    if ¬avTl then (* Delay Read Request *)
        new := AllocTuple(1:Constant);
        heap[new] := ⟨mod, ⟨context, offset⟩:Data, head⟩:HeapWord;
        heap[tuple] := ⟨_, _, new:Data⟩:HeapWord;
        if gc then GarbageCollection(tuple0, wght) endif
    endif

endif
```

Figure A.24: Read Requests (Continued)

- The storage cell that holds the particular field of the tuple is searched by traversing n−1 times the chain of 'tail' pointers linking the cells that the tuple is constructed of.
- If the appropriate cell is found, the desired field may be the 'head' or the 'tail' field of the cell. The corresponding available flag ('avHd' respectively 'avTl') is tested. If it is false, the field does not yet hold the desired value. In this case, the field actually contains a pointer to a list of all those instructions that have up to now demanded this value. A storage cell is allocated to hold the return address of the read request. This cell is linked into the list of the deferred read requests. As soon as the desired value will become available, this and the other deferred requests will be fulfilled (see Section 4.5.2).
- If the tuple field already holds the desired value, the type flag of the field is checked. If the stored value is an integer atom, a message token is constructed and the integer is sent to the denoted instruction. Otherwise, the value is a pointer i.e. a reference to another tuple. In this case, it is duplicated by the function 'Copy' that decreases the weight of the reference. If 'Copy' succeeds, the reference is (a) written back into the tuple (with the weight decreased) and (b) delivered by a message token to the denoted instruction. However, if the reference weight is already $1 = 2^0$, 'Copy' has to fail. In this case, a copy request is sent to the memory unit that holds the tuple referenced by the pointer to be duplicated. The tuple field is marked as empty and the read request is queued. As soon as a new reference (with a new weight) is delivered, the read request can be fulfilled.
- Having processed the request, the reference contained in the request is deleted. Hence, the memory manager triggers the garbage collection process by calling the function 'Garbage-Collection' that decreases the reference count of the actual tuple and reclaims its cells if the tuple is not referenced any more. However, if a copy request was sent to another memory module, the reference was not deleted but forwarded to this module. In this case, the garbage collection process has to be suppressed (since the total reference count remains unchanged) by resetting the flag 'gc'.

Submission Requests

Figures A.25 and A.26 show the memory manager dealing with an *submission request*:

- First, the 'info' field of the the submission request is split into its components: 'context' and 'mod' denote the address of the activation to which the arguments of the higher-order function referenced by 'tuple' are to be submitted, 'wght' is the weight of the reference and 'error' its error flag. If 'error' is set to true or 'cell' holds the nil-pointer, a severe program error has occurred and execution is terminated.
- Otherwise, 'tuple' is the address of a tuple containing the description of a higher-order function: The first tuple field holds the pair of a function descriptor and a value 'n' denotes the parameter number of the first lifted function argument i.e. this argument is the 'n'-th argument of the function. All other tuple fields hold these lifted arguments in sequence. The heap word referenced by 'cell' is now split into its components. If the 'error' flag of the head component is set to true or the head is not yet available, a severe program error has occurred and execution is terminated.
- By traversing the chain of 'tail' links, the memory manager reads the contents of each lifted argument i.e. the 'head' field of each cell in the chain and the 'tail' field of the last cell. If the i-th tuple field contains an integer atom, a message token with this argument is constructed and sent to the denoted activation, i.e. to the ('n'+i)-th instruction of the activation. If it contains a tuple reference, the reference is duplicated in the same way as for read requests

```
SubRequest(info: ExpInfo)

    var
        context: Context;
        mod: ModAddr;
        cell, next, new: TupleAddr;
        wght: RefWght;
        error, open, avHd, avTl: Boolean;
        count: RefCnt;
        head, tail, head0, tail0: Data;
        fun: FunDescr;
        n: Constant;

    ⟨context, _, mod, ⟨⟨tuple, wght⟩, error⟩⟩ := info;
    if error ∨ (tuple = 0:TupleAddr) then ErrorHalt endif;
    ⟨⟨open, count⟨avHd, _⟩, _⟩, head0, tail0⟩ := heap[tuple]:Cell;
    if ¬avHd then ErrorHalt endif;
    ⟨head, tail⟩ := ⟨head0, tail0⟩;
    ⟨⟨_, n⟩, error⟩ := head:Data;
    if error then ErrorHalt endif;

    while open (* Submit Lambda-Lifted Arguments *)
        next := tail:TupleAddr;
        ⟨⟨open, _, ⟨avHd, avPtr⟩, ⟨avTl, avPtr⟩⟩, head, tail⟩ := tuple[next]:Cell;
        if avHd & ¬ptrHd then (* Head Available and Not Pointer *)
            inpMem := ⟨mod, mess, ⟨context, n:BlockOff, head⟩⟩:NetLink
        elsif avHd & ptrHd then (* Head Available and Pointer *)
            Copy(head, done);
            if done then (* Copy Head Pointer *)
                heap[tuple] := ⟨_, head, _⟩:HeapWord;
                inpMem := ⟨mod, mess, ⟨context, offset, head⟩⟩:NetLink
            else (* Send Copy Request *)
                ⟨tmod, tptr⟩ := head;
                inpMem := ⟨tmod, copRef,
                    ⟨n+1:Constant, ⟨⟨tuple0, wght⟩, error⟩, ActMod, tptr⟩;
                heap[tuple] := ⟨⟨_, _, ⟨false, _⟩, _⟩, 0:Data, _⟩:HeapWord
                count := count+1:RefCnt
            endif
        elsif ¬AvHd then (* Delay Read Request *)
            new := AllocTuple(1:Constant);
            heap[new] := ⟨mod, ⟨context, offset⟩:Data, head⟩:HeapWord;
            heap[tuple] := ⟨_, new:Data, _⟩:HeapWord
        endif
        n:= n+1:Constant;
    endwhile;
```

Continued in Figure A.26

Figure A.25: Submission Requests

Continued from Figure A.25

```
if avTl & ¬ptrTl then (* Tail Available and Not Pointer *)
    inpMem := ⟨mod, mess, ⟨context, n:BlockOff, tail⟩⟩:NetLink
elsif avTl & ptrTL then (* Tail Available and Pointer *)
    Copy(tail, done);
    if done then (* Copy Tail Pointer *)
        heap[tuple] := ⟨_, _, tail⟩:HeapWord;
        inpMem := ⟨mod, mess, ⟨context, n, tail⟩⟩:NetLink;
        GarbageCollection(tuple0, wght)
    else (* Send Copy Request *)
        ⟨tmod, tptr⟩ := tail;
        inpMem := ⟨tmod, copRef,
            ⟨n+1:Constant, ⟨⟨tuple0, wght⟩, error⟩, ActMod, tptr⟩;
        heap[tuple] := ⟨⟨_, _, _ ⟨false, _⟩⟩, _, 0:Data⟩:HeapWord;
        count := count+1:RefCnt;
    endif
elsif ¬AvTl then (* Delay Read Request *)
    new := AllocTuple(1:Constant);
    heap[new] := ⟨mod, ⟨context, offset⟩:Data, head⟩:HeapWord;
    heap[tuple] := ⟨_, _, new:Data⟩:HeapWord;
endif
heap[tuple0] := ⟨⟨_, count, _, _⟩, _, _⟩; GarbageCollection(tuple0, wght)
```

Figure A.26: Submission Requests (Continued)

CheapRequest

```
var
    type: CheapType;
    info: CheapInfo;

⟨type, info⟩ := Fetch(cheapStore):CheapRequest;
case type of
    wrInt, wrPtr:
        WriteRequest(type, info);
    incRef, decRef, copRef:
        RefRequest(type, info);
    mLoad:
        LoadMessage(info)
    stop:
        Empty Expensive Store;
        terminate := true
endcase
```

Figure A.27: Cheap Memory Requests

(i.e. a copy request may have to be submitted). If the head is not yet available, the manager allocates a storage cell, writes the address of the destination instruction into this cell and links it to the queue of deferred reads.

- At last, the reference contained in the submission request is deleted and the garbage collection process is triggered. However, before ‘GarbageCollection’ may be called, the reference count field of the tuple has to be updated by adding the weights of all the references to this tuple that were contained in copy requests sent to other memory units. Since, all of these references had weight 1, it now suffices to increase the reference count by the number of requests sent.

A.4.7 Cheap Requests

Figure A.27 describes the memory manager's behaviour when having received a “cheap” request. Depending on the request type (write requests, reference requests and load reports), one of the subalgorithms ‘WriteRequests’, ‘RefRequest’ or ‘LoadMessage’ is called. However, if a stop token has encountered, the expensive store is emptied and the termination flag is set.

WriteRequests

Figures A.28 and A.29 show the activities of the memory manager after having received a *write request*:

- First, the ‘info’ field of the request is split into its components ‘n’ (the field number of the tuple), ‘data’ (the value to be written into that field), ‘tuple’ (the address of the tuple), ‘wght’ (the weight of this reference) and ‘error’ (the error flag of the reference).
- If the error bit of the reference is set to true or the reference holds the nil-pointer, the write request is invalid, which signals a severe program error. In this case, execution is terminated with an error message.

WriteRequest(type: CheapType, info: WriteInfo)

```
var
    n, num: Constant;
    data, head, tail, nTail: Data;
    error, open, going, avHd, avTl, isAtom, isHd: Boolean;
    infHd, infTl: FieldInfo;
    tuple, tuple0, next: TupleAddr;
    wght: RefWght;
    refCnt: RefCnt;
    mod, tmod: ModuleAddr;
    tref: Pointer;
    context: ContPtr;
    offset: BlockOff;

⟨num, data, ⟨⟨tuple, wght⟩, error⟩⟩ := info;
if error | (tuple = 0:TupleAddr) then ErrorHalt endif;
⟨n, tuple0 ⟩ := ⟨num, tuple⟩;
⟨⟨open, _, ⟨avHd, ptrHd⟩, ⟨avTl, ptrTl⟩⟩, head, tail⟩ := heap[tuple0]:Cell;

while open & (n > 0:Constant) do (* Search Tuple Cell *)
    n := n − 1:Constant;
    tuple0 := tail:TupleAddr;
    ⟨⟨open, _, ⟨avHd, ptrHd⟩, ⟨avTl, ptrTl⟩⟩, head, tail⟩ := heap[tuple0]:Cell;
endwhile;

if n > 1:Constant then ErrorHalt endif;
⟨_, error⟩ := data;
⟨isHd, isAtom⟩ := ⟨(n=0)|open|error, type=wrInt | data = 0:Data⟩;

if isHd then (* Head Field of Cell *)
    if avHd then ErrorHalt endif;
    next := head:TupleAddr;
    heap[tuple0] := ⟨⟨_, _, ⟨true, ¬isAtom⟩, _⟩, _, _⟩
else (* Tail Field of Cell *)
    if avTl then ErrorHalt endif;
    next := tail:TupleAddr;
    heap[tuple0] := ⟨⟨_, _, _, ⟨true, ¬isAtom⟩⟩, _, _⟩
endif;
```

Continued in Figure A.29

Figure A.28: Write Requests

Continued from Figure A.28

```
going := true;
while going & (next ≠ 0:TupleAddr) do (* Fulfill Deferred Reads *)

    if isAtom then

        ⟨mod, ⟨context, offset⟩, nTail⟩ := heap[next]:Cell;
        inpMem := ⟨mod, mess, ⟨context, offset, data⟩⟩:NetLink;
        heap[next] := ⟨_, _, free:Data⟩:HeapWord;
        free := next;
        next := nTail:TupleAddr;
        heapLoad := heapLoad − 1;

    else

        Copy(data, done);

        if done then (* Pointer Could Be Copied *)
            if isHd
                then heap[tuple0] := ⟨_, data, _⟩
                else heap[tuple0] := ⟨_, _, data⟩
            endif inpMem := ⟨mod, mess, ⟨context, offset, data⟩⟩:NetLink;
            free := next;
            next := nTail:TupleAddr;
            heapLoad := heapLoad − 1;
        else (* Send Copy Request *)
            ⟨⟨tmod, tptr⟩, _⟨ := data;
            inpMem := ⟨tmod, copRef, ⟨num, ⟨⟨tuple, wght⟩, false⟩, ActMod, tptr⟩;
            if isHd
                then heap[tuple] := ⟨⟨_, _, ⟨false, _, _⟩⟩, next, _⟩:HeapWord;
                else heap[tuple] := ⟨⟨_, _, _ ⟨false, _⟩⟩, _, next⟩:HeapWord;
            endif;
            going := false
        endif

    endif

endwhile

if next = 0:NodeAddr then GarbageCollection(tuple, wght) endif
```

Figure A.29: Write Requests (Continued)

- Otherwise, the appropriate heap cell is searched by following the 'tail' links of the chain starting at the cell addressed by 'tuple'. If the chain holds too few tuple fields, execution is terminated with an error message.

- If the desired cell is found, the available bit of the appropriate field of the cell is investigated. If the field already holds a value, the "single assignment" rule of the heap is violated and execution is terminated with an error message. Otherwise, the available bit is set and the type of the value to be written is determined.

- In this case, the cell field holds a reference to a list of all those instructions to that the is to be delivered. If an integer atom is to be written (also the nil-pointer and the error value represent integer atoms), a message token holding the desired value is constructed and sent to all the denoted instructions. The heap cells used to store the requests are reclaimed into the free list.

- However, if a tuple reference has to be written, the weight of the reference has to be decreased by the routine 'Copy' each time before a message token is sent to one of the denoted instructions. If the weight decreases to its minimum, the rest of the deferred read requests cannot be fulfilled any more. They have to remain in the waiting queue and a copy reference message is sent to the module that holds the tuple referenced by the pointer to be duplicated. That module will send a reference with a new weight by another write message such that more deferred requests will be fulfilled.

- If no copy reference request had to be submitted (and hence no more waiting read requests are left), the garbage collection process is triggered. The tuple reference of the write request is deleted, the reference count of the tuple is updated and the tuple is reclaimed into the free list, if neccessary.

Reference Requests

Figure A.30 shows the behaviour of the memory manager after having received a *reference request*:

- If the request is an *increment request*, a new tuple reference with maximal weight has to be delivered to some instruction. The 'info' field of the request is split into the components 'context', 'offset' and 'mod' that denote the address of the instruction to that the new pointer is to be delivered and the reference 'tuple'.

  Since the weight of a 'tuple' reference in an increment request is always 1, the reference count is increased by $2^{\underline{\text{MaxWght}}}-1$, where $\underline{\text{MaxWght}}$ is the maximal weight of a reference. However, the reference count of the tuple has an upper bound of $\underline{\text{MaxRef}}$ that must not be exceeded. Anyway, 'tuple' is given the weight $\underline{\text{MaxWght}}$ and sent to the denoted instruction.

- If the request is an *decrement request*, a tuple reference is to be deleted. In this case, the reference 'tuple' and its weight 'wght' are extracted from the token. Then the routine 'GarbageCollection' is called that updates the reference count of the denoted tuple and reclaims its heap cells into the free list, if required.

- If the request is an *copy request*, a new tuple reference with maximal weight has to be delivered to some memory unit by a write message. The values 'field', 'ptr', and 'mod' representing the address of some particular tuple field and 'tuple' denoting the tuple reference to be copied are extracted.

  The reference count field of the addressed tuple is adjusted in the same way as for increment requests. However, the new reference with maximal weight is delivered by a write request token to the memory unit on that the tuple field (fixed by 'field', 'ptr' and 'mod') is sited into that the new reference is to be written.

RefRequest(type: CheapType, info: (RefInfo|CopyInfo))

```
var
    context: ContPtr;
    offset: BlockOff;
    mod: ModAddr;
    tuple: TupleAddr;
    wght: RefWght;
    refCnt: RefCnt;
    field: Constant;
    ptr: TupleRef;

if type = incRef then

    ⟨context, offset, mod, ⟨⟨tuple, _⟩, _⟩⟩ := info:RefInfo;
    ⟨⟨_, refCnt, _, _⟩, _, _⟩ := heap[tuple]:Cell;
    heap[tuple] :=
        ⟨⟨_, Min(MaxRef, refCnt+(2^MaxWght − 1):RefCnt), _, _⟩, _, _⟩:HeapWord;
    inpMem := ⟨mod, mess,
        ⟨context, offset, ⟨ActMod, ⟨tuple, MaxWght⟩⟩, error⟩⟩:NetLink;

elsif type = decRef then

    ⟨_, _, _, ⟨⟨tuple, wght⟩, _⟩⟩ := info:RefInfo;
    GarbageCollection(tuple, wght)

else (* type = copRef *)

    ⟨field, ptr, mod, ⟨⟨tuple, _⟩, _⟩⟩ := info:CopyInfo;
    ⟨⟨_, refCnt, _, _⟩, _, _⟩ := heap[tuple]:Cell;
    heap[tuple] :=
        ⟨⟨_, Min(MaxRef, refCnt+(2^MaxWght − 1):RefCnt), _, _⟩, _, _⟩:HeapWord;
    inpMem := ⟨mod, wrPtr, field
        ⟨⟨⟨ActMod, ⟨tuple, MaxWght-1:RefWght⟩⟩, false⟩, ptr⟩:NetLink;

endif
```

Figure A.30: Reference Requests

```
LoadMessage(info: LoadInfo)

    var
        pos: LinkAddr;
        load: MemLoad;

    ⟨pos, load⟩ := info:LoadInfo;
    memLoad[pos] := load;
```

Figure A.31: Load Messages

```
LoadReport

    var
        i: LinkAddr;

    (* Send Load Report to Master *)
    inpMem := ⟨0:ModAddr, mLoad, ⟨ActMod, ⟨heapLoad, expLoad⟩⟩⟩:NetLink

    (* Send Load Report to Neighbour Modules *) for i:=1 to LinkNum do
        inpMem := ⟨modAddr[i], mLoad, ⟨linkAddr[i], ⟨heapLoad, expLoad⟩⟩⟩:NetLink
    endfor
```

Figure A.32: Load Reports

Load Messages

Figure A.31 shows the activities of the memory manager when dealing with a *load message*:

The incoming message is split into its components 'pos' and 'load' where 'pos' denotes a position in the working load table 'memLoad' and 'load' is the datum to be stored at this position. 'pos' has such a value that the entry 'linkMod[pos]' holds the address of the neighbour module that has sent the load message. Using these pieces of information, 'memLoad' is updated. The new load description will influence future decisions of the manager, in particular to which memory units allocation requests may be favourably forwarded.

A.4.8 Load Reports

Figure A.32 shows the memory manager's behaviour after having decided to inform the neighbours about its actual load:

To each neighbour (and also to the master unit), a *memory load* message is sent holding the actual load of the module. Each of these messages carries the table position 'linkAddr[i]' at which this load description will be stored into the local array 'memLoad' of the neighbour module 'i' (for details, see Section A.4.3).

Appendix B

The ADAM Instruction Set

This chapter defines the operational semantics of the ADAM instruction set. This set consists of 49 instructions that may be subdivided into four major classes:

1. Arithmetical-Logical Instructions,
2. Control Instructions,
3. Function Call Instructions,
4. Storage Instructions.

Figure B.1 shows a list of these instructions, their names, their types (see Section A.3.2) and one-line descriptions. The parenthesized name extensions denote special forms of the instructions. However, due to lack of space only the principal types are presented in the list.

When the processor unit starts the execution of an instruction sequence, it fetches an instruction word from the program store, exctracts its operation code and operand word and performs some task according to the value of the operation code.

Each instruction is therefore defined by a subalgorithm that describes the task the processor performs using the operand word of this instruction. Each subalgorithm is presented in a scheme of the form

```
OPCODE Format
Description

    Algorithm
```

where `OPCODE` is the operation code of the instruction, `Format` denotes the interpretation of the operand word as defined in Section A.3.2 and *Description* gives a short hint on the purpose of the instruction.

Name	Type	Description
ADD(C)	OpReReIn	Integer Addition (with Constant)
SUB((R)C)	OpReReIn	Integer Subtraction ((Reverse) with Constant)
MUL(C)	OpReReIn	Integer Multiplication (with Constant)
DIV((R)C)	OpReReIn	Integer Division ((Reverse) with Constant)
MOD((R)C)	OpReReIn	Integer Remainder ((Reverse) with Constant)
AND	OpReReIn	Boolean Conjunction
OR	OpReReIn	Boolean Disjunction
EQUAL(C)	OpReReIn	Test for Integer Equality (with Constant)
LESS((R)C)	OpReReIn	Test for Integer Strict Order ((Reverse) with Constant)
LESSEQ((R)C)	OpReReIn	Test for Integer Reflexive Order ((Reverse) with Constant)
NEG	OpReReIn	Integer Negation
NOT	OpReReIn	Boolean Negation
FAILED	OpReReIn	Test for Error Value
COPYI	OpReReIn	Duplicate an Integer Value
COPYP	OpReReIn	Duplicate a Tuple Pointer
DELP	OpReIn	Delete a Pointer Value
SPLIT	OpInIn	Split Control Flow Into Parallel Paths
SWITCH	OpReInIn	Switch Control Flow
JUMP	OpJu	Route Control Flow
WAIT	OpEvIn	Wait for Synchronization Event
CONST	OpReIn	Generate Long Constant
FASTCALL	OpFuIn	Request Standard Function Activation
TAILCALL	OpFuIn	Request Tail-Recursive Function Activation
CONTEXT	OpReIn	Submit Actual Result Address to Callee
APPLY	OpReIn	Request Activation of a Non-Fixed Function
SEND(C)	OpReReCo	Send (Constant) Argument to a Function Activation
SUBMIT	OpReReIn	Request Submission of Curried Function Arguments
RETURN(C)	OpReIn	Send Result Value back to Caller
TERMINATE	Op	Terminate Actual Activation
ALLOC	OpCoIn	Allocate Tuple
READ	OpCoReIn	Read Contents of a Tuple Field
WRITEI	OpReReCo	Write Integer into a Tuple Field
WRITEP	OpReReCo	Write Pointer into a Tuple Field
WRITEC	OpCoReCo	Write Integer Constant into a Tuple Field

Figure B.1: The ADAM Instruction Set

B.1 Arithmetical-Logical Instructions

The *arithmetical-logical instructions* perform the "actual" computations, i.e. they construct new data from given ones. According to the number of operands each operation works on, they may be divided in three subclasses:

1. Binary Operations,
2. Binary Operations with one Constant Operand,
3. Unary Operations,

where the second class is a slight variant of the first one. In order to avoid duplications (and possible inconsistencies), we will describe these classes with the help of some standard functions that are given in the next section.

B.1.1 Standard Functions

The following statements define some mathematical functions with operands of type 'Integer' and results of type 'Data'. 'Integer' is that subrange of the infinite type 'Number' that can be represented in one word of the frame store. 'Data' is basically 'Integer' augmented with an *error flag* indicating whether the value it holds is valid or not (see Section A.3.2). Each function converts its operands into the infinite type 'Number' and performs the desired operation on that type. If the result can be represented in type 'Integer' it is converted to that form, otherwise the error flag is set.

For more details on this topic, see Section A.3.2.

add(left, right): Integer $\times$ Integer $\rightarrow$ Data
Integer Addition

⟨result, error⟩
  where
    sum:Number = left:Number + right:Number;
    error:Boolean = sum $\notin$ [MinInt..MaxInt];
    result:Integer =
      if error **then** left **else** sum:Integer;
  end

sub(left, right): Integer $\times$ Integer $\rightarrow$ Data
Integer Subtraction

⟨result, error⟩
  where
    diff:Number = left:Number − right:Number;
    error:Boolean = diff $\notin$ [MinInt..MaxInt];
    result:Integer =
      if error **then** left **else** diff:Integer;
  end

mul(left, right): Integer $\times$ Integer $\rightarrow$ Data
Integer Multiplication

⟨result, error⟩
where
prod:Number = left:Number $*$ right:Number;
error:Boolean = prod $\notin$ [MinInt..MaxInt];
result:Integer =
if error **then** left **else** prod:Integer;
end

div(left, right): Integer $\times$ Integer $\rightarrow$ Data
Integer Division

if right:Number = 0
then ⟨left, true⟩
else ⟨result, error⟩
where
quot:Number = left:Number / right:Number;
error:Boolean = quot $\notin$ [MinInt..MaxInt];
result:Integer =
if error **then** left **else** quot:Integer;
end

mod(left, right): Integer $\times$ Integer $\rightarrow$ Data
Integer Remainder Computation

if right:Number = 0
then ⟨left, true⟩
else ⟨result, false⟩
where
b:Number = left:Number;
a:Number = right:Number;
result:Number = a $-$ b $*$ (a/b)
end

and(left, right): Integer $\times$ Integer $\rightarrow$ Data
Boolean Conjunction

⟨left:Boolean $\wedge$ right:Boolean, false⟩

or(left, right): Integer $\times$ Integer $\rightarrow$ Data
Boolean Disjunction

$\langle$left:Boolean $\vee$ right:Boolean, <u>false</u>$\rangle$;

equal(left, right): Integer $\times$ Integer $\rightarrow$ Data
Test for Integer Equality

$\langle$left:Number $=$ right:Number, <u>false</u>$\rangle$

less(left, right): Integer $\times$ Integer $\rightarrow$ Data
Test for Integer Strict Order

$\langle$left:Number $<$ right:Number, <u>false</u>$\rangle$

lesseq(left, right): Integer $\times$ Integer $\rightarrow$ Data
Test for Integer Reflexive Order

$\langle$left:Number $\leq$ right:Number, <u>false</u>$\rangle$

B.1.2 Binary Instructions

The algorithms of all binary arithmetical-logical instructions are instances of the instruction scheme `BINAL` listed below. This scheme takes a function as argument that computes a result value of type 'Data' from two 'Integer' arguments.

`BINAL(f: Integer × Integer → Data) OpReReIn`
Binary Arithmetical-Logical Instructions

var
regL, regR: Register;
succ: InstOff;
opL, opR: FramePtr;
infL, infR: Info;
errL, errR: Boolean;
result: Data;

⟨regL, regR, succ⟩ := opWd:OpReReIn;
opL := frameBase + regL:FramePtr;
opR := frameBase + regR:FramePtr;
⟨infL, errL⟩ := frameStore[opL]:Data;
⟨infR, errR⟩ := frameStore[opR]:Data;
errL := errL ∨ errR;
if errL
then result := ⟨infL, true⟩
else result := f(infL:Integer, infR:Integer)
endif
frameStore[opL] := result:FrameWord;
instruction := instruction + succ:ProgPtr

The operand word of all binary arithmetical-logical instructions consists of the components 'regL', 'regR' and 'succ' where 'regL' and 'regR' denote register numbers within the current activation frame and 'succ' is the program offset to the successor instruction. The values of the denoted registers are loaded and their *error flags* are tested. If one of the operands is an error value, the result of the whole operation is also the error value, since all binary arithmetical-logical instructions demand their operands to be valid. Otherwise, the appropriate function is applied to the operands and the result value is stored into the first register. The instruction address is updated with the successor of the current instruction and a new cycle starts.

The following schemes define the semantics of all binary-arithmetical logical instructions by applying the `BINAL` to some of the functions defined in the previous section.

`ADD OpReReIn`
Integer Addition

BINAL(add)

`SUB OpReReIn`
Integer Subtraction

BINAL(sub)

`MUL` `OpReReIn`
Integer Multiplication

BINAL(mul)

`DIV` `OpReReIn`
Integer Division

BINAL(div)

`MOD` `OpReReIn`
Integer Remainder

BINAL(mod)

`AND` `OpReReIn`
Boolean Conjunction

BINAL(and)

`OR` `OpReReIn`
Boolean Disjunction

BINAL(or)

`EQUAL` `OpReReIn`
Test for Integer Equality

BINAL(equal)

`LESS` `OpReReIn`
Test for Integer Strict Order

BINAL(less)

`LESSEQ`	`OpReReIn`
Test for Integer Reflexive Order	

BINAL(lesseq)

B.1.3 Binary Instructions with Constants

If binary instructions have a small constant as one of their arguments, this argument may be directly encoded in the instruction itself, as described in the scheme `BINALC`.

`BINALC(f: Integer` $\times$ `Integer` $\rightarrow$ `Info)`	`OpCoReIn`
Binary Arithmetical Logical Instructions with a Constant	

var
  reg: Register;
  con: Constant;
  succ: InstOff;
  op: FramePtr;
  inf: Info;
  val: Value;
  err: Boolean;

$\langle$con, reg, succ$\rangle$:= opWd:OpReCoIn;
op := frameBase + reg:FramePtr;
$\langle$inf, err$\rangle$:= frameStore[op]:Data;
if err
  then result := $\langle$inf, $\underline{\text{true}}\rangle$;
  else result := f(inf:Integer, con:Integer)
endif
frameStore[op] := result:FrameWord;
instruction := instruction + succ:ProgPtr

This scheme is similar to the `BINAL` scheme of the previous section, except the fact that the second operand does not denote a register number but a constant value itself. This integer value needs not be tested for the error flag, since only proper integer constants may be encoded in an instruction.

The following schemes define the semantics of binary instructions with one constant argument by applying the `BINALC` scheme to some of the functions defined in Section B.1.1.

The instructions `SUBRC`, `DIVRC`, `MODRC`, `LESSRC`, `LESSEQRC` denote instructions, where the register and the constant have changed their input order in the case of non-commutative functions.

`ADDC`	`OpCoReIn`
Integer Addition of a Constant	

BINALC(add)

`SUBC` `OpCoReIn`
Integer Subtraction of a Constant

BINALC(sub)

`SUBRC` `OpCoReIn`
Reverse Integer Subtraction with a Constant

BINALC(subr)
  where
    subr(left, right): Integer $\times$ Integer $\rightarrow$ Data =
      sub(right, left)
  end

`MULC` `OpCoReIn`
Integer Multiplication with a Constant

BINALC(mul)

`DIVC` `OpCoReIn`
Integer Division by a Constant

BINALC(div)

`DIVRC` `OpCoReIn`
Reverse Integer Division by a Constant

BINALC(divr)
  where
    divr(left, right): Integer $\times$ Integer $\rightarrow$ Data =
      div(right, left)
  end

`MODC` `OpCoReIn`
Integer Remainder by a Constant

BINALC(mod)

`MODRC`	`OpCoReIn`
Reverse Integer Remainder with a Constant	

BINALC(modr)
  where
    modr(left, right): Integer $\times$ Integer $\rightarrow$ Data =
      mod(right, left)
    end

`EQUALC`	`OpCoReIn`
Integer Constant Test for Equality	

BINALC(equal)

`LESSC`	`OpCoReIn`
Integer Constant Test for Strict Order	

BINALC(less)

`LESSRC`	`OpCoReIn`
Reverse Integer Constant Test for Strict Order	

BINALC(lessr)
  where
    lessr(left, right): Integer $\times$ Integer $\rightarrow$ Data =
      less(right, left)
    end

`LESSEQC`	`OpCoReIn`
Integer Constant Test for Reflexive Order	

BINALC(lesseq)

`LESSEQRC`	`OpCoReIn`
Reverse Integer Constant Test for Reflexive Order	

BINALC(lesseqr)
  where
    lesseqr(left, right): Integer $\times$ Integer $\to$ Data =
      lesseq(right, left)
    end

B.1.4 Unary Instructions

The instruction scheme `UNAL` defines the behaviour of most of the unary arithmetical-logical instructions. Similar to the binary scheme, the right operand is fetched from the frame store, then it is tested for its error flag. If the error flag is set to true, the error flag of the result value is also set. Otherwise, a unary function is called and the result is stored into the frame store.

`UNAL(f: Integer → Info)`	`OpReReIn`
Scheme for Unary Arithmetical-Logical Instructions	

var
  regL, regR: Register;
  succ: InstOff;
  infL, infR: Info;
  errR: Boolean;
  result: Data;

$\langle$regL, regR, succ$\rangle$:= opWd:OpReReIn;
$\langle$infR, errR$\rangle$:= frameStore[frameBase+regR:FramePtr]:Data;
if errR
  then result := $\langle$infR, true$\rangle$
  else result := f(infR:Integer)
endif;
frameStore[frameBase+regL:FramePtr] := result:FrameWord;
instruction := instruction + succ:ProgPtr

`NEG`	`OpReReIn`
Integer Negation	

UNAL(neg)
  where
    neg(i): Integer $\to$ Data = $\langle$result, error$\rangle$
      where
        negi:Number = $-$ i:Number;
        error:Boolean = negi $\notin$ [MinInt..MaxInt];
        result:Integer = **if** error **then** i **else** negi;
      end
  end

`NOT` `OpReReIn`
Boolean Negation

UNAL(not)
    where
        not(i): Integer → Data = ⟨¬ i:Boolean, <u>false</u>⟩
    end

The instruction `FAILED` tests the errror flag of its operand, and delivers as its result a boolean value that says whether the value is the error value or not.

`FAILED` `OpReReIn`
Test for Error Value

var
    regL, regR: Register;
    infL, infR: Info;
    errR, delR: Boolean;
    mod: ModAddr;

⟨regL, regR, succ⟩ := opWd:OpReReIn;
⟨infR, errR⟩ := frameStore[frameBase+opR:FramePtr]:Data;
infL := (errR:Boolean = <u>true</u>);
frameStore[frameBase+opL:FramePtr] := ⟨infL, <u>false</u>⟩:FrameWord;
instruction := instruction + succ:ProgPtr

B.2 Control Instructions

Most of the *control instructions* route the control flow of the execution, some of them perform related tasks.

B.2.1 Split Control Flow

The `SPLIT` instruction is the only instruction with two successors. It forks an instruction sequence into two parallel program paths:

The first successor field of the operand word is used to form a trigger that is stored it into the trigger store. Thereby, this successor is declared as the begin of a parallel task which the processor may handle laterwards. The instruction sequence is continued with the instruction that is denoted by the second successor field of the operand word.

SPLIT	OpInIn
Split Control Flow Into Parallel Paths	

var
succ1, succ2: InstOff;

⟨succ1, succ2⟩ := opWd:OpInIn;
Put ⟨context,
instruction−codeBase+succ1:ProgPtr⟩:TrigWord *into* trigStore;
instruction := instruction + succ2:ProgPtr

B.2.2 Duplicate Atomic Values

The instructions `COPYI` and `COPYP` are used to duplicate atomic values (i.e. integers) and pointers (i.e. tuple references), respectively: The contents of the frame register denoted by the second operand are copied into the register denoted by the first one and execution is continued with the successor instruction.

COPYI	OpReReIn
Duplicate an Integer Value	

var
regL, regR: Register;
succ: InstOff;

⟨regL, regR, succ⟩ := opWd:OpReReIn;
frameStore[frameBase+regL:FramePtr] :=
frameStore[frameBase+regR:FramePtr]
instruction := instruction + succ:ProgPtr

In the case of the `COPYP` instruction, the task gets significantly more complicated:

Before a tuple reference may be duplicated, the processor has to look up its weight. If the weight is greater than $1 = 2^0$, the reference weight is divided by 2 (i.e. the binary logarithm stored in the weight field is decreased by 1) and the modified pointer is copied. If the field has already reached its minimum, an *increase request* is formed and sent to the memory unit that the referenced tuple holds. The instruction sequence has then to be interrupted. It is continued, as soon as the memory unit sends a message token to the interrupted `COPYP` instruction. This message delivers a new pointer that references the same tuple as before but has a weigth greater than zero and may therefore be duplicated without further trouble.

For more details on the weighted reference counting algorithm for garbage collection, see Section 4.5.4.

`COPYP`	`OpReReIn`
Duplicate a Tuple Pointer	

var
  regL, regR: Register;
  succ: InstOff;
  ref: TupleRef;
  weight: RefWght;
  del, err: Boolean;
  mod: ModAddr;
  tuple: TupleAddr;

⟨regL, regR, succ⟩ := opWd:OpReReIn;
⟨⟨mod, ref⟩, err⟩ :=
  frameStore[frameBase+regR:FramePtr]:Data;
⟨tuple, weight⟩ := ref:TupleRef;
if err | tuple = 0:TupleAddr **then** weight := MaxWght:RefWght **endif**;
if weight:Number > 0 **then**
  frameStore[frameBase+regL:FramePtr] :=
    ⟨⟨mod, ⟨tuple, weight−1:Weight⟩⟩, err⟩:FrameWord;
  frameStore[frameBase+regR:FramePtr] :=
    ⟨⟨mod, ⟨tuple, weight−1:Weight⟩⟩, err⟩:FrameWord;
  instruction := instruction + succ:ProgPtr
else
  inpProc := ⟨mod, incRef,
    ⟨context, offset, ActMod, ⟨ref, err⟩⟩⟩:NetLink;
  break := true
endif

B.2.3 Delete Pointer Values

For correct garbage collection, each memory unit has to be informed if a reference to one of the tuples it holds is deleted, i.e. if the pointer value in a frame register is no more used. The instruction `DELP` looks up the contents of the denoted register and sends a *decrement message* to that memory unit that holds the referenced tuple. Having submitted this message, the pointer may be deleted, i.e. the register may be overwritten. For more details on the garbage collection mechanism, see Section 4.5.4.

`DELP`	`OpReIn`
Delete Pointer Value	

var
  reg: Register;
  succ: InstOff;
  mod: ModAddr;
  ref: TupleRef;
  err: Boolean;

$\langle$reg, succ$\rangle$:= opWd:OpReIn;
$\langle\langle$mod, ref$\rangle$, err$\rangle$:= frameStore[frameBase+reg:FramePtr]:Data;
if $\neg$err & (ref:Number $\neq 0$) **then**
  inpProc := $\langle$mod, <u>decRef</u>, $\langle$ref, err$\rangle\rangle$:NetLink;
endif;
instruction := instruction + succ:CodePtr

B.2.4 Route Control Flow

The `SWITCH` instruction is the only instruction whose successor is not fixed but depends on the value of its register operand. The operand field of this instruction consists of three parts, one register number and two instruction offsets.

- If the specified register contains the boolean value <u>true</u>, the control flow is routed to the instruction referenced by the first instruction offset, i.e. the instruction counter is increased by this offset.
- If the register contains the value <u>false</u>, the control flow is routed to the instruction denoted by the second instruction offset.
- If the register contains an error value, the instruction sequence continues with that instruction that is immediately next in the program store.

`SWITCH`	`OpReInIn`
Switch Control Flow	

var
  reg: Register;
  succ1, succ2: InstOff;
  err: Boolean;

$\langle$reg, succ1, succ2$\rangle$:= opWd:OpReInIn;
$\langle$info, err$\rangle$:= frameStore[frameBase+reg:FramePtr];
if err **then**
  instruction := instruction + 1:ProgPtr
else if info:Boolean = <u>true</u> **then**
  instruction := instruction + succ1:ProgPtr
else
  instruction := instruction + succ2:ProgPtr
endif

Almost all instructions have the addresses of their successors directly encoded in their operand words as offsets to the actual instruction addresses. However, the maximum instruction offset that may be represented in this way may be to small to express the desired jump. If the compiler is not able to avoid such a "long jump" by code shifts or if "backward jumps" are necessary, the instruction `JUMP` may be used. The operand field of this instruction just contains a large *signed* instruction offset. The program counter is updated by this offset and thus a jump to the desired instruction is performed.

`JUMP`	`OpJu`
Route Control Flow	

instruction := instruction + opWd:ProgPtr

B.2.5 Wait for Synchronization Event

The `WAIT` instruction is the only tool for synchronization between parallel instruction paths. This instruction has two arguments, an event number and an instruction offset. The event number is a reference to a certain synchronization bit within the actual activation frame (see Section 4.4.2). The value of this synchronization bit is fetched and immediately inverted in the frame. If the bit has the value <u>true</u>, the synchronization event has already been triggered b a parallel task and the current instruction sequence may continue. Otherwise, the sequence has to be interrupted.

`WAIT`	`OpEvIn`
Wait for Synchronization Event	

var
  event: Event;
  succ: InstOff;
  slotNum, bitNum: Number;
  frameAddr: FramePtr;
  isSet: Boolean;

$\langle$event, succ$\rangle$:= opWd:OpEvIn;
slotNum := event:Number / FrameWordLen;
bitNum := event:Number − FrameWordLen$*$slotNum;
frameAddr := frameBase − (slotNum+3):FramePtr;
GetAndInvertBit(isSet, bitNum, frameAddr);
if isSet
  then instruction := instruction + succ:CodePtr;
  else break := <u>true</u>
endif

B.2.6 Deliver Long Constant

If binary instructions have a small constant as one of their arguments, this constant may be encoded directly in the instruction itself (see Section B.1.3). However, large constants as well as error constants have to be explicitely generated by the `CONST` instruction. This instruction assumes, that the next word in the program store does not contain an instruction but a numeric constant (flagged with an error bit). This constant is read and stored into a specified register and the instruction sequence is continued.

`CONST`	`OpReIn`
Generate Large Constant	

var
reg: Register;
succ: InstOff;

$\langle$reg, succ$\rangle$:= opWd:OpReIn;
frameStore[FrameBase+reg:FramePtr] :=
progStore[instruction+1:ProgPtr]:FrameWord;
instruction := instruction + succ:ProgPtr

B.3 Function Call Instructions

The following instruction are used to handle function calls, in particular to submit call requests and to submit function arguments.

B.3.1 Call Requests

In the ADAM architecture, there are two kinds of call requests, *fast calls* and *tail calls*. Both of them submit call request tokens into the network containing

1. the description of the function to activate and
2. the address of the instruction to that the description of the new function activation is to be sent.

This token is sent to that processor unit among the neighbour modules that the actual processor thinks to be least loaded. Then the instruction sequence is interrupted. As soon as the description of the new function activation has arrived, this is continued and the arguments may be sent to the activation.

The essential distinction between fast calls and tail calls is based on the fact how the address of the instruction is transmitted that shall receive the result of the activation:

Fast Calls are used, if the address of the instruction denoted by the call request (that will receive the activation description) also implies the address of the instruction that the activation's result is to be sent to. In detail, the "result instruction" has to be sited in the program store immediately after the "activation instruction".

Tail-Recursive Calls are used, if the result address is submitted in a seperate *context message*. This is especially useful, if the result value of a function is not submitted to the caller of the function but to another function activation. This feature has been introduced in order to optimize tail recursive function calls.

For more details on how to generate code for function calls, see Chapter 5.

Fast calls are generated by the instruction `FASTCALL`.

`FASTCALL`	`OpFuIn`
Request Standard Function Activation	

var
  fun: FunDescr;
  mod: ModAddr;
  succ: InstOff;

$\langle$fun, succ$\rangle$:= opWd:OpFuIn;
mod := MinLoaded(procLoad
  triggerLoad, frameLoad, tableLoad, messageLoad, requestLoad);
inpProc := $\langle$mod, fCall,
    $\langle$context, instruction-codeBase+succ:CodePtr, ActMod,
    fun$\rangle\rangle$:NetLink;
break := true

Tail-recursive calls are generated by the instruction `TAILCALL`.

`TAILCALL`	`OpFuIn`
Request Tail-Recursive Function Activation	

var
  fun: FunDescr;
  mod: ModAddr;
  succ: InstOff;

$\langle$fun, succ$\rangle$:= opWd:OpFuIn;
mod := MinLoaded(procLoad
  triggerLoad, frameLoad, tableLoad, messageLoad, requestLoad);
inpProc := $\langle$mod, tCall,
    $\langle$context, instruction-codeBase+succ:CodePtr, ActMod,
    fun$\rangle\rangle$:NetLink;
break := true

The `CONTEXT` instruction submits the result address stored in the frame of the current function activation to a newly created activation. Together with the instruction `TAILCALL`, this feature can be used to optimize tail recursive calls in following way: Each instance in a sequence of tail recursive calls uses `TAILCALL` to forward the result address of its caller, then submits the arguments to the next instance and finally terminates. The last instance in the chain will return its result directly to the caller of the first instance without passing all intermediate instances.

`CONTEXT`	`OpReIn`
Submit Actual Result Address to Callee	

var
reg: Register;
succ: InstOff;
mod, resMod: ModAddr;
cont: ContextPtr;
resAddr: Trigger;

$\langle$reg, succ$\rangle$:= opWd:OpReIn;
$\langle$mod, cont$\rangle$:= frameStore[frameBase+reg:FramePtr]:ActivAddr;
resMod := frameStore[frameBase-2]:ModAddr;
resAddr := frameStore[frameBase-1]:Trigger;
inpProc := $\langle$mod, <u>cMess</u>, $\langle$resAddr, resMod, cont$\rangle\rangle$:NetLink;
instruction := instruction + succ:CodePtr;

The `APPLY` instruction performs the same task as `FASTCALL` for variable function arguments. Its first argument denotes a register that contains the description of a function. This function descriptor is fetched and a call request is submitted. Since the function descriptor is not fixed, it may also be the error value. In this case, the request message is cancelled and error values are sent to the instructions that wait for the context of the function activation and the result value of the function call, respectively.

`APPLY`	`OpReIn`
Request Activation of a Non-Fixed Function	

```
var
    reg, reg1, reg2: Register;
    succ: InstOff;
    fun: FunDescr;
    mod: ModAddr;

⟨reg, succ⟩ := opWd:OpReIn;
⟨⟨fun, _⟩, err⟩ := frameStore[frameBase+reg:FramePtr]:Data;
if err then
    ⟨_, ⟨_, reg1, _⟩⟩ :=
        progStore[instruction+succ:ProgPtr]:OpReReIn;
    ⟨_, ⟨_, reg2, _⟩⟩ :=
        progStore[instruction+succ:ProgPtr+1:ProgPtr]:OpReReIn;
    frameStore[frameBase+reg1:FramePtr]:=
        ⟨_, true⟩:FrameWord;
    frameStore[frameBase+reg2:FramePtr] :=
        ⟨_, true⟩:FrameWord;
else
    mod := MinLoaded(procLoad,
        triggerLoad, frameLoad, tableLoad, messageLoad, requestLoad);
    inpProc := ⟨mod, fCall,
        ⟨context, instruction-codeBase+succ:CodePtr, ActMod,
        fun⟩⟩:NetLink;
endif
break := true
```

B.3.2 Sending of Arguments and Result Values

The `SEND` instruction has two registers and a small constant as its arguments. The first register contains the module address and context number of a function activation, the second one the argument to submit. The constant denotes the parameter number of the argument, i.e. the offset address of the instruction that the argument is to be sent to. `SEND` constructs a message token and sends the argument to the appropriate instruction in the specified function activation. The instruction sequence is continued with the immediately next instruction in the program store.

However, if the description of the function activation is actually an error value, `SEND` cancels the transmission the argument and continues the instruction sequence.

`SEND`	`OpReReCo`
Send Argument to a Function Activation	

var
regL, regR: Register;
n: Constant;
arg: FrameWord;
mod: ModAddr;
context: ContextPtr;
err: Boolean;

$\langle$regL, regR, n$\rangle$:= opWd:OpReReCo;
arg := frameStore[FrameBase+regL:FramePtr];
$\langle\langle$mod, context$\rangle$, err$\rangle$:=
frameStore[FrameBase+regR:FramePtr]:ActivAddr;
if ¬err **then**
inpProc := $\langle$mod, <u>mess</u>, $\langle$context, n:BlockOff, arg$\rangle\rangle$:NetLink
endif;
instruction := instruction + 1:CodePtr

The instruction `SENDC` performs the same task as the `SEND` instruction for arguments that are small constant integer values.

`SENDC`	`OpCoReIn`
Send a Constant to a Function Activation	

var
reg: Register;
c, n: Constant; mod: ModAddr;
context: ContextPtr;
err: Boolean;

$\langle$c, reg, n$\rangle$:= opWd:OpCoReCo;
$\langle\langle$mod, context$\rangle$, err$\rangle$:=
frameStore[FrameBase+reg:FramePtr]:ActivAddr;
if ¬err **then**
inpProc :=
$\langle$mod, <u>mess</u>, $\langle$context, n:BlockOff, $\langle$c, <u>false</u>$\rangle\rangle\rangle$:NetLink
endif;
instruction := instruction + 1:CodePtr

The `SUBMIT` instruction triggers the transmission of the fixed arguments of a curried function that are together with the description of the function stored in a tuple on the heap. The instruction has two register arguments 'regL' and 'regR': 'regL' contains the address of the tuple that holds the curried function, while 'regR' denotes the context of some recently created activation of that function. If the error bits of both arguments are not set, `SUBMIT` constructs a *submission request* holding the new context and sends this request to the memory unit where the tuple is stored. The memory unit will send the lifted function arguments to the denoted activation.

`SUBMIT`	`OpReReIn`
Request Submission of Curried Function Arguments	

var
    regL, regR: Register;
    succ: InstOff;
    mod, cMod: ModAddr;
    ref: TupleRef;
    context: ContPtr;
    err, cErr: Boolean;

⟨regL, regR, succ⟩ := opWd:OpReReIn;
⟨⟨mod, ref⟩, err⟩ :=
    frameStore[frameBase+regL:FramePtr]:Pointer;
⟨⟨cMod, context⟩, cErr⟩ :=
    frameStore[frameBase+regR:FramePtr]:ActivAddr;
if ¬err ∧ ¬cErr **then**
    inpProc := ⟨mod, <u>submit</u>, ⟨context, cMod, ⟨ref, error⟩⟩⟩:NetLink;
endif
instruction := instruction + succ:ProgPtr;

The `RETURN` instruction is used to deliver the result value of a function activation. A value message is constructed that takes its value from the first operand register and is submitted to the instruction whose address is stored as result address in the current activation frame. Afterwards, the instruction sequence is continued as specified by the successor field of the operand word.

`RETURN`	`OpReIn`
Send Result Value back to Caller	

var
    reg: Register;
    succ: InstOff;
    value: FrameWord;
    retMod: ModAddr;
    retCon: ContPtr;
    retIns: ProgPtr;

⟨reg, succ⟩ := opWd:OpReIn;
value := frameStore[frameBase+reg:FramePtr];
retMod := frameStore[frameBase-2:FramePtr]:ModAddr;
⟨retCon, retIns⟩ := frameStore[frameBase-1:FramePtr]:Trigger;
inpProc := ⟨retMod, <u>mess</u>, ⟨retCon, retIns, value⟩⟩:NetLink;
instruction := instruction + succ:ProgPtr

`RETURNC` performs the same task as `RETURN` except for the fact that the result value is encoded as a small integer constant in the instruction's operand word.

`RETURNC`	`OpCoIn`
Send Constant Result Value back to Caller	

var
reg: Register;
succ: InstOff;
con: Constant;
retMod: ModAddr;
retCon: ContPtr;
retIns: ProgPtr;

$\langle$con, succ$\rangle$:= opWd:OpCoIn;
retMod := frameStore[frameBase-2:FramePtr]:ModAddr;
$\langle$retCon, retIns$\rangle$:= frameStore[frameBase-1:FramePtr]:Trigger;
inpProc := $\langle$retMod, mess, $\langle$retCon, retIns, $\langle$con:Integer, false$\rangle\rangle\rangle$:NetLink;
instruction := instruction + succ:ProgPtr

B.3.3 Terminate Activation

The `TERMINATE` instruction is the only instruction that has no argument and no successor. Its single purpose is to deallocate the current activation frame and by this to terminate the actual function instance. When this instruction is executed, there must not exist any reference to this activation in the whole network. In particular, there must not exist any more triggers for this activation in the trigger store and there must not be any more messages on the way to this activation.

`TERMINATE`	`Op`
Terminate Actual Activation	

var
events, regs: Number;

$\langle$events, regs$\rangle$:= progStore[codeBase-1:ProgPtr]:Resource;
tableLoad := DeallocContext(context);
frameLoad := DeallocFrame(frameBase, events, regs); break := true

B.4 Storage Instructions

The following sections define all instructions that deal with tuples stored in the memory units of the worker modules. The operations performed by these instructions can be classified as follows:

1. Allocating tuples,
2. Reading the contents of tuple fields,
3. Writing the contents of tuple fields.

B.4.1 Allocating Tuples

The processor's task for the allocation of a tuple simply consists of sending an *allocation message* to some memory unit. For simplicity, allocation requests are always sent to the memory unit that resides on the same module as the actual processor unit. However, the memory unit is free to forward allocation requests to other memory units (e.g. if the required amount of storage is not available). After having sent the request, the instruction sequence is interrupted and continued as soon as the memory unit sends the address of the newly allocated tuple.

Each tuple may consist of an arbitrary but fixed number of atomic components (i.e. integer values or pointers). The number of the components required is encoded as a small constant within the operand part of the `ALLOC` instruction.

`ALLOC`	`OpCoIn`
Request Allocation of Tuple	

var
  n: Constant;
  succ: InstOff;

⟨n, succ⟩ := opWd:OpCoIn;
inpProc := ⟨ActMod, alloc,
  ⟨n, context, (instruction+succ-codeBase):BlockOff,
  ActMod⟩⟩:NetLink;
break := true

B.4.2 Reading the Contents of Tuple Fields

In order to receive the contents of some tuple field, the processor has to send a *read request* to the appropriate memory unit. The instruction `READ` performs this task.

A `READ` instruction has as arguments a register number, a small constant and an instruction offset. The specified register holds the description of a tuple (basically consisting of module address and tuple address). The small constant denotes the number of the tuple field to be read where 0 references the first field of the tuple.

Finally, the offset value defines the instruction to that the content of the tuple field are to be sent. Using these pieces of information, a read request token is formed and sent to the memory unit.

READ	OpCoReIn
Read Contents of a Tuple Field	

var
  n: Constant;
  reg, reg0: Register;
  succ: InstOff;
  inf: Info;
  err: Boolean;
  mod: ModAddr;
  tuple: TupleAddr;

⟨n, reg, succ⟩ := opWd:OpCoReIn;
⟨inf, err⟩ := frameStore[frameBase+reg:FramePtr]:Data;
⟨mod, tuple⟩ := inf:Pointer;
if err | tuple = 0:TupleAddr **then**
  ⟨_, ⟨_, reg0, _⟩⟩ := progStore[instruction+succ:CodePtr];
  frameStore[frameBase+reg0:FramePtr] := ⟨_, <u>true</u>⟩:FrameWord;
  instruction := instruction + succ:CodePtr
else
  Send ⟨mod, <u>read</u>,
    ⟨n, context, instruction+succ:CodePtr, <u>ActMod</u>,
    ⟨tuple, err⟩⟩⟩:NetLink;
  break := <u>true</u>
endif

B.4.3 Writing the Contents of Tuple Fields

In order to write the desired contents into some tuple field, the processor unit has to send a *write request* to the memory unit. A write request consists of the address of the specified memory unit, the address of the tuple in that unit, the number of the tuple field and the contents that shall be written into that field. In order to allow the correct collection of tuples that are not more used, the processor has to tell the memory unit whether the contents to be written are integer values or pointers that themselves reference other tuples.

Therefore, there exist two operations for writing the contents of a tuple field, `WRITEI` for integer values and `WRITEP` for pointers. A third instruction, `WRITEC`, is used for writing small integer constants that may be directly encoded into the instruction.

`WRITEI` and `WRITEP` have as arguments two registers and a small constant. The first register contains the datum to be written, while the second one holds the description of the tuple (i.e. module address and tuple address). The constant denotes the number of the tuple field to be written.

After having submitted the write request, program execution is continued with the instruction that is immediately next in the program store.

`WRITEI`	`OpReReCo`
Write Integer into a Tuple Field	

var
  regL, regR: Register;
  n: Constant;
  data: Word;
  mod: ModAddr;
  tuple: TupleAddr;
  err: Boolean;

⟨regL, regR, n⟩ := opWd:OpReReCo;
data := frameStore[frameBase+regL:FramePtr]:Data;
⟨⟨mod, tuple⟩, err⟩ := frameStore[frameBase+regR:FramePtr]:Data;
if err **then** *ErrorHalt* **endif**;
inpProc := ⟨mod, <u>wrInt</u>, ⟨n, ⟨tuple, err⟩, data⟩⟩:NetLink;
instruction := instruction + 1:CodePtr;

`WRITEP`	`OpReReCo`
Write Pointer into a Tuple Field	

var
  regL, regR: Register;
  n: Constant;
  data: Word;
  mod: ModAddr;
  tuple: TupleAddr;
  err: Boolean;

⟨regL, regR, n⟩ := opWd:OpReReCo;
data := frameStore[frameBase+regL:FramePtr]:Data;
⟨⟨mod, tuple⟩, err⟩ := frameStore[frameBase+regR:FramePtr]:AddrInfo;
if err **then** *ErrorHalt* **endif**;
inpProc ⟨mod, <u>wrPtr</u>, ⟨n, ⟨tuple, err⟩, data⟩⟩:NetLink;
instruction := instruction + 1:CodePtr;

The `WRITEC` instruction has the value to be written directly encoded in its operand word, which is only possible for small integer constants.

WRITEC	OpCoReCo
Write Integer Constant into a Tuple Field	

var
  reg: Register;
  n, c: Constant;
  data: Word;
  mod: ModAddr;
  tuple: TupleAddr;
  err: Boolean;

⟨c, reg, n⟩ := opWd:OpCoReCo; data := ⟨c:Integer, <u>false</u>⟩;
⟨⟨mod, tuple⟩, err⟩ := frameStore[frameBase+reg:FramePtr]:Data;
if err **then** *ErrorHalt* **endif**;
inpProc := ⟨mod, <u>wrInt</u>, ⟨n, ⟨tuple, err⟩, data⟩⟩:NetLink;
instruction := instruction + 1:ProgPtr;

Appendix C

The ADAM Source Code

A prototype of the ADAM interpreter has been implemented in OCCAM 2 on a network of INMOS T800 transputers. This appendix contains the full source code of this interpreter (but not of the ADAM assembly program). The source consists of of a sample network description (extension `.PGM`), the actual program files (extension `.OCC`) and several include files (extension `.INC`). This collection of files is organized into the following groups:

1. **Network Description**

   This file contains a sample network description for a chordal ring topology consisting of 8 worker modules and one master module.

ADAM Network Description		
File Name	*Description*	*Page*
`ADAM8.PGM`	ADAM Network Description	205

2. **General Include Files**

   These include files define several constants and protocols used in almost all procedures that the interpreter consists of.

General Include Files		
File Name	*Description*	*Page*
`ADERROR.INC`	Error Messages	212
`ADGLOBAL.INC`	Global Constants	216
`ADNET.INC`	Network Parameters	218

3. **The ADAM Master Module**

   This group consists of all source and include files used for the implementation of the ADAM master module.

The ADAM Master Module (Sources)		
File Name	*Description*	*Page*
`ADCONN.OCC`	Network Connection	220
`ADGENER.OCC`	Result Generator	225
`ADINOUT.OCC`	Input/Output Interface	247
`ADSTAT.OCC`	Statistics Generator	265
`ADMASTER.OCC`	Master Main Program	268

The ADAM Master Module (Includes)		
File Name	*Description*	*Page*
`ADSTAT.INC`	Statistics Constants	270
`ADMASTER.INC`	Master Module Parameters	271

4. **The ADAM Worker Module**

This group consists of all source and include files used for the implementation of the ADAM worker module.

The ADAM Worker Module (Sources)		
File Name	*Description*	*Page*
`ADARBIT.OCC`	Arbiter	273
`ADMEMORY.OCC`	Memory Unit	283
`ADPROC.OCC`	Processor Unit	322
`ADMODULE.OCC`	Worker Main Program	349

The ADAM Worker Module (Includes)		
File Name	*Description*	*Page*
`ADARBIT.INC`	Arbiter Parameters	351
`ADEXEC.INC`	Processor Execution Unit	352
`ADMEMORY.INC`	Memory Unit Parameters	396
`ADPROC.INC`	Processor Unit Parameters	398

```
-----------------------------------------------------------------------------
--
-- ADAM8.PGM                                              Wolfgang Schreiner
-- Adam Configuration                                     (c) RISC-Linz 1990
--
-- A Topology Description for a Network of 8 Worker Modules.
--
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- External References
-----------------------------------------------------------------------------

-- include files
#INCLUDE "adglobal.inc" -- global constants
#INCLUDE "adnet.inc"    -- network constants and protocols
#INCLUDE "hostio.inc"   -- server constants and protocols
#INCLUDE "linkaddr.inc" -- link address constants

-- compilation units
#USE "admaster.c8h" -- the adam master module
#USE "admodule.c8h" -- the adam network module

-----------------------------------------------------------------------------
-- Network Channel Definitions:
--
--   net.i.j is the channel from worker module i to worker module j,
--   0 denotes the master module.
-----------------------------------------------------------------------------

-- input, output channels from/to server
CHAN OF SP fs, ts:

-- the ring connection
CHAN OF Net.Link net.0.1:
CHAN OF Net.Link net.1.0:
CHAN OF Net.Link net.1.2:
CHAN OF Net.Link net.2.1:
CHAN OF Net.Link net.2.3:
CHAN OF Net.Link net.3.2:
CHAN OF Net.Link net.3.4:
CHAN OF Net.Link net.4.3:
CHAN OF Net.Link net.4.5:
CHAN OF Net.Link net.5.4:
CHAN OF Net.Link net.5.6:
CHAN OF Net.Link net.6.5:
CHAN OF Net.Link net.6.7:
CHAN OF Net.Link net.7.6:
CHAN OF Net.Link net.7.8:
CHAN OF Net.Link net.8.7:
CHAN OF Net.Link net.8.0:
CHAN OF Net.Link net.0.8:

-- first chordal connection
CHAN OF Net.Link net.8.2:
```

```
CHAN OF Net.Link net.2.8:
CHAN OF Net.Link net.2.4:
CHAN OF Net.Link net.4.2:
CHAN OF Net.Link net.4.6:
CHAN OF Net.Link net.6.4:
CHAN OF Net.Link net.6.8:
CHAN OF Net.Link net.8.6:

-- second chordal connection
CHAN OF Net.Link net.1.3:
CHAN OF Net.Link net.3.1:
CHAN OF Net.Link net.3.5:
CHAN OF Net.Link net.5.3:
CHAN OF Net.Link net.5.7:
CHAN OF Net.Link net.7.5:
CHAN OF Net.Link net.7.1:
CHAN OF Net.Link net.1.7:

-----------------------------------------------------------------------------
-- The Adam Topology:
--
--   8 worker modules are connected to a ring (with the master module being
--   linked between workers 8 and 1). Additionally, there are two rings
--   connecting modules 1, 3, 5, 7 and 2, 4, 6, 8 respectively.
--
--   For further explanation of the following procedure arguments, see the
--   source file 'ADMASTER.OCC' for the master module and the source file
--   'ADMODULE.OCC' for the worker modules.
--
-----------------------------------------------------------------------------
PLACED PAR

  ---------------------------------------------------------------------------
  -- master module on root transputer
  ---------------------------------------------------------------------------
  PROCESSOR 0 T800

    -- place channels on links
    PLACE fs      AT link0.in:
    PLACE ts      AT link0.out:
    PLACE net.1.0 AT link1.in:
    PLACE net.0.1 AT link1.out:
    PLACE net.8.0 AT link2.in:
    PLACE net.0.8 AT link2.out:

    -- the master module
    Adam.Master(net.1.0, net.0.8, net.8.0, net.0.1, fs, ts)

  ---------------------------------------------------------------------------
  -- worker module on transputer 1
  ---------------------------------------------------------------------------
  PROCESSOR 1 T800

    -- place channels
    PLACE net.3.1  AT link0.in:
```

```
  PLACE net.1.3  AT link0.out:
  PLACE net.0.1  AT link1.in:
  PLACE net.1.0  AT link1.out:
  PLACE net.7.1  AT link2.in:
  PLACE net.1.7  AT link2.out:
  PLACE net.2.1  AT link3.in:
  PLACE net.1.2  AT link3.out:

  -- dummy arrays of network channels
  [4]CHAN OF Net.Link in:  -- input from neighbours
  [4]CHAN OF Net.Link out: -- output to neighbours

  -- overlay channel arrays on links
  PLACE in  AT link0.in:
  PLACE out AT link0.out:

  -- network module 1
  Adam.Module(1(BYTE),
    [3(BYTE), 8(BYTE), 7(BYTE), 2(BYTE)],
    [2(BYTE), 3(BYTE), 0(BYTE), 1(BYTE)],
    [1, 255, 3, 0, 3, 0, 2, 2, 1], 1,
    [TRUE, FALSE, FALSE, TRUE], in, out)

---------------------------------------------------------------------------
-- worker module on transputer 2
---------------------------------------------------------------------------
PROCESSOR 2 T800

  -- place channels
  PLACE net.4.2  AT link0.in:
  PLACE net.2.4  AT link0.out:
  PLACE net.1.2  AT link1.in:
  PLACE net.2.1  AT link1.out:
  PLACE net.8.2  AT link2.in:
  PLACE net.2.8  AT link2.out:
  PLACE net.3.2  AT link3.in:
  PLACE net.2.3  AT link3.out:

  -- dummy arrays of network channels
  [4]CHAN OF Net.Link in:  -- input from neighbours
  [4]CHAN OF Net.Link out: -- output to neighbours

  -- overlay channel arrays on links
  PLACE in  AT link0.in:
  PLACE out AT link0.out:

  -- network module 2
  Adam.Module(2(BYTE),
    [4(BYTE), 1(BYTE), 8(BYTE), 3(BYTE)],
    [2(BYTE), 3(BYTE), 0(BYTE), 1(BYTE)],
    [1, 1, 255, 3, 0, 3, 0, 2, 2], 1,
    [FALSE, FALSE, FALSE, FALSE], in, out)

---------------------------------------------------------------------------
-- worker module on transputer 3
```

```
---------------------------------------------------------------------------
PROCESSOR 3 T800

  -- place channels
  PLACE net.5.3  AT link0.in:
  PLACE net.3.5  AT link0.out:
  PLACE net.2.3  AT link1.in:
  PLACE net.3.2  AT link1.out:
  PLACE net.1.3  AT link2.in:
  PLACE net.3.1  AT link2.out:
  PLACE net.4.3  AT link3.in:
  PLACE net.3.4  AT link3.out:

  -- dummy arrays of network channels
  [4]CHAN OF Net.Link in:  -- input from neighbours
  [4]CHAN OF Net.Link out: -- output to neighbours

  -- overlay channel arrays on links
  PLACE in  AT link0.in:
  PLACE out AT link0.out:

  -- network module 3
  Adam.Module(3(BYTE),
    [5(BYTE), 2(BYTE), 1(BYTE), 4(BYTE)],
    [2(BYTE), 3(BYTE), 0(BYTE), 1(BYTE)],
    [2, 2, 1, 255, 3, 0, 3, 0, 1], 2,
    [TRUE, FALSE, FALSE, TRUE], in, out)

---------------------------------------------------------------------------
-- worker module on transputer 4
---------------------------------------------------------------------------
PROCESSOR 4 T800

  -- place channels
  PLACE net.6.4  AT link0.in:
  PLACE net.4.6  AT link0.out:
  PLACE net.3.4  AT link1.in:
  PLACE net.4.3  AT link1.out:
  PLACE net.2.4  AT link2.in:
  PLACE net.4.2  AT link2.out:
  PLACE net.5.4  AT link3.in:
  PLACE net.4.5  AT link3.out:

  -- dummy arrays of network channels
  [4]CHAN OF Net.Link in:  -- input from neighbours
  [4]CHAN OF Net.Link out: -- output to neighbours

  -- overlay channel arrays on links
  PLACE in  AT link0.in:
  PLACE out AT link0.out:

  -- network module 4
  Adam.Module(4(BYTE),
    [6(BYTE), 3(BYTE), 2(BYTE), 5(BYTE)],
    [2(BYTE), 3(BYTE), 0(BYTE), 1(BYTE)],
```

```
    [2, 1, 2, 1, 255, 3, 0, 3, 0], 1,
    [FALSE, FALSE, FALSE, FALSE], in, out)

------------------------------------------------------------------------
-- worker module on transputer 5
------------------------------------------------------------------------
PROCESSOR 5 T800

  -- place channels
  PLACE net.7.5  AT link0.in:
  PLACE net.5.7  AT link0.out:
  PLACE net.4.5  AT link1.in:
  PLACE net.5.4  AT link1.out:
  PLACE net.3.5  AT link2.in:
  PLACE net.5.3  AT link2.out:
  PLACE net.6.5  AT link3.in:
  PLACE net.5.6  AT link3.out:

  -- dummy arrays of network channels
  [4]CHAN OF Net.Link in:  -- input from neighbours
  [4]CHAN OF Net.Link out: -- output to neighbours

  -- overlay channel arrays on links
  PLACE in  AT link0.in:
  PLACE out AT link0.out:

  -- network module 5
  Adam.Module(5(BYTE),
    [7(BYTE), 4(BYTE), 3(BYTE), 6(BYTE)],
    [2(BYTE), 3(BYTE), 0(BYTE), 1(BYTE)],
    [0, 2, 1, 2, 1, 255, 3, 0, 3], 2,
    [FALSE, FALSE, FALSE, FALSE], in, out)

------------------------------------------------------------------------
-- worker module on transputer 6
------------------------------------------------------------------------
PROCESSOR 6 T800

  -- place channels
  PLACE net.8.6  AT link0.in:
  PLACE net.6.8  AT link0.out:
  PLACE net.5.6  AT link1.in:
  PLACE net.6.5  AT link1.out:
  PLACE net.4.6  AT link2.in:
  PLACE net.6.4  AT link2.out:
  PLACE net.7.6  AT link3.in:
  PLACE net.6.7  AT link3.out:

  -- dummy arrays of network channels
  [4]CHAN OF Net.Link in:  -- input from neighbours
  [4]CHAN OF Net.Link out: -- output to neighbours

  -- overlay channel arrays on links
  PLACE in  AT link0.in:
  PLACE out AT link0.out:
```

```

    -- network module 6
    Adam.Module(6(BYTE),
      [8(BYTE), 5(BYTE), 4(BYTE), 7(BYTE)],
      [2(BYTE), 3(BYTE), 0(BYTE), 1(BYTE)],
      [0, 3, 2, 1, 2, 1, 255, 3, 0], 0,
      [FALSE, FALSE, FALSE, FALSE], in, out)

  --------------------------------------------------------------------------
  -- worker module on transputer 7
  --------------------------------------------------------------------------
  PROCESSOR 7 T800

    -- place channels
    PLACE net.1.7  AT link0.in:
    PLACE net.7.1  AT link0.out:
    PLACE net.6.7  AT link1.in:
    PLACE net.7.6  AT link1.out:
    PLACE net.5.7  AT link2.in:
    PLACE net.7.5  AT link2.out:
    PLACE net.8.7  AT link3.in:
    PLACE net.7.8  AT link3.out:

    -- dummy arrays of network channels
    [4]CHAN OF Net.Link in:  -- input from neighbours
    [4]CHAN OF Net.Link out: -- output to neighbours

    -- overlay channel arrays on links
    PLACE in  AT link0.in:
    PLACE out AT link0.out:

    -- network module 7
    Adam.Module(7(BYTE),
      [1(BYTE), 6(BYTE), 5(BYTE), 8(BYTE)],
      [2(BYTE), 3(BYTE), 0(BYTE), 1(BYTE)],
      [0, 0, 3, 2, 1, 2, 1, 255, 3], 3,
      [FALSE, FALSE, FALSE, FALSE], in, out)

  --------------------------------------------------------------------------
  -- worker module on transputer 8
  --------------------------------------------------------------------------
  PROCESSOR 8 T800

    -- place channels
    PLACE net.2.8  AT link0.in:
    PLACE net.8.2  AT link0.out:
    PLACE net.7.8  AT link1.in:
    PLACE net.8.7  AT link1.out:
    PLACE net.6.8  AT link2.in:
    PLACE net.8.6  AT link2.out:
    PLACE net.0.8  AT link3.in:
    PLACE net.8.0  AT link3.out:

    -- dummy arrays of network channels
    [4]CHAN OF Net.Link in:  -- input from neighbours
```

```
    [4]CHAN OF Net.Link out: -- output to neighbours

    -- overlay channel arrays on links
    PLACE in  AT link0.in:
    PLACE out AT link0.out:

    -- network module 8
    Adam.Module(8(BYTE),
      [2(BYTE), 7(BYTE), 6(BYTE), 1(BYTE)],
      [2(BYTE), 3(BYTE), 0(BYTE), 1(BYTE)],
      [3, 3, 0, 0, 2, 1, 2, 1, 255], 3,
      [FALSE, TRUE, TRUE, FALSE], in, out)

-----------------------------------------------------------------------------
-- End of ADAM8.OCC
-----------------------------------------------------------------------------
```

ADERROR.INC Page 1

```
------------------------------------------------------------------------------
--
-- ADERROR.IN                                          Wolfgang Schreiner
-- Error Messages                                      (c) RISC-Linz 1990
--
-- Includes all error messages used in the Adam machine.
--
------------------------------------------------------------------------------

------------------------------------------------------------------------------
-- head of all Adam assertion errors
------------------------------------------------------------------------------

VAL []BYTE adam.assert IS "ADAM assertion failed: ":

------------------------------------------------------------------------------
-- master module error messages
------------------------------------------------------------------------------

-- file name too long
VAL []BYTE file.len   IS "Interface> File name too long.":

-- missing file names
VAL []BYTE prog.miss  IS "Interface> Program file name missing.":
VAL []BYTE expr.miss  IS "Interface> Expression file name missing.":

-- replacing file extensions
VAL []BYTE file.ext   IS "Interface> Error when replacing file extension.":

-- program file errors
VAL []BYTE prog.open  IS "Interface> Error when opening program file.":
VAL []BYTE prog.read  IS "Interface> Error when reading program file.":
VAL []BYTE prog.close IS "Interface> Error when closing program file.":

-- expression file errors
VAL []BYTE expr.open  IS "Interface> Error when opening expression file.":
VAL []BYTE expr.read  IS "Interface> Error when reading expression file.":
VAL []BYTE expr.close IS "Interface> Error when closing expression file.":

-- type file errors
VAL []BYTE type.open  IS "Interface> Error when opening type descriptor file.":
VAL []BYTE type.read  IS "Interface> Error when reading type descriptor file.":
VAL []BYTE type.close IS "Interface> Error when closing type descriptor file.":
VAL []BYTE type.err   IS "Interface> Error when compiling type description.":

-- output file errors
VAL []BYTE out.open   IS "Interface> Error when opening ouput file.":
VAL []BYTE out.write  IS "Interface> Error when writing to output file.":
VAL []BYTE out.close  IS "Interface> Error when closing output file.":

-- temporary file errors
VAL []BYTE tmp.open   IS "Interface> Error when opening temporary file.":
VAL []BYTE tmp.read   IS "Interface> Error when reading temporary file.":
VAL []BYTE tmp.write  IS "Interface> Error when writing to temporary file.":
VAL []BYTE tmp.close  IS "Interface> Error when closing temporary file.":
```

```
VAL []BYTE tmp.delete IS "Interface> Error when deleting temporary file.":

-- keyboard input errors
VAL []BYTE key.read   IS "Interface> Error when reading keyboard input.":

-- type descriptor compilation errors
VAL []BYTE type.label IS "Generator> Invalid label in type desription.":
VAL []BYTE type.redef IS "Generator> Type label redefined.":

-- invalid token type
VAL []BYTE token.type IS "Generator> Invalid token received.":

-- invalid union variant
VAL []BYTE union.err  IS "Generator> Invalid union variant received.":

-- no load token receved
VAL []BYTE load.exp   IS "Statistics> Load token expected.":

-- string concatentation failed
VAL []BYTE str.cat    IS "Statistics> String concatenation failed.":

-- append signal received at invalid time
VAL []BYTE app.err   IS  "Connector> Invalid append signal received.":

-- write operation failed
VAL []BYTE write.err IS  "Interface> Error when writing string to file.":

----------------------------------------------------------------------------
-- processor unit error messages
----------------------------------------------------------------------------

-- program store too small for program
VAL []BYTE prog.full  IS "Execution> Program Store Full.":

-- message buffer or request buffer is not a multiple of token size
VAL []BYTE buff.err   IS "Processor> Buffer Invalid Size.":

-- message buffer or request buffer full
VAL []BYTE buff.full  IS "Processor> Buffer Full.":

-- program constant 0 encountered in division or remainder computation
VAL []BYTE const.zero IS "Instruction> DIV/MOD Constant Zero.":

-- boolean value neither represents true nor false
VAL []BYTE true.false IS "Instruction> SWITCH flag neither TRUE nor FALSE.":

-- tuple reference error flag set in write instruction
VAL []BYTE ref.err    IS "Instruction> WRITE Tuple Reference Error.":

-- unknown instruction code
VAL []BYTE inst.err   IS "Instruction> Unknown Opcode.":

-- invalid request token type
VAL []BYTE requ.err   IS "Execution> Invalid Request Type.":
```

```
-- invalid put attempt, trigger store is full
VAL []BYTE trig.full  IS "Execution> Trigger Store Full.":

-- invalid fetch attempt, trigger store is empty
VAL []BYTE trig.empt  IS "Execution> Trigger Store Empty.":

-- invalid allocation attempt, context store is full
VAL []BYTE cont.full  IS "Execution> Context Store Full.":

-- size of requested allocation frame larger than Free.Max
VAL []BYTE frame.size IS "Execution> Frame Too Large.":

-- not all synchronization events occurred when deallocating activation frame
VAL []BYTE synch.err  IS "Execution> Synchronization Error.":

-----------------------------------------------------------------------------
-- memory unit error messages
-----------------------------------------------------------------------------

-- tuple reference nil or error bit set
VAL []BYTE ref.err    IS "Memory> Invalid tuple reference.":

-- field number out of range
VAL []BYTE field.err  IS "Memory> Invalid field number.":

-- attempty to write a tuple field twice
VAL []BYTE mult.write IS "Memory> Single assignment violated.":

-- reference weight larger than reference count
VAL []BYTE weight.err IS "Memory> Reference weight error.":

-- function descriptor not yet available
VAL []BYTE submit.err IS "Memory> Submisson error.":

-- error during the allocation of heap cells
VAL []BYTE  alloc.err IS "Memory> Allocation error.":

-- error during the deallocation of heap cells
VAL []BYTE  dealloc.err IS "Memory> Deallocation error.":

-- error during the deallocation of heap cells
VAL []BYTE  space.err IS "Memory> Insufficient heap.":

-----------------------------------------------------------------------------
-- arbiter error messages
-----------------------------------------------------------------------------

-- network map invalid, program must not be forwarded to boot link
VAL []BYTE boot.err   IS "Arbiter> Can*'t forward program to boot link.":

-- acknowledge received from wrong link
VAL []BYTE ack.err    IS "Arbiter> Acknowledge signal from invalid link.":

-- arbiter buffer full
VAL []BYTE arbit.full IS "Arbiter> Token buffer is full.":
```

```

-----------------------------------------------------------------------------
-- End of ADERROR.INC
-----------------------------------------------------------------------------
```

ADGLOBAL.INC Page 1

```
-----------------------------------------------------------------------------
--
-- ADGLOBAL.INC                                          Wolfgang Schreiner
-- Global Constants                                      (c) RISC-Linz 1990
--
-- All constants that are globally referenced in the Adam interpreter
--
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- network constants
-----------------------------------------------------------------------------

VAL INT Mod.Num  IS  8:      -- number of network modules (excl. master)
VAL INT Link.Num IS  4:      -- number of modules each module is connected to

VAL INT Cluster.Size IS 128: -- number of words per program cluster
VAL INT Token.Size IS 3:     -- number of words per token

VAL INT Max.Debug IS 40:     -- maximal length of debugging message

----------------------------------------------------------------------------
-- value ranges
----------------------------------------------------------------------------

-- integer ranges
VAL INT32 Max.Int     IS  1073741823(INT32): -- maximal integer value  2^30-1
VAL INT32 Min.Int.Abs IS  1073741824(INT32): -- min. int. value absolute 2^30
VAL INT32 Min.Int     IS -1073741824(INT32): -- minimal integer value -2^30

-- constant ranges
VAL INT Max.Con     IS  127(INT): -- maximal constant value 2^7-1
VAL INT Min.Con.Abs IS  128(INT): -- minimal constant value absolute 2^7
VAL INT Min.Con     IS -128(INT): -- minimal constant value absolute -2^7
VAL INT Range.Con   IS  256(INT): -- constant range 2^8

-----------------------------------------------------------------------------
-- error codes
-----------------------------------------------------------------------------

-- least significant bit 1, i.e. error code must always be odd!
VAL INT32 Error.Overflow IS 1(INT32): -- integer overflow
VAL INT32 Error.ZeroDiv  IS 3(INT32): -- zero division (remainder)
VAL INT32 Error.Read     IS 5(INT32): -- nil pointer dereferenced

----------------------------------------------------------------------------
-- table of integers with exactly one bit set (used for bit operations)
----------------------------------------------------------------------------

VAL [32]INT32 bits IS
  [#00000001(INT32), #00000002(INT32), #00000004(INT32), #00000008(INT32),
   #00000010(INT32), #00000020(INT32), #00000040(INT32), #00000080(INT32),
   #00000100(INT32), #00000200(INT32), #00000400(INT32), #00000800(INT32),
   #00001000(INT32), #00002000(INT32), #00004000(INT32), #00008000(INT32),
   #00010000(INT32), #00020000(INT32), #00040000(INT32), #00080000(INT32),
```

```
   #00100000(INT32), #00200000(INT32), #00400000(INT32), #00800000(INT32),
   #01000000(INT32), #02000000(INT32), #04000000(INT32), #08000000(INT32),
   #10000000(INT32), #20000000(INT32), #40000000(INT32), #80000000(INT32)]:

------------------------------------------------------------------------------
-- Transputer Constants
------------------------------------------------------------------------------

VAL INT Ticks IS 15625: -- transputer clock ticks per second

------------------------------------------------------------------------------
-- End of ADGLOBAL.INC
------------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------
--
-- ADNET.INC                                          Wolfgang Schreiner
-- AdaM Global Network Constants and Protocols        (c) RISC-Linz 1990
--
-- Defines all the constants and protocols that refer to the communication
-- between different ADAM modules and between different units within a
-- module.
--
----------------------------------------------------------------------------

----------------------------------------------------------------------------
-- protocols
----------------------------------------------------------------------------

-- network protocol (between modules)
PROTOCOL Net.Link
  CASE
    net.startprog                    -- new program signal
    net.appendprog                   -- append program signal
    net.program; [Cluster.Size]INT32 -- program cluster
    net.endprog                      -- end of program signal
    net.ack                          -- acknowledgement signal
    net.token;   [Token.Size]INT32   -- network token
:

-- unit protocol (within modules)
PROTOCOL Mod.Link
  CASE
    mod.startprog                    -- new program signal
    mod.appendprog                   -- append program signal
    mod.program; [Cluster.Size]INT32 -- program cluster
    mod.endprog                      -- end of program signal
    mod.ack                          -- acknowledgement signal
    mod.token;   [Token.Size]INT32   -- network token
:

-- debug protocol (from units to debug procedure)
PROTOCOL Debug IS INT :: []BYTE:

----------------------------------------------------------------------------
-- token tags
----------------------------------------------------------------------------

-- stop token
VAL BYTE Stop.Token  IS #00(BYTE):

-- processor tokens (the more significant nibble of tag has to be 0 or 1)

-- message tokens (the more significant nibble of tag has to be 0!)
VAL BYTE Mess.PToken   IS #01(BYTE): -- value message
VAL BYTE Cont.PToken   IS #02(BYTE): -- context message
VAL BYTE PLoad.PToken  IS #03(BYTE): -- processor load message

-- request tokens (the more significant nibble of tag has to be 1!)
```

```
VAL BYTE FCall.PToken  IS #14(BYTE): -- fast call request
VAL BYTE TCall.PToken  IS #15(BYTE): -- tail call request

-- memory tokens (the more significant nibble of tag has to be 2 or 3)

-- cheap request tokens (the more significant nibble of tag has to be 2!)
VAL BYTE Alloc.MToken  IS #20(BYTE): -- allocation request
VAL BYTE Read.MToken   IS #21(BYTE): -- read request
VAL BYTE Submit.MToken IS #22(BYTE): -- submission request

-- expensive request tokens (the more significant nibble of tag has to be 3!)
VAL BYTE WrInt.MToken  IS #33(BYTE): -- write integer request
VAL BYTE WrPtr.MToken  IS #34(BYTE): -- write pointer request
VAL BYTE MLoad.MToken  IS #35(BYTE): -- memory load message
VAL BYTE IncRef.MToken IS #36(BYTE): -- increase reference request
VAL BYTE DecRef.MToken IS #37(BYTE): -- decrease reference request
VAL BYTE CopRef.MToken IS #38(BYTE): -- copy tuple reference

-----------------------------------------------------------------------------
-- End of ADNET.INC
-----------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------
--
-- ADCONN.OCC                                            Wolfgang Schreiner
-- Master Module Network Connection                      (c) RISC-Linz 1990
--
-- Connects the Adam master module to the network.
--
----------------------------------------------------------------------------

----------------------------------------------------------------------------
-- External References
----------------------------------------------------------------------------

-- include files
#INCLUDE "adglobal.inc" -- global constants
#INCLUDE "adnet.inc"    -- network constants and protocols
#INCLUDE "aderror.inc"  -- error messages
#INCLUDE "admaster.inc" -- master module constants and protocols
#INCLUDE "hostio.inc"   -- server constants and protocols

-- compilation units
#USE "hostio.lib"       -- server input/output routines

----------------------------------------------------------------------------
-- Connect Adam Master to Network
--
--   This procedure represents the connection between master module and
--   network. It can be placed between any two modules of the network
--   and is totally transparent for these two modules, i.e. all incoming
--   tokens are unchanged forwarded. The only exception are tokens that
--   are destined for the master module. They are filtered by the
--   connection and sent to the master module.
--
--   The modules on both sides of the connector are regarded as the roots
--   of two independent module trees to each of which the program is
--   forwarded, from each of which acknowledge signals are received and to
--   each of which stop tokens are sent. However, the trigger for the
--   start of the program execution is always sent to the left module.
--
-- IN  from.left, from.right:  input channels from connected modules
-- OUT to.left, to.right:      output channels to connected modules
-- IO  from.result, to.result: channels from/to result generator
-- OUT to.stat:                output channel to statistics generator
-- OUT from.inout, to.inout:   channels from/to input/output interface
-- IO  fs, ts:                 input/output channels from/to server
----------------------------------------------------------------------------
PROC Adam.Connector(CHAN OF Net.Link from.left, to.right, from.right,
  to.left, from.result, to.result, to.stat, from.inout, to.inout,
  CHAN OF SP fs, ts)

  BOOL going:      -- termination variable
  BOOL waiting:    -- termination variable
  BOOL append:     -- append signal received
  BYTE addr:       -- token address
  BYTE type:       -- token type
```

```
BOOL wait.left:  -- waiting for event from left module
BOOL wait.right: -- waiting for acknowledge from right module

[Token.Size]INT32 token:     -- network token
[Cluster.Size]INT32 cluster: -- program cluster

---------------------------------------------------------------------------
-- Assert Condition
--
--   Checks condition and prints error message to screen,
--   if neccessary.
--
-- IN cond: test condition
-- IN mess: error message
---------------------------------------------------------------------------
PROC Assert(VAL BOOL cond, VAL []BYTE mess)

  -------------------------------------------------------------------------
  -- Body of Assert
  -------------------------------------------------------------------------
  SEQ
    IF
      cond
        SKIP
      TRUE
        SEQ
          so.write.string.nl(fs, ts, adam.assert)
          so.write.string.nl(fs, ts, mess)
          STOP
:

---------------------------------------------------------------------------
-- Body of Adam.Connector
---------------------------------------------------------------------------
SEQ

  WHILE TRUE
    SEQ

      ---------------------------------------------------------------------
      -- transmit program to network
      ---------------------------------------------------------------------

      -- receive signal for start of program transmission
      from.inout ? CASE
        net.startprog
          SEQ
            to.left ! net.startprog
            to.right ! net.startprog
            append := FALSE
        net.appendprog
          SEQ
            to.left ! net.appendprog
            to.right ! net.appendprog
            append := TRUE
```

```

-- receive program clusters and forward them to both modules
going := TRUE
WHILE going
  from.inout ? CASE
    net.program; cluster                 -- program cluster
      SEQ
        to.left ! net.program; cluster
        to.right ! net.program; cluster
    net.appendprog                       -- append signal
      SEQ                                -- (must not occur twice)
        Assert(NOT append, app.err)
        to.left ! net.appendprog
        to.right ! net.appendprog
        append := TRUE
    net.endprog                          -- end of program signal
      SEQ
        to.left ! net.endprog
        to.right ! net.endprog
        going := FALSE

-- wait for acknowledge signals and forward signal to interface
from.left ? CASE net.ack
from.right ? CASE net.ack
to.inout ! net.ack

---------------------------------------------------------------------
-- handle result requests from result generator
---------------------------------------------------------------------

-- receive requests from result generator and return results to it,
-- and also handle token transmissions between modules
going := TRUE
WHILE going
  ALT

    -- token received from result generator
    from.result ? CASE net.token; token
      SEQ
        -- extract type of token
        [4]BYTE toks RETYPES token[2]:
        type := toks[1]
        going := type <> Stop.Token
        IF
          going
            -- forward request to left module
            to.left ! net.token; token
          TRUE
            SEQ
              -- forward stop token to both modules
              to.left ! net.token; token
              to.right ! net.token; token

    -- token received from left neighbour
    from.left ? CASE net.token; token
```

```
      SEQ
        [4]BYTE toks RETYPES token[2]:
        SEQ
          addr := toks[2]
          type := toks[1]
        CASE addr
          0(BYTE)
            -- token received for master module
            CASE type
              Mess.PToken
                -- result value received
                to.result ! net.token; token
              ELSE
                -- statistics received
                to.stat ! net.token; token
          ELSE
            -- forward token to right neighbour
            to.right ! net.token; token

    -- token received from right neighbour
    from.right ? CASE net.token; token
      SEQ
        [4]BYTE toks RETYPES token[2]:
        SEQ
          addr := toks[2]
          type := toks[1]
        CASE addr
          0(BYTE)
            -- token received for master module
            CASE type
              Mess.PToken
                -- result value received
                to.result ! net.token; token
              ELSE
                -- statistics received
                to.stat ! net.token; token
          ELSE
            -- forward token to left neighbour
            to.left ! net.token; token

--------------------------------------------------------------------
-- terminate program execution
--------------------------------------------------------------------

-- wait for acknowledge signals and skip tokens
wait.left, wait.right := TRUE, TRUE
WHILE (wait.left OR wait.right)
  ALT
    wait.left & from.left ? CASE
      net.token; token
        SKIP
      net.ack
        wait.left := FALSE
    wait.right & from.right ? CASE
      net.token; token
```

```
                SKIP
              net.ack
                wait.right := FALSE

        -- send acknowledge signal to input/output interface
        to.inout ! net.ack

    -- never done (just to avoid compiler warnings, if not used elsewhere)
    so.write.string(fs, ts, "")

:
-----------------------------------------------------------------------------
-- End of ADCONN.OCC
-----------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------
--
-- ADGENER.OCC                                            Wolfgang Schreiner
-- Result Generator                                       (c) RISC-Linz 1990
--
-- Triggers generation of result and converts it into printable form.
--
----------------------------------------------------------------------------

----------------------------------------------------------------------------
-- External References
----------------------------------------------------------------------------

-- include files
#INCLUDE "adglobal.inc" -- global constants
#INCLUDE "aderror.inc"  -- error messages
#INCLUDE "adnet.inc"    -- network constants and protocols
#INCLUDE "admaster.inc" -- master protocols and constants
#INCLUDE "hostio.inc"   -- server constants and protocols

-- compilation units
#USE "string.lib"       -- string handling facilities
#USE "hostio.lib"       -- server input/output routines

----------------------------------------------------------------------------
-- Result Generator
--
--  Triggers program execution, receives result tokens and transforms
--  result values in printable form.
--
-- IN  from.net:   input channel from network
-- OUT to.net:     output channel to network
-- IN  from.inout: input channel from input/output interface
-- OUT to.inout:   output channel to input/output interface
-- IO  fs, ts:     interface to server (for debugging purposes)
----------------------------------------------------------------------------
PROC Adam.Generator(CHAN OF Net.Link from.net, to.net,
  CHAN OF Result to.inout, from.inout, CHAN OF SP fs, ts)

  [CType.Size]BYTE type:   -- type description characters
  [CLabel.Size]BYTE label: -- positions of type descriptor lines
  [Token.Size]INT32 token: -- network token

  INT fun:  -- number of function to be activated
  INT root: -- root of type description tree

  --------------------------------------------------------------------------
  -- Assert Condition
  --
  --   Checks condition and prints error message to screen,
  --   if neccessary.
  --
  -- IN cond: test condition
  -- IN mess: error message
  --------------------------------------------------------------------------
```

```
PROC Assert(VAL BOOL cond, VAL []BYTE mess)

  ------------------------------------------------------------------------
  -- Body of Assert
  ------------------------------------------------------------------------
  SEQ
    IF
      cond
        SKIP
      TRUE
        SEQ
          so.write.string.nl(fs, ts, adam.assert)
          so.write.string.nl(fs, ts, mess)
          STOP
:

--------------------------------------------------------------------------
-- Compile Type Descriptor File
--
--   Receives type descriptor file and compiles it into standard form.
--   If type descriptor invalid, reports this fact to inout interface
--   and waits for new type descriptor file. Only returns after having
--   compiled valid type description.
--
-- OUT type:  array of type descriptions
-- OUT label: array of label references
-- OUT root:  root of type descriptions
-- OUT fun:   number of function to be activated
--------------------------------------------------------------------------
PROC Compile.Type([CType.Size]BYTE type, [CLabel.Size]BYTE label,
  INT root, INT fun)

  BYTE pos:   -- actual position in type array
  INT  len:   -- length of actual text line
  INT  i:     -- actual position in actual text line
  BOOL okay:  -- compilation successfull?
  BOOL going: -- termination variable

  [Type.Size]BYTE  line:    -- actual text line
  [CType.Size]BOOL used:    -- which labels have been used?
  [CType.Size]BOOL defined: -- which labels have been defined?

  ------------------------------------------------------------------------
  -- Skip White Space
  --
  --   Skips blanks and comments from line
  --
  -- IN okay: no error occurred until now
  -- IN line: actual text line
  -- IN len:  length of line
  -- IO i:    starting position of search, afterwards position of
  --          first non-blank character (= len, if none)
  ------------------------------------------------------------------------
  PROC White.Space(VAL BOOL okay, VAL []BYTE line, VAL INT len, INT i)

```

```
    BOOL going: -- termination variable

    ----------------------------------------------------------------------
    -- Body of White.Space
    ----------------------------------------------------------------------
    IF
      okay
        SEQ
          going := i < len
          WHILE going
            CASE line[i]
              ' '       -- blank
                SEQ
                  i := i+1
                  going := i<len
              '*t'      -- tabulator
                SEQ
                  i := i+1
                  going := i<len
              Comm.Chr  -- comment
                SEQ
                  i := len
                  going := FALSE
              ELSE      -- non-blank character
                going := FALSE
      TRUE
        -- error already occurred, skip line
        SKIP
  :

  ------------------------------------------------------------------------
  -- Get Unsigned Number
  --
  --   Reads unsigned number from line.
  --
  -- IO okay: no error occurred until now
  -- IN line: actual text line
  -- IN len:  length of line
  -- IO i:    start position for search, position of first non-number
  --          character afterwards (= len, if none)
  -- OUT num: result number
  ------------------------------------------------------------------------
  PROC Number(BOOL okay, VAL []BYTE line, VAL INT len, INT i, INT num)

    BOOL going: -- termination variable

    ----------------------------------------------------------------------
    -- Body of Number
    ----------------------------------------------------------------------
    SEQ
      -- skip white space
      White.Space(okay, line, len, i)
      okay := okay AND (i < len) AND is.digit(line[i])
      -- read number
      going := okay
```

```
    num := 0
    WHILE going
      SEQ
        num := (10*num)+((INT line[i])-48)
        i := i+1
        going := (i < len) AND is.digit(line[i])
:

------------------------------------------------------------------------
-- Determine Root of Type Description
--
--   The first non-whitespace line of type description has to contain a
--   root description of the form  "#<number1>:<number2>" where <number1>
--   is the number of the function to be activated and <number2> is the
--   label of the line that describes the result type of the function.
--   This procedure reads all lines up to this line and returns the
--   function number and the label of the root.
--
-- OUT okay: everything is okay until now
-- OUT root: label of root (only iff okay)
-- OUT fun:  function number
------------------------------------------------------------------------
PROC Compile.Root(BOOL okay, INT root, INT fun)

  INT i:      -- actual line position
  BOOL going: -- termination variable
  INT len:    -- length of actual line

  [Type.Size]BYTE line: -- actual line of file

  ----------------------------------------------------------------------
  -- Body of Compile.Root
  ----------------------------------------------------------------------
  SEQ

    going := TRUE
    WHILE going
      SEQ

        -- fetch line from file
        from.inout ? CASE
          is.type; len :: line
            i := 0
          end.type
            okay := FALSE

        -- skip white space
        White.Space(okay, line, len, i)

        IF
          NOT okay
            -- terminate loop
            going := FALSE
          i = len
            -- skip empty line
```

```
      SKIP
     TRUE
      -- read root description
      SEQ
       -- root character
       IF
        line[i] = Root.Chr
         i := i+1
        TRUE
         okay := FALSE
       -- function number
       Number(okay, line, len, i, fun)
       -- skip white space
       White.Space(okay, line, len, i)
       -- label character
       okay := okay AND (i < len)
       IF
        okay AND (line[i] = Label.Chr)
         i := i+1
        TRUE
         okay := FALSE
       -- label number
       Number(okay, line, len, i, root)
       okay := okay AND (root < CLabel.Size)
       -- rest of line only white space
       White.Space(okay, line, len, i)
       okay := okay AND (i = len)
       -- terminate loop
       going := FALSE
:

-------------------------------------------------------------------------
-- Get Type Label
--
--  Reads type label from line and enters it into type description
--
-- IO okay: no error occurred until now
-- IN line: actual text line
-- IN len:  length of line
-- IO i:    start position for search, position of first non-label
--          character afterwards (= len, if none)
-- IN pos:  next position in type descriptor
-------------------------------------------------------------------------
PROC Type.Label(BOOL okay, VAL []BYTE line, VAL INT len, INT i,
 VAL BYTE pos)

 INT  num:   -- label number

 -----------------------------------------------------------------------
 -- Body of Type.Label
 -----------------------------------------------------------------------
 SEQ
  -- get unsigned label number
  Number(okay, line, len, i, num)
  -- number must be new label
```

```
    okay := okay AND (num < CLabel.Size) AND (NOT defined[num])
    -- enter label
    IF
      okay
        SEQ
          Assert(NOT defined[num], type.redef)
          defined[num] := TRUE -- label defined
          label[num] := pos    -- label references type description
      TRUE
        SKIP
:

------------------------------------------------------------------------
-- Get Type Description
--
--  Reads type description from line and enters it into type array
--
-- IO okay: no error occurred until now
-- IN line: actual text line
-- IN len:  length of line
-- IO i:    start position for search, position of first non-type
--          character afterwards (= len, if none)
-- IO pos:  next position in type descriptor
------------------------------------------------------------------------
PROC Type.Descr(BOOL okay, VAL []BYTE line, VAL INT len, INT i,
  BYTE pos)

  INT num: -- actual type label

  ----------------------------------------------------------------------
  -- Body of Type.Descr
  ----------------------------------------------------------------------
  SEQ

    -- Skip whitespace
    White.Space(okay, line, len, i)

    IF
      okay AND (i < len)

        -- recognize types
        CASE line[i]

          -- atomic types
          Int.Chr, Bool.Chr, Char.Chr, Str.Chr
            SEQ
              -- enter type
              type[(INT pos)] := line[i]
              type[(INT pos)+1] := End.Type
              pos, i := BYTE((INT pos)+2), i+1
              -- rest of line whitespace
              White.Space(okay, line, len, i)
              okay := okay AND (i = len)

          -- list type
```

```
            List.Chr
              SEQ
                -- enter type
                type[INT pos] := line[i]
                pos, i := BYTE((INT pos)+1), i+1
                -- read label number
                Number(okay, line, len, i, num)
                okay := okay AND (num < CLabel.Size)
                IF
                  okay
                    -- enter label
                    SEQ
                      used[num] := TRUE
                      type[INT pos] := (BYTE num)
                      type[(INT pos)+1] := End.Type
                      pos := BYTE((INT pos)+2)
                  TRUE
                    SKIP

            -- tuple or union type
            Tuple.Chr, Union.Chr
              SEQ
                -- enter type
                type[INT pos] := line[i]
                pos, i := BYTE((INT pos)+1), i+1
                -- at least one label
                okay := i < len
                -- enter type labels
                WHILE okay AND (i < len)
                  SEQ
                    -- read label number
                    Number(okay, line, len, i, num)
                    okay := okay AND (num < CLabel.Size)
                    IF
                      okay
                        -- enter label
                        SEQ
                          used[num] := TRUE
                          type[INT pos] := (BYTE num)
                          pos := BYTE((INT pos)+1)
                      TRUE
                        SKIP
                    -- skip white space
                    White.Space(okay, line, len, i)
                -- signal end of type
                type[INT pos] := End.Type
                pos := BYTE ((INT pos)+1)

            -- unknown type
            ELSE
              okay := FALSE

        -- error already occurred
        TRUE
          okay := FALSE
```

```
:

-----------------------------------------------------------------------
-- Compile Type Descriptor Line
--
--   Receives one line of type descriptor and compiles it into
--   standard form.
--
-- IO okay:  actual line and all previous lines successfully compiled
-- IN line:  actual line to be compiled
-- IN len:   length of actual line
-- IO pos:   actual position in type array
-----------------------------------------------------------------------
PROC Compile.Line(BOOL okay, [Type.Size]BYTE line, INT len, BYTE pos)

  ---------------------------------------------------------------------
  -- Body of Compile.Line
  ---------------------------------------------------------------------
  SEQ

    -- actual position in line
    i := 0

    -- skip blanks and comments
    White.Space(okay, line, len, i)

    IF
      (NOT okay) OR (i = len)
        -- skip line
        SKIP
      TRUE
        -- compile line
        SEQ
          -- type label
          Type.Label(okay, line, len, i, pos)
          -- white space
          White.Space(okay, line, len, i)
          -- label character
          IF
            okay AND (i < len) AND (line[i] = Label.Chr)
              i := i+1
            TRUE
              okay := FALSE
          -- actual type description
          Type.Descr(okay, line, len, i, pos)
          -- rest of line whitespace
          White.Space(okay, line, len, i)
          okay := okay AND (i = len)

:

-----------------------------------------------------------------------
-- Body of Compile.Type
-----------------------------------------------------------------------
SEQ
```

```

    -- receive type description and compile it successfully
    okay := FALSE
    WHILE NOT okay
      SEQ

        -- no error occurred until now
        okay := TRUE

        -- initalize label references
        SEQ i = 0 FOR CType.Size
          defined[i], used[i] := FALSE, FALSE

        -- position of next free item in type array
        pos := 0(BYTE)

        -- compile root description
        Compile.Root(okay, root, fun)

        -- compile type description
        going := TRUE
        WHILE going
          from.inout ? CASE
            is.type; len :: line
              -- receive type descriptor line
              Compile.Line(okay, line, len, pos)
            end.type
              -- receive end of type description signal
              going := FALSE

        -- check if all referenced labels defined
        SEQ i=0 FOR CLabel.Size
          okay := okay AND ((NOT used[i]) OR (defined[i]))

        -- signal success of compilation
        to.inout ! type.succ; okay

:

---------------------------------------------------------------------------
-- Convert Result Tokens into Printable Form
--
--   Receives result tokens, converts them in printable form and forwards
--   them to inout interface. If the result type is a structure, it
--   releases request tokens to network in order to receive the components
--   of the structure.
--
-- IN type:  array of type descriptions
-- IN label: array of label references
-- IN root:  root of type descriptions
---------------------------------------------------------------------------
PROC Result([CType.Size]BYTE type, [CLabel.Size]BYTE label, INT root)

  INT   top:     -- top of stack
  INT   variant: -- tuple variant
```

```
INT32 result:  -- contents of message token
INT   len:     -- lenght of output string
BOOL  error:   -- value has error flag set

[Line.Size]BYTE line:    -- output string
[Token.Size]INT32 token: -- network token

-- top elements of stacks
INT   act.type:  -- actual type pointer
INT   act.field: -- actual field offset
BYTE  act.entry: -- actual entry point
INT32 act.list:  -- actual list pointer

-- stacks
[Stack.Size]INT   stack.type:  -- stack of pointers to type characters
[Stack.Size]INT   stack.field: -- stack of field offsets
[Stack.Size]BYTE  stack.entry: -- stack of entry points
[Stack.Size]INT32 stack.list:  -- stack of list pointers

--------------------------------------------------------------------------
-- Receive Token
--
--   Receives message token and checks message for error flag.
--
-- OUT result: value of message
-- OUT string: error message (if error = TRUE)
-- OUT len:    length of error message (if error, 0 otherwise)
-- OUT error:  error flag set
--------------------------------------------------------------------------
PROC Receive(INT32 result, [Line.Size]BYTE string, INT len, BOOL error)

  [Token.Size]INT32 token: -- network token

  ------------------------------------------------------------------------
  -- Body of Receive
  ------------------------------------------------------------------------
  SEQ

    -- wait for token
    from.net ? CASE net.token; token

    -- token must be message token
    [4]BYTE toks RETYPES token[2]:
    Assert(toks[1]=Mess.PToken, token.type)

    -- extract result
    result := token[0]

    -- error flag set?
    error := (result /\ 1(INT32)) = 1(INT32)
    IF
      error
        SEQ
          len := 6
          [string FROM 0 FOR len] := "ERROR("
```

```
          append.int(len, string, INT (result >> 1), 0)
          string[len] := ')'
          len := len+1
      TRUE
        SKIP
:

------------------------------------------------------------------------
-- Copy Tuple Reference
--
--   Copies tuple reference by decreasing the weight field of the
--   reference. If the weight decreases to zero a 'increase reference'
--   message is sent to the denoted memory module. This module returns
--   a new reference (with a non-zero weight field) that may then be
--   copied.
--
-- IO ref: tuple reference to be copied.
------------------------------------------------------------------------
PROC Copy(INT32 ref)

  INT wght: -- reference weight field

  [Token.Size]INT32 token: -- network token

  ----------------------------------------------------------------------
  -- Body of Copy
  ----------------------------------------------------------------------
  IF
    ref = 0(INT32)

      -- nil pointer
      SKIP

    (ref /\ #00000001(INT32)) > 0(INT32)

      -- error flag set
      SKIP

    TRUE
      SEQ

        -- actual reference weight
        wght := (INT (ref >> 1)) /\ #00000003

        IF
          wght = 0
            SEQ

              -- construct request token
              [4]BYTE t RETYPES token[2]:
              [4]BYTE r RETYPES ref:
              SEQ
                t[2] := r[3]              -- module address
                t[1] := IncRef.MToken     -- increase reference request
              token[0] := ref             -- tuple reference
```

```
                [4]BYTE t RETYPES token[0]:
                t[3] := 0(BYTE)              -- actual module address

                -- send token into network
                to.net ! net.token; token

                -- receive token from network
                from.net ? CASE net.token; token

                -- extract new reference
                ref := token[0]
                wght := (INT (ref >> 1)) /\ #00000003

            TRUE
              SKIP

          -- decrease weight field
          ref := (ref /\ #FFFFFFF9(INT32)) \/ (INT32 ((wght-1) << 1))
:

------------------------------------------------------------------------
-- Body of Result
------------------------------------------------------------------------
SEQ

  -- push root of type expression on stack
  stack.type[0] := INT label[root] -- pointer to root type character
  stack.entry[0] := 0(BYTE)        -- actual entry point
  top := 0                         -- top of stack

  WHILE top >= 0
    SEQ

      -- pop actual values from stack
      act.type  := stack.type[top]
      act.field := stack.field[top]
      act.entry := stack.entry[top]
      act.list  := stack.list[top]
      top := top-1

      CASE type[act.type]

        ----------------------------------------------------------------
        -- integer value
        ----------------------------------------------------------------
        Int.Chr
          SEQ
            Receive(result, line, len, error)
            IF
              error
                SKIP
              TRUE
                SEQ
                  len := 0
                  append.int(len, line, ASHIFTRIGHT(INT result, 1), 0)
```

```
--------------------------------------------------------------
-- boolean value
--------------------------------------------------------------
Bool.Chr
  SEQ
    Receive(result, line, len, error)
    IF
      error
        SKIP
      ASHIFTRIGHT(INT result, 1) = 1
        SEQ
          len := 4
          [line FROM 0 FOR len] := "TRUE"
      TRUE
        SEQ
          len := 5
          [line FROM 0 FOR len] := "FALSE"

--------------------------------------------------------------
-- character value
--------------------------------------------------------------
Char.Chr
  SEQ
    Receive(result, line, len, error)
    IF
      error
        SKIP
      TRUE
        SEQ
          line[0] := '*''
          result := INT32 ASHIFTRIGHT(INT result, 1)
          VAL [4]BYTE chr RETYPES result:
          line[1] := chr[0]
          line[2] := '*''
          len := 3

--------------------------------------------------------------
-- string value
--------------------------------------------------------------
Str.Chr

  CASE act.entry

    ----------------------------------------------------------
    -- receive pointer to string
    ----------------------------------------------------------
    0(BYTE)
      SEQ

        -- receive token with pointer to string
        Receive(result, line, len, error)
        IF
          error
            SKIP
```

```
      TRUE
        SEQ
          -- push continuation on stack
          top := top+1
          stack.type[top] := act.type
          stack.list[top] := result
          stack.entry[top] := 1(BYTE)
          -- construct print message
          len := 1
          line[0] := '*"'

------------------------------------------------------------
-- non-error pointer received, character of string required
------------------------------------------------------------
1(BYTE)

  IF

    act.list = 0(INT32)

      -- rest line is empty
      SEQ
        -- construct print message
        len := 1
        line[0] := '*"'

    TRUE

      -- request next character from line
      SEQ

        -- copy string reference
        Copy(act.list)

        -- push continuation on stack
        top := top+1
        stack.type[top] := act.type
        stack.list[top] := act.list
        stack.entry[top] := 2(BYTE)

        -- construct request token
        [4]BYTE act RETYPES act.list:
        BYTE    mod  IS act[3]:
        [3]BYTE ref  IS [act FROM 0 FOR 3]:
        SEQ
          [4]BYTE toks RETYPES token[2]:
          SEQ
            toks[2] := mod
            toks[1] := Read.MToken
            toks[0] := 0(BYTE)
          -- ignore context and offset (not required)
          [4]BYTE toks RETYPES token[0]:
          SEQ
            toks[3] := 0(BYTE)
            [toks FROM 0 FOR 3] := ref
```

```

        -- send token into network
        to.net ! net.token; token

        -- receive result
        Receive(result, line, len, error)

        IF
          error
            SKIP
          TRUE
            SEQ
              -- construct print message
              len := 1
              result := result >> 1
              VAL [4]BYTE res RETYPES result:
              line[0] := res[0]

------------------------------------------------------------
-- tail of string is required
------------------------------------------------------------
2(BYTE)
  SEQ

    -- construct read token for rest of string
    [4]BYTE act RETYPES act.list:
    BYTE    mod  IS act[3]:
    [3]BYTE ref  IS [act FROM 0 FOR 3]:
    SEQ
      [4]BYTE toks RETYPES token[2]:
      SEQ
        toks[2] := mod
        toks[1] := Read.MToken
        toks[0] := 1(BYTE)
      -- ignore context and offset (not required)
      [4]BYTE toks RETYPES token[0]:
      SEQ
        toks[3] := 0(BYTE)
        [toks FROM 0 FOR 3] := ref

    -- send token to network
    to.net ! net.token; token

    -- receive pointer to string
    Receive(result, line, len, error)

    IF
      error
        SKIP
      TRUE
        SEQ
          -- push continuation on stack
          top := top+1
          stack.entry[top] := 1(BYTE)
          stack.type[top] := act.type
```

```
                stack.list[top] := result
                -- empty print message
                len := 0

--------------------------------------------------------------
-- list value
--------------------------------------------------------------
List.Chr

  CASE act.entry

    ----------------------------------------------------------
    -- start of list encountered
    ----------------------------------------------------------
    0(BYTE)
      SEQ

        -- receive message token with pointer to list
        Receive(result, line, len, error)

        IF
          error
            SKIP
          result = 0(INT32)
            SEQ
              -- construct print message
              len := 2
              [line FROM 0 FOR 2] := "[]"
          TRUE
            SEQ
              -- construct print message
              len := 1
              line[0] := '['
              -- push continutation on stack
              top := top+1
              stack.type[top] := act.type
              stack.list[top] := result
              stack.entry[top] := 2(BYTE)

    ----------------------------------------------------------
    -- continuation of list encountered
    ----------------------------------------------------------
    1(BYTE)
      SEQ

        -- receive message token with pointer to tail of list
        Receive(result, line, len, error)

        IF
          error
            SKIP
          result = 0(INT32)
            SEQ
              -- construct print message
              len := 1
```

```
          line[0] := ']'
      TRUE
        SEQ
          -- construct print message
          len := 1
          line[0] := ','
          -- push continutation on stack
          top := top+1
          stack.type[top] := act.type
          stack.list[top] := result
          stack.entry[top] := 2(BYTE)

--------------------------------------------------------------
-- head of list is required
--------------------------------------------------------------
2(BYTE)
  SEQ

    -- copy list reference
    Copy(act.list)

    -- push list continutation on stack
    top := top+1
    stack.type[top]  := act.type
    stack.list[top]  := act.list
    stack.entry[top] := 3(BYTE)

    -- push element type and pointer on stack
    top := top+1
    stack.type[top] :=
      INT label[INT type[act.type+1]]
    stack.entry[top] := 0(BYTE)

    -- construct read token for head of list
    [4]BYTE act RETYPES act.list:
    BYTE    mod  IS act[3]:
    [3]BYTE ref  IS [act FROM 0 FOR 3]:
    SEQ
      [4]BYTE toks RETYPES token[2]:
      SEQ
        toks[2] := mod
        toks[1] := Read.MToken
        toks[0] := 0(BYTE)
      -- ignore context and offset (not required)
      [4]BYTE toks RETYPES token[0]:
      SEQ
        toks[3] := 0(BYTE)
        [toks FROM 0 FOR 3] := ref

      -- send token to network
      to.net ! net.token; token

      -- empty print message
      len := 0
```

```
          -------------------------------------------------------------
          -- tail of list is required
          -------------------------------------------------------------
          3(BYTE)
            SEQ

              -- push continuation on stack
              top := top + 1
              stack.type[top]  := act.type
              stack.entry[top] := 1(BYTE)

              -- construct read token for rest of list
              [4]BYTE act RETYPES act.list:
              BYTE    mod  IS act[3]:
              [3]BYTE ref  IS [act FROM 0 FOR 3]:
              SEQ
                [4]BYTE toks RETYPES token[2]:
                SEQ
                  toks[2] := mod
                  toks[1] := Read.MToken
                  toks[0] := 1(BYTE)
                -- ignore context and offset (not required)
                [4]BYTE toks RETYPES token[0]:
                SEQ
                  toks[3] := 0(BYTE)
                  [toks FROM 0 FOR 3] := ref

                -- send token to network
                to.net ! net.token; token

                -- empty print message
                len := 0

      -----------------------------------------------------------------
      -- tuple value
      -----------------------------------------------------------------
      Tuple.Chr

        CASE act.entry

          -------------------------------------------------------------
          -- start of tuple encountered
          -------------------------------------------------------------
          0(BYTE)
            SEQ

              -- receive message token with pointer to tuple
              Receive(result, line, len, error)

              IF
                error
                  SKIP
                TRUE
                  SEQ

```

```
          -- construct print message
          len := 1
          line[0] := '('

          -- push continuation on stack
          top := top+1
          stack.type[top] := act.type
          stack.field[top] := 0
          stack.list[top] := result
          stack.entry[top] := 2(BYTE)

-------------------------------------------------------------
-- continuation of tuple encountered
-------------------------------------------------------------
1(BYTE)

  IF
    type[act.type+(act.field+2)] = End.Type
      SEQ
        -- construct print message
        len := 1
        line[0] := ')'
    TRUE
      SEQ
        -- construct print message
        len := 1
        line[0] := ','
        -- push continuation on stack
        top := top+1
        stack.type[top] := act.type
        stack.field[top] := act.field+1
        stack.list[top] := act.list
        stack.entry[top] := 2(BYTE)

-------------------------------------------------------------
-- tuple field is to be read
-------------------------------------------------------------
2(BYTE)
  SEQ

    -- copy tuple reference
    Copy(act.list)

    -- push continuation on stack
    top := top+1
    stack.type[top]  := act.type
    stack.field[top] := act.field
    stack.list[top] := act.list
    stack.entry[top] := 1(BYTE)

    -- push tuple element type on stack
    top := top + 1
    stack.type[top] :=
      INT label[INT type[act.type+(act.field+1)]]
    stack.entry[top] := 0(BYTE)
```

```

              -- construct read request for tuple field
              [4]BYTE act RETYPES act.list:
              BYTE    mod IS act[3]:
              [3]BYTE ref IS [act FROM 0 FOR 3]:
              SEQ
                [4]BYTE toks RETYPES token[2]:
                SEQ
                  toks[2] := mod
                  toks[1] := Read.MToken
                  toks[0] := BYTE act.field
                [4]BYTE toks RETYPES token[0]:
                SEQ
                  toks[3] := 0(BYTE)
                  [toks FROM 0 FOR 3] := ref

              -- send token into network
              to.net ! net.token; token

              -- empty print message
              len := 0

      ---------------------------------------------------------------
      -- union value
      ---------------------------------------------------------------
      Union.Chr
        SEQ

          -- receive message
          Receive(result, line, len, error)
          act.list := result

          IF
            error
              SKIP
            TRUE
              SEQ

                -- copy union reference
                Copy(act.list)

                -- construct request token for variant part
                [4]BYTE act RETYPES act.list:
                BYTE    mod IS act[3]:
                [3]BYTE ref IS [act FROM 0 FOR 3]:
                SEQ
                  [4]BYTE toks RETYPES token[2]:
                  SEQ
                    toks[2] := mod
                    toks[1] := Read.MToken
                    toks[0] := 0(BYTE)
                  [4]BYTE toks RETYPES token[0]:
                  SEQ
                    toks[3] := 0(BYTE)
                    [toks FROM 0 FOR 3] := ref
```

```

                    -- send token to network
                    to.net ! net.token; token

                    -- receive result token from network
                    Receive(result, line, len, error)
                    Assert(NOT error, union.err)
                    variant := INT(result >> 1)
                    SEQ i= act.type FOR variant
                      Assert(type[i]<>End.Type, union.err)

                    -- construct print message
                    len := 0
                    append.int(len, line, variant, 0)
                    line[len] := '$'
                    len := len+1

                    -- push continuation on stack
                    top := top+1
                    stack.type[top] :=
                      INT label[INT type[act.type+variant]]
                    stack.entry[top] := 0(BYTE)

                    -- construct request token for element part
                    [4]BYTE act RETYPES act.list:
                    BYTE    mod IS act[3]:
                    [3]BYTE ref IS [act FROM 0 FOR 3]:
                    SEQ
                      [4]BYTE toks RETYPES token[2]:
                      SEQ
                        toks[2] := mod
                        toks[1] := Read.MToken
                        toks[0] := 1(BYTE)
                      [4]BYTE toks RETYPES token[0]:
                      SEQ
                        toks[3] := 0(BYTE)
                        [toks FROM 0 FOR 3] := ref

                      -- send token to network
                      to.net ! net.token; token

        -------------------------------------------------------------------
        -- send result message to inout interface
        -------------------------------------------------------------------
        to.inout ! is.result; len :: line

:

---------------------------------------------------------------------------
-- Body of Generator
---------------------------------------------------------------------------
SEQ

  -- loop forever
  WHILE TRUE
```

```
      SEQ

        -- receive and compile type description
        Compile.Type(type, label, root, fun)

        -- construct activation token
        [4]BYTE toks RETYPES token[2]:
        SEQ
          toks[2] := 1(BYTE)            -- address of first module
          toks[1] := FCall.PToken       -- fast call
        -- ignore context and offset fields (not required)
        token[0] := (INT32 fun)         -- function descriptor
        [4]BYTE toks RETYPES token[0]:
        toks[3] := 0(BYTE)              -- address of master module

        -- wait for start signal from inout interface
        from.inout ? CASE start

        -- send activation token to network
        to.net ! net.token; token

        -- receive activation address from network (unused)
        from.net ? CASE net.token; token

        -- receive result tokens, convert them into printable form and
        -- forward result to inout interface
        Result(type, label, root)

        -- send end of result signal
        to.inout ! end.result

        -- construct stop token
        [4]BYTE toks RETYPES token[2]:
        toks[1] := Stop.Token

        -- send stop token to network
        to.net ! net.token; token
:

-----------------------------------------------------------------------------
-- End of ADGENER.OCC
-----------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------
--
-- ADINOUT.OCC                                        Wolfgang Schreiner
-- Input/Output Interface                             (c) RISC-Linz 1990
--
-- Handles interaction of the ADAM architecture with the outside world.
--
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- External References
-----------------------------------------------------------------------------

-- include files
#INCLUDE "adglobal.inc" -- global constants
#INCLUDE "aderror.inc"  -- error messages
#INCLUDE "adnet.inc"    -- network constants and protocols
#INCLUDE "admaster.inc" -- master protocols and constants
#INCLUDE "hostio.inc"   -- input/output protocols and constants

-- compilation units
#USE "hostio.lib"       -- input/output interface to server
#USE "string.lib"       -- string handling facilities
#USE "convert.lib"      -- type conversion library

-----------------------------------------------------------------------------
-- Input/Output Interface to Server
--
--   Handles interaction with the outside world, i.e.
--
--      * reads program and forwards it to connector,
--      * reads type descriptor file and forwards it to result generator,
--      * receives result and prints it,
--      * receives execution statistics and prints it.
--
-- IN  from.net:    channel from that acknowledge signals are received
-- OUT to.net:      channel through that program is forwarded
-- OUT to.result:   channel through that type description is forwarded
-- IN  from.result: channel from that printable result is received
-- IN  from.stat:   channel from that statistics is received
-- IO  fs, ts:      input/output channels from/to server
-----------------------------------------------------------------------------
PROC Adam.In.Out(CHAN OF Net.Link from.net, to.net,
  CHAN OF Result from.result, to.result, CHAN OF Stat from.stat,
  CHAN OF SP fs, ts)

  -- parameters for command line parsing
  VAL [Opt.Num][]BYTE Opt.Str IS
    [Prog.Opt, Expr.Opt, Out.Opt, Load.Opt, Batch.Opt, Stat.Opt]:
  VAL [Opt.Num]INT Opt.Par IS
    [spopt.maybe, spopt.always, spopt.always,
      spopt.never, spopt.never, spopt.never]:

  -- result values of command line parsing
  [Cmd.Size]BYTE  command:   -- command line
```

```
[Opt.Num]BOOL   opt.exist: -- which options do exist
[Opt.Num][2]INT opt.par:   -- position of option parameters in command line
INT             error:     -- length of error message

-- option flags
BOOL prog:  -- name of program file given
BOOL expr:  -- name of expression file given
BOOL out:   -- name of output file given
BOOL load:  -- program already loaded
BOOL batch: -- work in batch mode
BOOL stat:  -- print statistics

-- directory and file names
[Name.Size]BYTE dir.name:  -- directory name
[Name.Size]BYTE prog.file: -- name of program file
[Name.Size]BYTE expr.file: -- name of expression file
[Name.Size]BYTE type.file: -- name of type file
[Name.Size]BYTE out.file:  -- name of output file

-- lengths of file names
INT prog.len: -- length of program file name
INT expr.len: -- length of expression file name
INT out.len:  -- length of output file name
INT type.len: -- length of type file name

[so.temp.filename.length]BYTE tmp.file: -- name of temporary file

-- stream identifiers:
INT32 out.stream: -- output stream (protocol of session)
INT32 tmp.stream: -- temporary stream (statistics)

-- time related variables
TIMER clock: -- transputer clock
INT   time:  -- time between two events
INT   time0: -- time when program started
INT   time1: -- time when program returned result
INT   time2: -- time when program terminated

-- miscellaneous variables
INT  dir.len: -- length of directory name
BOOL going:   -- termination variable
BOOL loop:    -- termination variable
INT  len:     -- length of string
BYTE key:     -- input character
BYTE result:  -- success of last file operation

[Line.Size]BYTE string: -- output string

--------------------------------------------------------------------------
-- Write String to Terminal and Output File
-- IN str: output string
--------------------------------------------------------------------------
PROC Write.String(VAL []BYTE str)

  BYTE result: -- success of operation
```

```

  ------------------------------------------------------------------------
  -- Body of Write.String
  ------------------------------------------------------------------------
  SEQ
    so.write.string(fs, ts, str)
    so.fwrite.string(fs, ts, out.stream, str, result)
:

--------------------------------------------------------------------------
-- Write New Line Character to Terminal and Output File
--------------------------------------------------------------------------
PROC Write.Nl()

  BYTE result: -- success of operation

  ------------------------------------------------------------------------
  -- Body of Write.Nl
  ------------------------------------------------------------------------
  SEQ
    so.write.nl(fs, ts)
    so.fwrite.nl(fs, ts, out.stream, result)
:

--------------------------------------------------------------------------
-- Assert Condition
--
--   Checks condition and prints error message to screen and output file,
--   if neccessary.
--
-- IN cond: test condition
-- IN mess: error message
--------------------------------------------------------------------------
PROC Assert(VAL BOOL cond, VAL []BYTE mess)

  ------------------------------------------------------------------------
  -- Body of Assert
  ------------------------------------------------------------------------
  SEQ
    IF
      cond
        SKIP
      TRUE
        SEQ
          Write.Nl()
          Write.String(adam.assert)
          Write.Nl()
          Write.String(mess)
          Write.Nl()
          STOP
:

--------------------------------------------------------------------------
-- Determine file names
--
```

```
--   Parses the command lines and determine names of program, expression,
--   and output files. Names that are not explicitly given are determined
--   by default rules and by user input. Missing names in batch mode
--   result in an error break of the program. After the call of this
--   procedure, all flags for that file names have been determined
--   are set to TRUE.
--
-- IO  prog:      name of program file given in command string
-- IO  expr:      name of expression file given in command string
-- IO  out:       name of output file given in command string
-- OUT prog.file: name of program file
-- OUT expr.file: name of expression file
-- OUT out.file:  name of output file
-- OUT type.file: name of type file
-- OUT prog.len:  length of program file name
-- OUT expr.len:  length of expression file name
-- OUT out.len:   length of output file name
-- OUT type.len:  length of type file name
--
-- ATTENTION: References several global variables!
---------------------------------------------------------------------------
PROC File.Names(BOOL prog, expr, out,
  []BYTE prog.file, expr.file, out.file, type.file,
  INT prog.len, expr.len, out.len, type.len)

  BOOL  exists: -- file exists
  BOOL  going:  -- termination variable
  INT32 stream: -- stream identifier
  BYTE  result: -- result of file operation

  -------------------------------------------------------------------------
  -- Append File Extension
  --
  --   Checks if file name contains extension. If not, a default
  --   extension is appended to name and length of name is updated.
  --   In any case, the default directory specification is inserted.
  --
  -- IO file: file name
  -- IO len:  length of name
  -- IN ext:  default extension
  -------------------------------------------------------------------------
  PROC Append.Ext([]BYTE file, INT len, VAL [Ext.Size]BYTE ext)

    INT  start: -- position of first non-empty character
    BYTE char:  -- first non-empty character
    BOOL error: -- error occurred

    -----------------------------------------------------------------------
    -- Body of Append.Ext
    -----------------------------------------------------------------------
    SEQ
      -- delete unused space
      SEQ i = len FOR (SIZE file)-len
        file[i] := Null
      -- append extension
```

```
    IF
      char.pos('.', file) = (-1)
        SEQ
          Assert((len+Ext.Size)<=Name.Size, file.len)
          [file FROM len FOR Ext.Size] := ext
          len := len+Ext.Size
      TRUE
        SKIP
    -- search for first non-empty character
    start, char := search.no.match([' ', Null], file)
    -- insert directory specification
    str.shift(file, start, len-start, dir.len-start, error)
    Assert(NOT error, file.len)
    [file FROM 0 FOR dir.len] := [dir.name FROM 0 FOR dir.len]
    len := len+(dir.len-start)
:

------------------------------------------------------------------------
-- Replace File Extension
--
--   Replaces the extension of a file name by another one.
--
-- IO file: file name with extension
-- IO len:  length of file name
-- IN ext:  new extension
------------------------------------------------------------------------
PROC Replace.Ext([]BYTE file, INT len, VAL [Ext.Size]BYTE ext)

  INT pos: -- position of dot in file name

  ----------------------------------------------------------------------
  -- Body of Replace.Ext
  ----------------------------------------------------------------------
  SEQ
    pos := char.pos('.', file)
    Assert(pos <> (-1), file.ext)
    Assert((pos+Ext.Size)<=Name.Size, file.len)
    [file FROM pos FOR Ext.Size] := ext
    len := pos+Ext.Size
:

------------------------------------------------------------------------
-- Test if File Exists
--
--   Tests if specified file exists and writes error message, if not.
--
-- IN  file:   name of file
-- IN  len:    length of file name
-- OUT exists: TRUE iff file exists
------------------------------------------------------------------------
PROC File.Exists(VAL []BYTE file, VAL INT len, BOOL exists)

  ----------------------------------------------------------------------
  -- Body of File.Exists
  ----------------------------------------------------------------------
```

```
      SEQ
        so.test.exists(fs, ts, file, exists)
        IF
          exists
            SKIP
          TRUE
            SEQ
              so.write.string(fs, ts, "File ")
              so.write.string(fs, ts, [file FROM 0 FOR len])
              so.write.string.nl(fs, ts, " not found.")
    :

    -----------------------------------------------------------------------
    -- Body of File.Names
    -----------------------------------------------------------------------
    SEQ

      ---------------------------------------------------------------------
      -- scan command line parameters
      ---------------------------------------------------------------------

      -- name of program file
      IF
        prog
          SEQ
            prog.len := opt.par[0][1]
            Assert(prog.len<=Name.Size, file.len)
            [prog.file FROM 0 FOR prog.len] :=
              [command FROM opt.par[0][0] FOR prog.len]
        TRUE
          SKIP

      -- name of expression file
      IF
        expr
          SEQ
            expr.len := opt.par[1][1]
            Assert(expr.len<=Name.Size, file.len)
            [expr.file FROM 0 FOR expr.len] :=
              [command FROM opt.par[1][0] FOR expr.len]
        TRUE
          SKIP

      -- name of output file
      IF
        out
          SEQ
            out.len  := opt.par[2][1]
            Assert(out.len<=Name.Size, file.len)
            [out.file FROM 0 FOR out.len] :=
              [command FROM opt.par[2][0] FOR out.len]
        TRUE
          SKIP

      ---------------------------------------------------------------------
```

```
      -- determine file names that are explicitly given
      -- and add default extensions, if required
      ----------------------------------------------------------------------

      -- program file explicitly given and existing?
      IF
        prog
          SEQ
            Append.Ext(prog.file, prog.len, Prog.Ext)
            File.Exists(prog.file, prog.len, prog)
        TRUE
          SKIP

      -- expression file explicitly given and existing?
      -- corresponding type file also existing?
      IF
        expr
          SEQ
            Append.Ext(expr.file, expr.len, Expr.Ext)
            File.Exists(expr.file, expr.len, expr)
            IF
              expr
                SEQ
                  type.file := expr.file
                  type.len := expr.len
                  Replace.Ext(type.file, type.len, Type.Ext)
                  File.Exists(type.file, type.len, expr)
              TRUE
                SKIP
        TRUE
          SKIP

      -- output file explicitly given?
      IF
        out
          Append.Ext(out.file, out.len, Out.Ext)
        TRUE
          SKIP

      ----------------------------------------------------------------------
      -- determine names of files that are not explicitly given
      -- and add extensions if required
      ----------------------------------------------------------------------

      -- determine name of program file
      IF
        prog OR load
          -- program file name found or not required
          SKIP
        batch
          -- no program file means error when working in batch mode
          Assert(FALSE, prog.miss)
        TRUE
          -- ask user for name of program file
          SEQ
```

```
          exists := FALSE
          WHILE (NOT exists) OR (result <> spr.ok)
            SEQ
              -- ask for file name
              so.write.string(fs, ts, "Enter name of program file (")
              so.write.string(fs, ts, [Prog.Ext FROM 0 FOR SIZE Prog.Ext])
              so.write.string(fs, ts, "): ")
              so.read.echo.line(fs, ts, prog.len, prog.file, result)
              IF
                result = spr.ok
                  SEQ
                    so.write.nl(fs, ts)
                    -- append extension, if required
                    Append.Ext(prog.file, prog.len, Prog.Ext)
                    -- test if file exists
                    File.Exists(prog.file, prog.len, exists)
                TRUE
                  SKIP
          -- set flag
          prog := TRUE

    -- determine name of expression file
    IF
      expr
        -- expression file name found
        SKIP
      batch
        -- no expression file means error when working in batch mode
        Assert(FALSE, expr.miss)
      TRUE
        -- ask user for name of expression file
        SEQ
          exists := FALSE
          WHILE (NOT exists) OR (result <> spr.ok)
            SEQ
              -- ask for file name
              so.write.string(fs, ts,
                "Enter name of expression file (")
              so.write.string(fs, ts, [Expr.Ext FROM 0 FOR SIZE Expr.Ext])
              so.write.string(fs, ts, "): ")
              so.read.echo.line(fs, ts, expr.len, expr.file, result)
              so.write.nl(fs, ts)
              IF
                result = spr.ok
                  SEQ
                    -- append extension, if required
                    Append.Ext(expr.file, expr.len, Expr.Ext)
                    -- test if file exists
                    File.Exists(expr.file, expr.len, exists)
                    IF
                      exists
                        SEQ
                          -- construct name of type file
                          type.file := expr.file
                          type.len := expr.len
```

```
                      Replace.Ext(type.file, type.len, Type.Ext)
                      -- test if file exists
                      File.Exists(type.file, type.len, exists)
                  TRUE
                    SKIP
            TRUE
              SKIP
      -- set flag
      expr := TRUE

-- determine name of output file
IF
  out
    -- output file name given
    SKIP
  prog
    -- take name of program file, add default extension
    SEQ
      out.file := prog.file
      out.len := prog.len
      Replace.Ext(out.file, out.len, Out.Ext)
      out := TRUE
  expr
    -- take name of expression file, add default extension
    SEQ
      out.file := expr.file
      out.len := expr.len
      Replace.Ext(out.file, out.len, Out.Ext)
      out := TRUE
  batch
    -- take default output file name
    SEQ
      [out.file FROM 0 FOR (SIZE Out.File)] := Out.File
      out := TRUE
  TRUE
    SEQ
      -- ask user for file name
      going := TRUE
      WHILE going
        SEQ
          so.write.string(fs, ts, "Enter name of output file (")
          so.write.string(fs, ts, [Out.Ext FROM 0 FOR SIZE Out.Ext])
          so.write.string(fs, ts, "): ")
          so.read.echo.line(fs, ts, out.len, out.file, result)
          so.write.nl(fs, ts)
          IF
            result = spr.ok
              SEQ
                -- append extension, if required
                Append.Ext(out.file, out.len, Out.Ext)
                -- try to open output file
                so.open(fs, ts, out.file, spt.text, spm.output,
                  stream, result)
                IF
                  result = spr.ok
```

```
                        -- close file again
                        SEQ
                          so.close(fs, ts, stream, result)
                          Assert(result=spr.ok, out.close)
                          going := FALSE
                      TRUE
                        -- some error occurred when opening the file
                        SEQ
                          so.write.string(fs, ts, "File ")
                          so.write.string(fs, ts,
                            [out.file FROM 0 FOR out.len])
                          so.write.string(fs, ts, " could not be opened.")
                          so.write.nl(fs, ts)
                          going := TRUE
                TRUE
                  -- some error occurred when reading the file name
                  going := TRUE
          -- set flag
          out := TRUE

:


------------------------------------------------------------------------
-- Load Program and Forward it to Network Connection
-- ATTENTION: References several global variables in In.Out!
------------------------------------------------------------------------
PROC Load.Prog()

  INT32 stream: -- stream id of program file
  BYTE  result: -- success of last file operation
  BOOL  going:  -- termination variable
  INT   len:    -- length of program cluster

  [Cluster.Size]INT32 cluster: -- program cluster

  ----------------------------------------------------------------------
  -- Body of Load.Prog
  ----------------------------------------------------------------------
  SEQ

    -- start status report
    Write.String("(")
    Write.String([prog.file FROM 0 FOR prog.len])

    -- open program file
    so.open(fs, ts, prog.file, spt.binary, spm.input, stream, result)
    Assert(result=spr.ok, prog.open)

    -- transmit program
    going := TRUE
    WHILE going
      SEQ

        -- read program cluster
```

```
        [Cluster.Size*4]BYTE clust RETYPES cluster:
        so.read(fs, ts, stream, len, clust)

        IF
          len < (SIZE cluster)
            SEQ
              -- eof must have occurred
              so.eof(fs, ts, stream, result)
              Assert(result=spr.ok,prog.read)
              going := FALSE
          TRUE
            SKIP

        -- forward program cluster
        to.net ! net.program; cluster

    -- close program file
    so.close(fs, ts, stream, result)
    Assert(result=spr.ok, prog.close)

    -- end status report
    Write.String(")")
:

---------------------------------------------------------------------------
-- Load Expression File to Network Connection
-- ATTENTION: References several global variables of Interface!
---------------------------------------------------------------------------
PROC Load.Expr()

  INT32 stream: -- stream id of expression file
  BYTE  result: -- success of last file operation
  BOOL  going:  -- termination variable
  INT   len:    -- length of expression cluster

  [Cluster.Size]INT32 cluster: -- expression cluster

  -------------------------------------------------------------------------
  -- Body of Load.Expr
  -------------------------------------------------------------------------
  SEQ

    -- start status report
    Write.String("(")
    Write.String([expr.file FROM 0 FOR expr.len])

    -- open expression file
    so.open(fs, ts, expr.file, spt.binary, spm.input, stream, result)
    Assert(result=spr.ok, expr.open)

    -- send net signal for begin of transmission
    to.net ! net.appendprog

    -- transmit program
    going := TRUE
```

```
      WHILE going
        SEQ

          -- read expression cluster
          [Cluster.Size*4]BYTE clust RETYPES cluster:
          so.read(fs, ts, stream, len, clust)

          IF
            len < (SIZE cluster)
              SEQ
                -- eof must have occurred
                so.eof(fs, ts, stream, result)
                Assert(result=spr.ok,expr.read)
                going := FALSE
            TRUE
              SKIP

          -- forward expression cluster
          to.net ! net.program; cluster

      -- close expression file
      so.close(fs, ts, stream, result)
      Assert(result=spr.ok, expr.close)

      -- end status report
      Write.String(")")

  :

  -------------------------------------------------------------------------
  -- Load Type Descriptor File to Result Generator
  --
  -- ATTENTION: References several global variables of Interface!
  -------------------------------------------------------------------------
  PROC Load.Type()

    INT32 stream: -- stream id of load descriptor file
    BYTE  result: -- success of last file operation
    INT   len:    -- length of last input line
    BOOL  going:  -- termination variable
    BOOL  okay:   -- success of type compilation

    [Type.Size]BYTE string: -- line of type descriptor

    -----------------------------------------------------------------------
    -- Body of Load.Type
    -----------------------------------------------------------------------
    SEQ

      -- start status report
      Write.String("(")
      Write.String([type.file FROM 0 FOR type.len])

      -- open type descriptor file
      so.open(fs, ts, type.file, spt.text, spm.input, stream, result)
```

```
    Assert(result=spr.ok, type.open)

    -- read type descriptor file
    going := TRUE
    WHILE going
      SEQ

        -- read one line of file
        so.gets(fs, ts, stream, len, string, result)

        -- line valid?
        going := result = spr.ok

        IF
          going
            -- forward cluster to result generator
            to.result ! is.type; len :: string
          TRUE
            SKIP

    -- end of file occurred?
    so.eof(fs, ts, stream, result)
    Assert(result=spr.ok, type.read)

    -- close type descriptor file
    so.close(fs, ts, stream, result)
    Assert(result=spr.ok, type.close)

    -- terminate transmission
    to.result ! end.type

    -- end status report
    Write.String(")")

    -- determine success of type compilation
    from.result ? CASE type.succ; okay
    Assert(okay, type.err)

:

---------------------------------------------------------------------------
-- Body of In.Out
---------------------------------------------------------------------------
SEQ

  -------------------------------------------------------------------------
  -- initializations
  -------------------------------------------------------------------------

  -- hello message
  so.write.string(fs, ts, "This is ADAM V")
  so.write.string(fs, ts, Version)
  so.write.string(fs, ts, " (")
  so.write.int(fs, ts, Mod.Num, 0)
  so.write.string(fs, ts, " modules used).")
```

```
so.write.nl(fs, ts)

-- read environment variable for directory specification
so.getenv(fs, ts, Dir.Var, dir.len, dir.name, result)
IF
  result = spr.ok
    IF
      (dir.len < (SIZE dir.name)) AND (dir.name[dir.len-1] <> '\')
        SEQ
          dir.name[dir.len] := '\'
          dir.len := dir.len+1
      TRUE
        SKIP
  TRUE
    dir.len := 0

-- get command line parameters
so.parse.command.line(fs, ts,
  Opt.Str, Opt.Par, opt.exist, opt.par, error, command)
Assert(error=0, [command FROM 0 FOR error])

-- fetch option flags
prog  := opt.exist[0] -- name of program file given
expr  := opt.exist[1] -- name of expression file given
out   := opt.exist[2] -- name of output file given
load  := opt.exist[3] -- program already loaded
batch := opt.exist[4] -- work in batch mode
stat  := opt.exist[5] -- print statistics

-- determine names of files
File.Names(prog, expr, out, prog.file, expr.file, out.file, type.file,
  prog.len, expr.len, out.len, type.len)

-- open output file for protocol of session
so.open(fs, ts, out.file, spt.text, spm.output, out.stream, result)
Assert(result=spr.ok, out.open)

-- write hello message to output file
so.fwrite.string(fs, ts, out.stream, "This is ADAM V", result)
so.fwrite.string(fs, ts, out.stream, Version, result)
so.fwrite.string(fs, ts, out.stream, " (", result)
so.fwrite.int(fs, ts, out.stream, Mod.Num, 0, result)
so.fwrite.string(fs, ts, out.stream, " processors used).", result)
so.fwrite.nl(fs, ts, out.stream, result)

--------------------------------------------------------------------------
-- main loop
--------------------------------------------------------------------------
going := TRUE
WHILE going
  SEQ

    ----------------------------------------------------------------------
    -- load files
    ----------------------------------------------------------------------
```

```
        -- send net signal for begin of transmission
        to.net ! net.startprog

        -- forward program to network connection
        IF
          load
            -- program is assumed to be loaded
            SKIP
          TRUE
            -- load program
            Load.Prog()

        -- forward expression file to network connection
        Load.Expr()

        -- signal end of transmission
        to.net ! net.endprog

        -- wait for acknowledge signal
        from.net ? CASE net.ack

        -- forward type descriptor file to result generator
        Load.Type()

        ---------------------------------------------------------------------
        -- receive results and statistics and write them to output file
        ---------------------------------------------------------------------

        -- open temporary file for statistics
        IF
          stat
            SEQ
              so.open.temp(fs, ts, spt.text, tmp.file, tmp.stream, result)
              Assert(result=spr.ok,tmp.open)
          TRUE
            SKIP

        -- make status report
        Write.Nl()
        Write.String("Program started.")
        Write.Nl()

        -- signal start of program execution
        to.result ! start

        -- remember time
        clock ? time0

        -- wait simultaneously for results and statistics and print them
        loop := TRUE
        WHILE loop
          ALT
            from.result ? CASE
              is.result; len :: string    -- write result
```

```
        Write.String([string FROM 0 FOR len])
      end.result                  -- end result
        SEQ
          clock ? time1
          Write.Nl()
          loop := FALSE
    from.stat ? len :: string     -- write statistics
      IF
        stat
          SEQ
            so.fwrite.string.nl(fs, ts, tmp.stream,
              [string FROM 0 FOR len], result)
            Assert(result=spr.ok,tmp.write)
        TRUE
          SKIP

-- report status and time
Write.String("Result received (")
time := (10*(time1-time0))/Ticks
INTTOSTRING(len, string, time/10)
Write.String([string FROM 0 FOR len])
Write.String(".")
INTTOSTRING(len, string, time REM 10)
Write.String([string FROM 0 FOR len])
Write.String(" s).")
Write.Nl()

-- wait for termination signal
from.net ? CASE net.ack
clock ? time2

-- report status and time
Write.String("Program terminated (")
time := (10*(time2-time0))/Ticks
INTTOSTRING(len, string, time/10)
Write.String([string FROM 0 FOR len])
Write.String(".")
INTTOSTRING(len, string, time REM 10)
Write.String([string FROM 0 FOR len])
Write.String(" s).")
Write.Nl()

----------------------------------------------------------------------
-- append statistics to output file
----------------------------------------------------------------------
IF
  stat
    SEQ

      -- close temporary file and open for reading
      so.close(fs, ts, tmp.stream, result)
      Assert(result=spr.ok, tmp.close)
      so.open(fs, ts, tmp.file, spt.text, spm.input, tmp.stream,
        result)
      Assert(result=spr.ok, tmp.open)
```

```
      -- read statistics from temporary file and write
      -- to output file (not to screen!)
      so.fwrite.string.nl(fs, ts, out.stream,
        "Program Statistics:", result)
      loop := TRUE
      WHILE loop
        SEQ
          so.read(fs, ts, tmp.stream, len, string)
          so.fwrite.string(fs, ts, out.stream,
            [string FROM 0 FOR len], result)
          loop := (len = (SIZE string))

      -- close temporary file and delete it
      so.close(fs, ts, tmp.stream, result)
      Assert(result=spr.ok, tmp.close)
      so.remove(fs, ts, tmp.file, result)
      Assert(result=spr.ok, tmp.delete)

  TRUE
    SKIP

----------------------------------------------------------------------
-- determine new file names
----------------------------------------------------------------------
IF
  batch

    -- terminate program if in batch mode
    going := FALSE

  TRUE
    SEQ

      -- Ask for another expression file
      loop := TRUE
      WHILE loop
        SEQ
          so.write.string(fs, ts, "Another Expression (y/n)? ")
          so.getkey(fs, ts, key, result)
          Assert(result=spr.ok, key.read)
          so.write.char(fs, ts, key)
          so.write.nl(fs, ts)
          IF
            is.lower(key)
              key := BYTE ((INT key) - 32)
            TRUE
              SKIP
          loop := (key <> 'Y') AND (key <> 'N')

      IF
        key = 'Y'
          -- reset expression flag
          expr := FALSE
        TRUE
```

```
                  SEQ
                    -- Ask for another program file
                    loop := TRUE
                    WHILE loop
                      SEQ
                        so.write.string(fs, ts, "Another Program (y/n)? ")
                        so.getkey(fs, ts, key, result)
                        Assert(result=spr.ok, key.read)
                        so.write.char(fs, ts, key)
                        so.write.nl(fs, ts)
                        IF
                          is.lower(key)
                            key := BYTE ((INT key) - 32)
                          TRUE
                            SKIP
                        loop := (key <> 'Y') AND (key <> 'N')
                    IF
                      key = 'Y'
                        -- reset program flag and expression flag
                        prog, expr := FALSE, FALSE
                      TRUE
                        -- exit program
                        going := FALSE

              -- determine new file names
              IF
                going
                  File.Names(prog, expr, out,
                    prog.file, expr.file, out.file, type.file,
                    prog.len, expr.len, out.len, type.len)
                TRUE
                  SKIP

    -- farewell message
    Write.String("Bye.")
    Write.Nl()

    -- close output file
    so.close(fs, ts, out.stream, result)
    Assert(result=spr.ok, out.close)

    -- terminate server
    so.exit(fs, ts, sps.success)

:

-----------------------------------------------------------------------------
-- End of ADINOUT.OCC
-----------------------------------------------------------------------------
```

```
---------------------------------------------------------------------------
--
-- ADSTAT.OCC                                         Wolfgang Schreiner
-- Statistics Generator                               (c) RISC-Linz 1990
--
-- Receives load messages from processors and memories and generates
-- printable statistics.
--
---------------------------------------------------------------------------

---------------------------------------------------------------------------
-- External References
---------------------------------------------------------------------------

-- include files
#INCLUDE "adglobal.inc" -- global constants
#INCLUDE "adnet.inc"    -- network constants and protocols
#INCLUDE "aderror.inc"  -- error messages
#INCLUDE "admaster.inc" -- master protocols and constants
#INCLUDE "adproc.inc"   -- processor unit protocols and constants
#INCLUDE "admemory.inc" -- memory unit protocols and constants
#INCLUDE "adstat.inc"   -- statistics constants
#INCLUDE "hostio.inc"   -- server constants and protocols

-- compilation units
#USE "convert.lib" -- type conversion facilities
#USE "string.lib"  -- string editing facilities
#USE "hostio.lib"  -- server interface

---------------------------------------------------------------------------
-- Statistics Generator
--
--   Receives load tokens from all processor and memory units, computes
--   the relative load of these units and converts them into printable form
--   (i.e. generates a little diagram). The procedure does not terminate.
--
--   This procedure is just a first prototype and will be extended to
--   generate more expressive statistics.
--
-- IN  from.net: tokens from net
-- OUT to.inout: output channel to input/output interface
-- IO  fs, ts:   input/output channel for error messages
---------------------------------------------------------------------------
PROC Adam.Statistics(CHAN OF Net.Link from.net, CHAN OF Stat to.inout,
  CHAN OF SP fs, ts)

  REAL32 rel.int.load: -- relative internal load
  REAL32 rel.ext.load: -- relative external load
  INT    len:          -- length of string
  BOOL   error:        -- error flag

  [Token.Size]INT32 token: -- network token
  [Stat.Len]BYTE    str:   -- statistics string

  -------------------------------------------------------------------------
```

```
-- Assert Condition
--
--   Checks condition and prints error message to screen,
--   if neccessary.
--
-- IN cond: test condition
-- IN mess: error message
--------------------------------------------------------------------------
PROC Assert(VAL BOOL cond, VAL []BYTE mess)

  ------------------------------------------------------------------------
  -- Body of Assert
  ------------------------------------------------------------------------
  SEQ
    IF
      cond
        SKIP
      TRUE
        SEQ
          so.write.string.nl(fs, ts, adam.assert)
          so.write.string.nl(fs, ts, mess)
          STOP
:

--------------------------------------------------------------------------
-- Body of Adam.Statistics
--------------------------------------------------------------------------
SEQ

  -- loop forever
  WHILE TRUE
    SEQ

      -- receive token
      from.net ? CASE net.token; token

      -- erase statistics line
      SEQ i = 0 FOR Stat.Len
        str[i] := ' '

      -- extract processor address
      [4]BYTE toks RETYPES token[2]:
      BYTE    addr IS toks[0]:
      INTTOSTRING(len, str, INT addr)

      -- extract pieces of information
      INT int.load RETYPES token[1]: -- internal load (frame or heap load)
      INT ext.load RETYPES token[0]: -- external load (req. or exp. queue)

      -- compute relative load
      [4]BYTE toks RETYPES token[2]:
      BYTE    type IS toks[1]:
      CASE type
        PLoad.PToken
          SEQ
```

```
       -- load of frame store and request queue
       rel.int.load := (REAL32 TRUNC int.load)/(REAL32 TRUNC Frame.Len
)
       rel.ext.load := (REAL32 TRUNC ext.load)/(REAL32 TRUNC Requ.Len)
       insert.string("P", len, str, 0, error)
       Assert(NOT error, str.cat)
     MLoad.MToken
       SEQ
       -- load of heap store and expensive request queue
       rel.int.load := (REAL32 TRUNC int.load)/(REAL32 TRUNC Heap.Len)
       rel.ext.load := (REAL32 TRUNC ext.load)/(REAL32 TRUNC Exp.Len)
       insert.string("M", len, str, 0, error)
       Assert(NOT error, str.cat)
     ELSE
      -- invalid token type
      Assert(FALSE, load.exp)

    -- insert relative load numbers
    len := Int.Pos
    append.real32(len, str, 100.0(REAL32) * rel.int.load, 3, 1)
    len := Ext.Pos
    append.real32(len, str, 100.0(REAL32) * rel.ext.load, 3, 1)

    -- show relative load by two columns of stars
    len := INT ROUND (rel.int.load * (REAL32 TRUNC Int.Len))
    SEQ i = Int.Start FOR len
      str[i] := '*'
    len := INT ROUND (rel.ext.load * (REAL32 TRUNC Ext.Len))
    SEQ i = Ext.Start FOR len
      str[i] := '*'

    -- send statistics to input/output interface
    to.inout ! (SIZE str) :: str
:

-----------------------------------------------------------------------------
-- End of ADSTAT.OCC
-----------------------------------------------------------------------------
```

ADMASTER.OCC Page 1

```
----------------------------------------------------------------------------
--
-- ADMASTER.OCC                                           Wolfgang Schreiner
-- Adam Master Module                                     (c) RISC-Linz 1990
--
-- Adam module serving as interface to the host.
--
----------------------------------------------------------------------------

----------------------------------------------------------------------------
-- External References
----------------------------------------------------------------------------

-- include files
#INCLUDE "adglobal.inc" -- global constants
#INCLUDE "adnet.inc"    -- network constants and protocols
#INCLUDE "admaster.inc" -- master protocols and constants
#INCLUDE "hostio.inc"   -- input/output protocols and constants

-- compilation units
#USE "adconn.t8h"  -- network connector
#USE "adgener.t8h" -- result generator
#USE "adstat.t8h"  -- statistics generator
#USE "adinout.t8h" -- input/output interface
#USE "hostio.lib"  -- server interface

----------------------------------------------------------------------------
-- Adam Master Module
--
--   Provides the interface to the host computer by performing several tasks:
--
--    * Loads basic program and distributes it over the network.
--    * Loads expression program and forwards it to first processor.
--    * Triggers start of execution (i.e. evaluation of expression).
--    * Requests result values, receives them and prints them.
--    * Receives load statistics, processes them and prints them.
--
-- IN  from.left:  input channel from left neighbour
-- OUT to.right:   output channel to right neighbour
-- IN  from.right: input channel from right neighbour
-- OUT to.left:    output channel to left neighbour
-- IO  fs, ts:     input/output channels from/to server
----------------------------------------------------------------------------
PROC Adam.Master(CHAN OF Net.Link from.left, to.right, from.right, to.left,
  CHAN OF SP from.server, to.server)

  -- internal channels
  CHAN OF Net.Link from.result: -- request tokens from result generator
  CHAN OF Net.Link to.result:   -- result tokens to result generator
  CHAN OF Net.Link to.stat:     -- load tokens to statistics generator
  CHAN OF Net.Link from.inout:  -- program from interface to connector
  CHAN OF Net.Link to.inout:    -- acknowledge signals to interface
  CHAN OF Result   result.in:   -- type description to result generator
  CHAN OF Result   result.out:  -- printable result to interface
  CHAN OF Stat     stat.out:    -- printable statistics to interface
```

```

  -- input/output channels
  [4]CHAN OF SP fs, ts:

  ---------------------------------------------------------------------------
  -- Body of Adam.Master
  ---------------------------------------------------------------------------
  PAR

    -- connection to network
    Adam.Connector(from.left, to.right, from.right, to.left,
      from.result, to.result, to.stat, from.inout, to.inout, fs[0], ts[0])

    -- result generator
    Adam.Generator(to.result, from.result, result.out, result.in,
      fs[1], ts[1])

    -- statistics generator
    Adam.Statistics(to.stat, stat.out, fs[2], ts[2])

    -- input/output interface
    Adam.In.Out(to.inout, from.inout, result.out, result.in, stat.out,
      fs[3], ts[3])

    -- input/output multiplexor
    CHAN OF BOOL stopper: -- not used
    so.multiplexor(from.server, to.server, ts, fs, stopper)

:
-----------------------------------------------------------------------------
-- End of ADMASTER.OCC
-----------------------------------------------------------------------------
```

ADSTAT.INC Page 1

```
-----------------------------------------------------------------------------
--
-- ADSTAT.INC                                             Wolfgang Schreiner
-- Constants for Statistics Generator                     (c) RISC-Linz 1990
--
-- Defines Several Constants for the Presentation of Runtime Statistics
-----------------------------------------------------------------------------

-- some parameters that control the format of the statics output
VAL INT Stat.Len  IS 80: -- length of a line of statistics
VAL INT Int.Pos   IS 3:  -- position for internal load number
VAL INT Int.Start IS 10: -- start of the column for the internal load
VAL INT Int.Len   IS 30: -- length of the column for the internal load
VAL INT Ext.Pos   IS 41: -- position for external load number
VAL INT Ext.Start IS 48: -- length of the column for the external load
VAL INT Ext.Len   IS 30: -- length of the column for the external load

-----------------------------------------------------------------------------
-- End of ADSTAT.INC
-----------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------
--
-- ADMASTER.INC                                          Wolfgang Schreiner
-- Adam Master Definitions                               (c) RISC-Linz 1990
--
-- Protocols and include files for Adam master module
--
----------------------------------------------------------------------------

----------------------------------------------------------------------------
-- Constants
----------------------------------------------------------------------------

-- version number
VAL []BYTE Version IS "1.0":

-- maximal sizes
VAL INT Cmd.Size    IS 140: -- maximal size of command line
VAL INT Name.Size   IS 80:  -- maximal size of file name (inclusive path)
VAL INT Ext.Size    IS 4:   -- size of ext. (with leading dot)
VAL INT Line.Size   IS 80:  -- maximal size of output line
VAL INT Type.Size   IS 80:  -- maximal size of type descriptor line
VAL INT CType.Size  IS 128: -- maximal size of compiled type description
VAL INT CLabel.Size IS 16:  -- maximal number of type labels (< End.Type!!!)
VAL INT Stack.Size  IS 16:  -- maximal number of nested types

-- command line options
VAL INT    Opt.Num   IS 6:   -- number of options allowed
VAL []BYTE Prog.Opt  IS " ": -- name of program file
VAL []BYTE Expr.Opt  IS "E": -- name of expression file
VAL []BYTE Out.Opt   IS "O": -- name of output file
VAL []BYTE Load.Opt  IS "L": -- program is already loaded
VAL []BYTE Batch.Opt IS "B": -- work in batch mode
VAL []BYTE Stat.Opt  IS "S": -- print statistics

-- null character
VAL BYTE Null IS 0(BYTE):

-- default extensions
VAL [Ext.Size]BYTE Prog.Ext IS ".ADM": -- program files
VAL [Ext.Size]BYTE Expr.Ext IS ".ADE": -- expression files
VAL [Ext.Size]BYTE Type.Ext IS ".ADT": -- type files
VAL [Ext.Size]BYTE Out.Ext  IS ".LIS": -- output files

-- name of environment variable for directory specification
VAL []BYTE Dir.Var IS "ADDIR":

-- default output file name (include trailing blank)
VAL []BYTE Out.File IS "ADAM.LIS ":

-- special characters in type descriptor file
VAL BYTE Comm.Chr  IS ';': -- comment type
VAL BYTE Root.Chr  IS '#': -- number type
VAL BYTE Label.Chr IS ':': -- label type
VAL BYTE Int.Chr   IS 'I': -- integer type
```

```
VAL BYTE Bool.Chr  IS 'B': -- boolean type
VAL BYTE Char.Chr  IS 'C': -- character type
VAL BYTE Str.Chr   IS 'S': -- string type
VAL BYTE List.Chr  IS 'L': -- list type
VAL BYTE Tuple.Chr IS 'T': -- tuple type
VAL BYTE Union.Chr IS 'U': -- union type

VAL BYTE End.Type IS 255(BYTE): -- signals end of type in compiled type

-- some parameters that control the format of the statics output
VAL INT Stat.Len  IS 80: -- length of a line of statistics
VAL INT Int.Start IS 4:  -- start of the column for the internal load
VAL INT Int.Len   IS 35: -- length of the column for the internal load
VAL INT Ext.Start IS 40: -- length of the column for the external load
VAL INT Ext.Len   IS 35: -- length of the column for the external load

----------------------------------------------------------------------------
-- Protocols
----------------------------------------------------------------------------

-- connection between interface and result generator
PROTOCOL Result
  CASE
    is.type; INT :: []BYTE   -- type descriptor line
    end.type                 -- end of type description
    type.succ; BOOL          -- type description successfully compiled
    start                    -- start signal
    is.result; INT :: []BYTE -- result string
    end.result               -- end of result signal
:

-- printable statistics string
PROTOCOL Stat IS INT :: []BYTE:

----------------------------------------------------------------------------
-- End of ADMASTER.INC
----------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------
--
-- ADARBIT.OCC                                           Wolfgang Schreiner
-- The Adam Arbiter                                      (c) RISC-Linz 1990
--
-- Interpreter for the arbiter process of the ADAM worker module
--
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- External References
-----------------------------------------------------------------------------

-- include files
#INCLUDE "adglobal.inc" -- global constants and protocols
#INCLUDE "adnet.inc"    -- network constants and protocols
#INCLUDE "adarbit.inc"  -- arbiter constants and protocols
#INCLUDE "aderror.inc"  -- error messages

-----------------------------------------------------------------------------
-- The Adam Arbiter
--
--   Routes the incoming program clusters and network tokens
--   to their correct destinations.
--
-- IN  act.mod:   address of actual module
-- IN  boot.from: number of link from that program is received
-- IN  boot.to:   numbers of links to that program has to be forwarded
-- IN  route:     link addresses of all modules in the network (incl. master)
-- IN  in:        input channels from neighbour modules
-- OUT out:       output channels to neighbour modules
-- IN  from.proc: input channel from local processor unit
-- OUT to.proc:   output channel to local processor unit
-- IN  from.mem:  input channel from local memory unit
-- OUT to.mem:    output channel to local memory unit
-- OUT debug:     debugging channel
-----------------------------------------------------------------------------
PROC Adam.Arbiter(VAL BYTE act.mod, VAL INT boot.from,
  VAL [Link.Num]BOOL boot.to, VAL [Mod.Num+1]INT route,
  [Link.Num]CHAN OF Net.Link in, out,
  CHAN OF Net.Link from.proc, CHAN OF Mod.Link to.proc,
  CHAN OF Net.Link from.mem, CHAN OF Mod.Link to.mem,
  CHAN OF Debug debug)

  INT  count:   -- number of modules in boot path (< Link.Num)

  [Link.Num]CHAN OF Permit   from.buff: -- channel for write permissions
  [Link.Num]CHAN OF Net.Link to.buff:   -- channel for buffer network tokens

  ---------------------------------------------------------------------------
  -- Assert Condition
  --
  --   Checks condition and forwards error message to debugging channel,
  --   if neccessary.
  --
```

```
-- IN cond: test condition
-- IN mess: error message
----------------------------------------------------------------------------
PROC Assert(VAL BOOL cond, VAL []BYTE mess)

  --------------------------------------------------------------------------
  -- Body of Assert
  --------------------------------------------------------------------------
  SEQ
    IF
      cond
        SKIP
      TRUE
        SEQ
          debug ! (SIZE mess) :: mess
          STOP
:

----------------------------------------------------------------------------
-- Load Program
--
--   Receives program from link specified by 'boot.from' and
--   forwards it to local processor and to all neighbour modules
--   specified by 'boot.to'.
--
-- IN  boot.from: number of link from that program is to be received
-- IN  boot.to:   boot.to[i]=TRUE iff program is to be sent to link i
-- IN  in:        input channels
-- IN  permit:    channels by that write permission are received
-- OUT out:       output channels
----------------------------------------------------------------------------
PROC Load.Prog(VAL INT boot.from, VAL [Link.Num]BOOL boot.to,
  [Link.Num]CHAN OF Net.Link in, [Link.Num]CHAN OF Permit permit,
  [Link.Num]CHAN OF Net.Link out)

  BOOL going: -- termination variable

  [Cluster.Size]INT32 program: -- program cluster

  --------------------------------------------------------------------------
  -- Body of Load.Prog
  --------------------------------------------------------------------------
  SEQ

    -- load program and forward it
    going := TRUE
    WHILE going

      in[boot.from] ? CASE

        -- new program signal
        net.startprog
          SEQ
            to.proc ! mod.startprog
            SEQ i = 0 FOR Link.Num
```

```
              IF
                boot.to[i]
                  SEQ
                    out[i] ! net.startprog
                    permit[i] ? CASE write.permit
                TRUE
                  SKIP

        -- append program signal
        net.appendprog
          SEQ
            to.proc ! mod.appendprog
            SEQ i = 0 FOR Link.Num
              IF
                boot.to[i]
                  SEQ
                    out[i] ! net.appendprog
                    permit[i] ? CASE write.permit
                TRUE
                  SKIP

        -- program cluster
        net.program; program
          SEQ
            to.proc ! mod.program; program
            SEQ i = 0 FOR Link.Num
              IF
                boot.to[i]
                  SEQ
                    out[i] ! net.program; program
                    permit[i] ? CASE write.permit
                TRUE
                  SKIP

        -- end of program signal
        net.endprog
          SEQ
            going := FALSE
            to.proc ! mod.endprog
            SEQ i = 0 FOR Link.Num
              IF
                boot.to[i]
                  SEQ
                    out[i] ! net.endprog
                    permit[i] ? CASE write.permit
                TRUE
                  SKIP

:

----------------------------------------------------------------------------
-- Receive Acknowledge Signals
--
--   Waits for acknowledge signals from all links to that the program
--   has been forwarded and then send own acknowledge to boot link.
```

```
--
-- IN  count:    number of expected acknowledge signals from links
-- IN  ack.to:   number of link to that acknowledge signal is to be sent
-- IN  ack.from: ack.from[i] = TRUE iff signal is expected from link i
-- IN  in:       input channels
-- IN  permit:   channels by that write permission are received
-- OUT out:      output channels
---------------------------------------------------------------------------
PROC Acknowledge(VAL INT count, VAL INT ack.to, VAL [Link.Num]BOOL ack.from,
  [Link.Num]CHAN OF Net.Link in, [Link.Num]CHAN OF Permit permit,
  [Link.Num]CHAN OF Net.Link out)

  [Link.Num]BOOL    wait:  -- on which links signal is still missing?
  [Token.Size]INT32 token: -- network token

  BOOL wait.proc: -- signal is missing from processor unit
  BOOL wait.mem:  -- signal is missing from memory unit
  INT n:          -- number of signals still missing

  -------------------------------------------------------------------------
  -- Body of Acknowledge
  -------------------------------------------------------------------------
  SEQ

    -- all signals are missing
    wait := ack.from
    wait.proc := TRUE
    wait.mem := TRUE
    n := count+2

    -- wait for acknowledge signals and skip all tokens received
    WHILE n > 0
      ALT

        -- acknowledge from processor unit
        wait.proc & from.proc ? CASE
          net.ack
            SEQ
              wait.proc := FALSE
              n := n-1
          net.token; token
            SKIP

        -- acknowledge from memory unit
        wait.mem & from.mem ? CASE
          net.ack
            SEQ
              wait.mem := FALSE
              n := n-1
          net.token; token
            SKIP

        -- acknowledge from neighbour modules
        ALT i = 0 FOR Link.Num
          wait[i] & in[i] ? CASE
```

```
            net.ack
              SEQ
                wait[i] := FALSE
                n := n-1
            net.token; token
              SKIP

    -- send own acknowledge
    out[ack.to] ! net.ack
    permit[ack.to] ? CASE write.permit

:

---------------------------------------------------------------------------
-- Transmit tokens
--
--   Receives tokens and stores them in an internal buffer until
--   token may be forwarded. If a stop token is received, this token
--   is forwarded to all links in the boot path. Afer the acknowledge
--   signals from all of these links have arrived, an acknowledge signal
--   is sent to the boot link.
--
-- IN  ack.to:   link to that acknowledges are to be sent
-- IN  stop.to:  stop.to[i] = TRUE iff stop token is to be sent to link i
-- IN  route:    route[i] is link number of tokens destined for module i
-- IN  in:       input channels
-- IN  permit:   channels by that write permission are received
-- OUT out:      output channels
---------------------------------------------------------------------------
PROC Transmit(VAL INT ack.to,
  VAL [Link.Num]BOOL stop.to, VAL [Mod.Num+1]INT route,
  [Link.Num]CHAN OF Net.Link in, [Link.Num]CHAN OF Permit permit,
  [Link.Num]CHAN OF Net.Link out)

  BOOL going:   -- termination variable
  BOOL waiting: -- waiting for arrival of new token
  BOOL proc:    -- token destined for local processor unit
  INT  free:    -- list of free buffer elements
  INT  j:       -- auxiliary pointer
  BYTE mod:     -- module address of token
  BYTE type:    -- type of token
  INT  link:    -- link number to send token through
  INT  new:     -- reference to new token space

  [Link.Num]BOOL wait:  -- waiting for send permission on link
  [Link.Num]INT  first: -- first tokens in waiting queues
  [Link.Num]INT  last:  -- last tokens in waiting queues

  [Arbit.Size*Token.Size]INT32 buffer: -- token buffer
  [Arbit.Size]INT              next:   -- link to next token
  [Token.Size]INT32            token:  -- network token

  -------------------------------------------------------------------------
  -- Body of Transmit
  -------------------------------------------------------------------------
```

```
    SEQ

      ----------------------------------------------------------------------
      -- initializations
      ----------------------------------------------------------------------

      -- link all buffer tokens into free list
      free := 0
      SEQ i = 0 FOR Arbit.Size
        next[i] := i+1
      next[Arbit.Size-1] := -1

      -- initalize waiting queues
      SEQ i = 0 FOR Link.Num
        SEQ
          wait[i] := FALSE
          first[i] := -1
          last[i] := -1

      ----------------------------------------------------------------------
      -- transmit tokens until stop token received
      ----------------------------------------------------------------------
      going := TRUE
      WHILE going
        SEQ

          ------------------------------------------------------------------
          -- wait for arrival of new token
          ------------------------------------------------------------------
          waiting := TRUE
          WHILE waiting
            ALT

              -- receive send permission signal from some output channel
              ALT i = 0 FOR Link.Num
                wait[i] & permit[i] ? CASE write.permit
                  IF
                    first[i] = (-1)
                      -- no token for that link in buffer
                      wait[i] := FALSE
                    TRUE
                      SEQ
                        -- send token through that link
                        out[i] ! net.token;
                          [buffer FROM first[i]*Token.Size FOR Token.Size]
                        -- unlink token space from linked list
                        j := first[i]
                        first[i] := next[j]
                        IF
                          first[i] = (-1)
                            last[i] := -1
                          TRUE
                            SKIP
                        -- collect buffer space in free list
                        next[j] := free
```

```
              free := j

    -- receive token from processor unit
    from.proc ? CASE net.token; token
      waiting := FALSE

    -- receive token from memory unit
    from.mem ? CASE net.token; token
      waiting := FALSE

    -- receive token from some neighbour module
    ALT i = 0 FOR Link.Num
      in[i] ? CASE
        net.token; token
          waiting := FALSE

-------------------------------------------------------------------
-- new token received
-------------------------------------------------------------------

-- extract destination address and type
[4]BYTE toks RETYPES token[2]:
mod, type  := toks[2], toks[1]

IF

  -----------------------------------------------------------------
  -- stop token encountered
  -----------------------------------------------------------------
  type = Stop.Token
    going := FALSE

  ---------------------------------------------------------------
  -- token destined for actual module
  ---------------------------------------------------------------
  mod = act.mod
    SEQ
      -- determine destination type of token
      proc  := ((INT16 type) /\ #E0(INT16)) = 0(INT16)
      IF
        proc
          -- token destined for local processor
          to.proc ! mod.token; token
        TRUE
          -- token destined for local memory
          to.mem ! mod.token; token

  -----------------------------------------------------------------
  -- token destined for another module
  -----------------------------------------------------------------
  TRUE
    SEQ
      -- lookup link address in routing table
      link := route[INT mod]
      IF
```

```
            -- arbiter has not send permission on that link
            wait[link]
              SEQ
                -- unlink buffer space from free list
                Assert(free>(-1), arbit.full)
                new := free
                free := next[free]
                -- store token in buffer
                [buffer FROM new*Token.Size FOR Token.Size] := token
                -- link token space into waiting queue
                next[new] := -1
                IF
                  last[link] = (-1)
                    SKIP
                  TRUE
                    next[last[link]] := new
                last[link] := new
                IF
                  first[link] = (-1)
                    first[link] := new
                  TRUE
                    SKIP
            -- arbiter has send permission on that link
            TRUE
              SEQ
                -- forward token through link
                out[link] ! net.token; token
                -- send permission lost
                wait[link] := TRUE

------------------------------------------------------------------------
-- stop token encountered
------------------------------------------------------------------------

-- forward stop token to local units
to.proc ! mod.token; token
to.mem ! mod.token; token

-- forward stop token to all modules in boot path
SEQ i = 0 FOR Link.Num
  IF
    stop.to[i]
      SEQ
        IF
          wait[i]
            permit[i] ? CASE write.permit
          TRUE
            SKIP
        out[i] ! net.token; token
        permit[i] ? CASE write.permit
    TRUE
      SKIP

-- get write permission on acknowledge link
IF
```

```
      wait[ack.to]
        permit[ack.to] ? CASE write.permit
      TRUE
        SKIP

:

----------------------------------------------------------------------------
-- Body of Adam.Arbiter
----------------------------------------------------------------------------
SEQ

  --------------------------------------------------------------------------
  -- initializations
  --------------------------------------------------------------------------

  -- program must not be forwarded to link from that it has been received
  IF
    boot.to[boot.from]
      SEQ
        debug ! (SIZE boot.err) :: boot.err
        STOP
    TRUE
      SKIP

  -- count number of links through that program is to be forwarded
  count := 0
  SEQ i = 0 FOR Link.Num
    IF
      boot.to[i]
        count := count + 1
      TRUE
        SKIP

  --------------------------------------------------------------------------
  -- let arbiter nucleus and link buffers work in parallel
  --------------------------------------------------------------------------
  PAR

    ------------------------------------------------------------------------
    -- nucleus of arbiter
    ------------------------------------------------------------------------
    WHILE TRUE
      SEQ
        -- boot program to network
        Load.Prog(boot.from, boot.to, in, from.buff, to.buff)
        -- receive acknowledge signals and send own to boot link
        Acknowledge(count, boot.from, boot.to, in, from.buff, to.buff)
        -- transmit tokens until stop signal received
        Transmit(boot.from, boot.to, route, in, from.buff, to.buff)
        -- receive acknowledge signals and send own to boot link
        Acknowledge(count, boot.from, boot.to, in, from.buff, to.buff)

    ------------------------------------------------------------------------
    -- link buffers
```

```
      -----------------------------------------------------------------------
      PAR i=0 FOR Link.Num
        [Token.Size]INT32   token:   -- network token
        [Cluster.Size]INT32 program: -- program cluster
        WHILE TRUE
          SEQ
            to.buff[i] ? CASE
              net.token; token
                out[i] ! net.token; token
              net.startprog
                out[i] ! net.startprog
              net.appendprog
                out[i] ! net.appendprog
              net.program; program
                out[i] ! net.program; program
              net.endprog
                out[i] ! net.endprog
              net.ack
                out[i] ! net.ack
            from.buff[i] ! write.permit

:
-----------------------------------------------------------------------------
-- End of ADARBIT.OCC
-----------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------
--
-- ADMEMORY.OCC                                          Wolfgang Schreiner
-- The Adam Memory Unit                                  (c) RISC-Linz 1990
--
-- Interpreter for all components of the Adam memory unit.
--
----------------------------------------------------------------------------

----------------------------------------------------------------------------
-- External References
----------------------------------------------------------------------------

-- include files
#INCLUDE "adglobal.inc" -- global constants
#INCLUDE "adnet.inc"    -- network constants
#INCLUDE "aderror.inc"  -- error messages
#INCLUDE "admemory.inc" -- memory unit constants

----------------------------------------------------------------------------
-- The Adam Memory Unit
--
--   The memory unit is responsible for the management of all non-atomic,
--   i.e. compound, data. The basic data structure is the 'tuple' i.e.
--   a data record of fixed size where each field of the tuple may contain
--   an (integer) atom or a reference to another tuple. Processor units
--   control creation of and access to tuples via 4 kinds of requests:
--
--     1. Allocation requests ask the memory unit to return a reference
--        to a new tuple of certain size with all tuple fields being empty.
--
--     2. Read requests ask the memory unit to return the component of some
--        particular tuple field. If that field has not yet been filled, the
--        request is deferred.
--
--     3. Write requests ask the memory unit to fill the components of some
--        tuple field with a particular value. If that field has already
--        been written, a runtime error is triggered. Otherwise, the value
--        is written, and all deferred read requests are resolved.
--
--     4. Submission requests ask the memory unit to submit the lambda-lifted
--        arguments of a higher order function stored in a tuple to a
--        particular activation of that function.
--
--   For reasons of program simplicity, tuples are internally represented by
--   linked lists of fixed size cells.
--
-- IN  act.mod:   actual module address
-- IN  mod.addr:  addresses of all neighbour modules
-- IN  link.addr: link addresses of actual module from view of neighbour
-- IN  in:        input channel from arbiter
-- OUT out:       output channel to arbiter
-- OUT debug:     debug channel to output process
----------------------------------------------------------------------------
PROC Adam.Memory(VAL BYTE act.mod,
```

```
VAL [Link.Num]BYTE mod.addr, VAL [Link.Num]BYTE link.addr,
CHAN OF Mod.Link in, CHAN OF Net.Link out, CHAN OF Debug debug)

-- internal channels
CHAN OF Memory.Token to.exp, to.cheap:     -- transmit tokens to buffers
CHAN OF Buffer.Token from.exp, from.cheap: -- transmit tokens to manager
CHAN OF Demand demand.exp, demand.cheap:   -- transmit demands

-- buffers for expensive and cheap store
[Token.Size*Exp.Len]INT32 exp.store:     -- expensive store
[Token.Size*Cheap.Len]INT32 cheap.store: -- cheap store

[4]CHAN OF Debug error: -- internal debugging channels

--------------------------------------------------------------------------
-- The Memory Unit Router:
--
--   The router receives incoming data from the arbiter and routes them
--   as follows:
--
--     1. Allocation, read and submission requests to the expensive store.
--     2. Write requests and memory load reports to the cheap store.
--
--   IN  in:       channel from arbiter
--   OUT to.exp:   channel to expensive store
--   OUT to.cheap: channel to cheap store
--   OUT debug:    debugging channel
--------------------------------------------------------------------------
PROC Router(CHAN OF Mod.Link in,
  CHAN OF Memory.Token to.exp, CHAN OF Memory.Token to.cheap,
  CHAN OF Debug debug)

  BYTE tag: -- actual token tag

  [Token.Size]INT32 token: -- network token

  ------------------------------------------------------------------------
  -- Body of Router
  ------------------------------------------------------------------------
  SEQ

    WHILE TRUE
      SEQ
        -- receive token and extract tag
        in ? CASE mod.token; token
        [4]BYTE toks RETYPES token[2]:
        tag := toks[1]
        -- forward token to appropriate store
        CASE tag
          Stop.Token                 -- stop token
            SEQ
              to.cheap ! token
              to.exp ! token
          Alloc.MToken, Read.MToken, Submit.MToken
            to.exp ! token           -- expensive requests
```

```
          ELSE
            to.cheap ! token         -- cheap requests

    -- never done (just to avoid compiler warnings)
    debug ! 0 :: ""

:

------------------------------------------------------------------------
-- The Memory Unit Buffer
--
--   Buffers incoming tokens in a queue and forwards them to the
--   memory manager on demand.
--
--   IN  in:     input channel from router
--   IN  demand: demand channel from memory manager
--   IN  buffer: store to hold tokens
--   OUT out:    output channel to memory unit
--   OUT debug:  debugging channel
------------------------------------------------------------------------
PROC Buffer(CHAN OF Memory.Token in, CHAN OF Demand demand, []INT32 buffer,
  CHAN OF Buffer.Token out, CHAN OF Debug debug)

  VAL buffer.size IS SIZE buffer:

  INT  first, last: -- indizes of the first and last queue elements
  INT  num:         -- number of elements in queue
  BOOL full, empty: -- denote the actual status of the queue
  BOOL going:       -- termination flag
  BYTE tag:         -- tag of actual token

  [Token.Size]INT32 token: -- token value

  ----------------------------------------------------------------------
  -- Assert Condition
  --
  --   Checks condition and forwards error message to debugging channel,
  --   if neccessary.
  --
  -- IN cond: test condition
  -- IN mess: error message
  ----------------------------------------------------------------------
  PROC Assert(VAL BOOL cond, VAL []BYTE mess)
    SEQ
      IF
        cond
          SKIP
        TRUE
          SEQ
            debug ! (SIZE mess) :: mess
            STOP
  :

  -----------------------------------------------------------------------
  -- Body of Buffer
```

```
-------------------------------------------------------------------------
WHILE TRUE
  SEQ

    -- buffer length must be a multiple of Token.Size
    Assert((buffer.size REM Token.Size) = 0, buff.err)

    -- initialize status variables
    num, first, last := 0, 0, 0
    full, empty := FALSE, TRUE

    -- loop until stop token encountered
    going := TRUE
    WHILE going

      -- full equals not empty
      -- if first <> last
      --   then first refers to the next free element
      --        last refers to the last stored element
      --   else if full
      --          then first is not valid
      --          else last is not valid

      -- listen on both input channels
      ALT

        -- demand received
        demand ? CASE demand.tag
          SEQ

            -- submit token
            [Token.Size]INT32 token IS [buffer FROM last FOR Token.Size]:
            out ! num; token

            -- extract tag
            [Token.Size]INT32 token IS [buffer FROM last FOR Token.Size]:
            [4]BYTE toks RETYPES token[2]:
            tag := toks[1]

            IF
              tag = Stop.Token  -- stop token submitted, empty buffer
                going := FALSE
              empty             -- submitted token was undefined
                SKIP
              TRUE              -- update state variables
                SEQ
                  num  := num-1
                  last := (last + Token.Size) REM buffer.size
                  full := FALSE
                  empty := first = last

        -- token received
        in ? token
          SEQ

```

```
              -- buffer must not be full
              Assert(NOT full, buff.full)

              -- store token
              [buffer FROM first FOR Token.Size] := token
              num   := num+1
              first := (first + Token.Size) REM buffer.size
              empty := FALSE
              full  := first = last

:

--------------------------------------------------------------------------
-- The Memory Manager
--
--   Manages the heap store and handles incoming requests.
--
-- IN  from.exp:     input channel from expensive store
-- IN  from.cheap:   input channel from cheap store
-- OUT demand.exp:   demand channel to expensive store
-- OUT demand.cheap: demand channel to cheap store
-- OUT error:        error channel for debugging purposes
--------------------------------------------------------------------------
PROC Manager(CHAN OF Buffer.Token from.exp, from.cheap,
  CHAN OF Demand demand.exp, demand.cheap, CHAN OF Debug debug)

  -- the basic stores
  [Heap.Len]INT16 tag:    -- tag fields of heap cells
  [Heap.Len]INT32 head:   -- head fields of heap cells
  [Heap.Len]INT32 tail:   -- tail fields of heap cells
  [Stack.Len]INT32 stack: -- deallocation stack

  -- the load parameters
  INT heap.load:  -- load of heap store
  INT exp.load:   -- load of expensive store
  INT cheap.load: -- load of cheap store
  INT mLoad:      -- relative load of actual memory
  INT lLoad:      -- last load reported to neighbours

  -- global state variables
  INT free: -- head of free list
  INT top:  -- top of deallocation stack

  -- network descriptors
  [Link.Num]INT mem.load: -- relative load of neighbour memories

  -- auxiliary variables
  INT   action:  -- next action to be executed
  BOOL  going:   -- termination flag

  ------------------------------------------------------------------------
  -- Assert Condition
  --
  --   Checks condition and forwards error message to debugging channel,
  --   if neccessary.
```

```
--
-- IN cond: test condition
-- IN mess: error message
-----------------------------------------------------------------------
PROC Assert(VAL BOOL cond, VAL []BYTE mess)

  ---------------------------------------------------------------------
  -- Body of Assert
  ---------------------------------------------------------------------
  SEQ
    IF
      cond
        SKIP
      TRUE
        SEQ
          debug ! (SIZE mess) :: mess
          STOP
:

------------------------------------------------------------------------
-- Minimal Loaded Neighbour
--
--   Returns the address of that module among the neighbours of the
--   actual module whose memory unit is probably least loaded. If all
--   neighbours are probably more loaded than the actual module, the
--   actual module address is returned.
--
------------------------------------------------------------------------
BYTE FUNCTION Min.Loaded()

  BYTE addr: -- address of minimal loaded neighbour found up to now
  INT  load: -- and its load

  ----------------------------------------------------------------------
  -- Body of Min.Loaded
  ----------------------------------------------------------------------
  VALOF
    SEQ
      addr, load := act.mod, mLoad  -- actual module is default
      SEQ i = 0 FOR Link.Num        -- search for less loaded modules
        VAL BYTE a IS mod.addr[i]:
        VAL INT  l IS mem.load[i]:
        IF
          l < load
            addr, load := a, l
          TRUE
            SKIP
    RESULT addr
:

------------------------------------------------------------------------
-- Compute Relative Load
--
--   Takes heap load and expensive store load and computes from these
--   values the relative memoire load, i.e. a value between 0 and 100
```

```
--   that (hopefully adequately) describes the overall working load of
--   the memory: a value of 0 should denote "the memory unit is not at
--   all occupied" and a value of 100 should denote "the memory unit is
--   not able to fulfill any further requests". Any value between should
--   in some way express a rough measure for the load of the memory.
--
-- IN heap: the heap load of the memory unit
-- IN exp:  the expensive store load of the memory unit
------------------------------------------------------------------------
INT FUNCTION Load(VAL INT heap, exp)

  ----------------------------------------------------------------------
  -- Body of Load
  ----------------------------------------------------------------------
  VALOF
    SKIP
    RESULT ((100 * heap) / Heap.Len) + (0 * exp)
:

------------------------------------------------------------------------
-- Determine Next Action
--
--   Determines the type of the next action to be executed.
--
-- IO action: type of action to be executed next
------------------------------------------------------------------------
PROC Mem.Action(INT action)

  ----------------------------------------------------------------------
  -- Body of Mem.Action
  ----------------------------------------------------------------------
  IF
    -- if load has significantly changed, report to neighbours
    (mLoad > Mem.Trash) AND
      (((100 * lLoad) < (Mem.Delay * mLoad)) OR
       ((100 * mLoad) < (Mem.Delay * lLoad)))
      action := Report.Act
    -- expensive store empty, give it a chance for the next time
    exp.load = 0
      SEQ
        exp.load := 1
        action := Cheap.Act
    -- expensive load too high and heap load not too high
    (exp.load > Exp.Bound) AND (heap.load < Heap.Bound)
      action := Exp.Act
    -- heap load too low
    heap.load  < Heap.Trash
      action := Exp.Act
    -- cheap store empty, give it a chance for the next time
    cheap.load = 0
      SEQ
        cheap.load := 1
        action := Exp.Act
    -- cheap requests are always possible
    TRUE
```

```
      action := Cheap.Act

:
------------------------------------------------------------------------
-- Check if Tuple Available on Heap
--
--   Returns TRUE iff a tuple with the specified number of fields can be
--   allocated on the heap.
--
-- IN num: number of heap cells to be allocated
------------------------------------------------------------------------
BOOL FUNCTION Avail.Tuple(VAL INT num)

  ----------------------------------------------------------------------
  -- Body of Avail.Tuple
  ----------------------------------------------------------------------
  VALOF
    SKIP
    RESULT num <= (Heap.Len-heap.load)
:

------------------------------------------------------------------------
-- Allocate Tuple
--
--   Allocates a tuple with the specified number of fields by
--   linking the required number of heap cells together. The
--   reference count field of the tuple is set to the maximal
--   reference weight, the 'available' bits of the tuple fields are
--   reset to FALSE, the heads of the associated deferred read lists
--   are reset to NIL. If the tuple cannot be allocated, the program
--   is terminated with an error message.
--
-- IN  num: number of tuple fields required
-- OUT new: address of new tuple allocated
------------------------------------------------------------------------
PROC Alloc.Tuple(VAL INT num, INT new)

  INT n:    -- auxiliary counter
  INT next: -- auxiliary pointer

  ----------------------------------------------------------------------
  -- Body of Alloc.Tuple
  ----------------------------------------------------------------------
  SEQ

    -- check if field number valid and required number of cells available
    Assert((num > 1) AND (num <= (Heap.Len-heap.load)), alloc.err)

    new := free                    -- address of new tuple

    -- link open cells together
    n := num
    WHILE n > 2
      SEQ
        tag[free] := #0810(INT16)  -- reference count 8, cell is open
```

```
          head[free] := 0(INT32)     -- deferred read list empty
          free := INT tail[free]     -- unlink cell from free list
          n := n-1                   -- one field less left

      next := INT tail[free]         -- next free heap cell

      tag[free] := #0800(INT16)      -- reference count 8, cell is closed
      head[free] := 0(INT32)         -- deferred read list head empty
      tail[free] := 0(INT32)         -- deferred read list tail empty

      free := next                   -- new head of free list
      heap.load := heap.load+(num-1) -- new heap load

  :

  ------------------------------------------------------------------------
  -- Deallocate Tuple
  --
  --   Deallocates specified tuple and relinks its cells into the free
  --   list. If not all tuple field have yet been filled, the program
  --   is terminated with an error message. The addresses of all tuples
  --   that were referenced by the deallocated tuple are put on the
  --   deallocation stack (if they reside in the actual memory) or
  --   decrease reference messages are sent to these tuples (if they
  --   reside on other memory units).
  --
  -- IN addr: address of tuple to be deallocated
  ------------------------------------------------------------------------
  PROC Dealloc.Tuple(VAL INT addr)

    BOOL open:   -- last tuple cell open?
    BOOL open0:  -- actual tuple cell open?
    BOOL avHd:   -- head field available?
    BOOL avTl:   -- tail field available?
    BOOL ptrHd:  -- head field pointer?
    BOOL ptrTl:  -- tail field pointer?
    BYTE mod:    -- module address
    INT  wght:   -- weight field
    INT  weight: -- reference weight
    INT  addr0:  -- actual cell address
    BOOL error:  -- error flag

    [Token.Size]INT32 token: -- network token

    ----------------------------------------------------------------------
    -- Body of Dealloc.Tuple
    ----------------------------------------------------------------------
    SEQ

      -- check tuple address
      Assert(addr > 0, dealloc.err)

      -- extract tuple tag bits
      INT16 tags IS tag[addr]:
      SEQ
```

```
  open0 := (tags /\ #0010(INT16)) > 0(INT16)  -- cell open?
  avHd  := (tags /\ #0008(INT16)) > 0(INT16)  -- head available?
  ptrHd := (tags /\ #0004(INT16)) > 0(INT16)  -- head pointer?

-- save address
addr0 := addr

---------------------------------------------------------------------
-- traverse head fields of tuple cells
---------------------------------------------------------------------

-- enter loop at least one time
open := TRUE
WHILE open
  SEQ

    -- check that head available
    Assert(avHd, dealloc.err)

    -- actual pointer
    INT32 ptr IS head[addr0]:

    -- field contains reference to another tuple
    IF
      ptrHd
        SEQ

          -- error flag of actual pointer
          error := BOOL (ptr /\ #00000001(INT32))

          IF
            error
              SKIP

            TRUE
              SEQ

                -- extract module address
                [4]BYTE p RETYPES ptr:
                mod := p[3]

                IF -- tuple on other module or stack full
                  (mod <> act.mod) OR (top = Stack.Len)
                    SEQ

                      -- construct decrease reference message
                      [4]BYTE t RETYPES token[2]:
                      SEQ
                        t[2] := mod
                        t[1] := DecRef.MToken
                      token[0] := ptr

                      -- send token into network
                      out ! net.token; token

```

```
                  TRUE -- tuple on actual module and stack not full
                    SEQ

                      -- push reference on stack
                      INT32 next IS stack[top]:
                      SEQ
                        next := ptr
                        [4]BYTE n RETYPES next:
                        n[3] := 0(BYTE)       -- delete mod address
                        next := next >> 1     -- delete error flag
                        wght :=               -- weight field
                          INT (next /\ #00000003(INT32))
                        next := next >> 2     -- tuple address
                        weight := 1           -- reference weight
                        SEQ i = 1 FOR wght
                          weight := weight * 2
                        [4]BYTE n RETYPES next:
                        n[3] := BYTE weight

                      -- increase stack pointer
                      top := top+1

      -- field contains atomic (integer value)
      TRUE
        SKIP

    heap.load := heap.load-1 -- update load
    open := open0            -- actual tuple closed?

    -- extract tuple tags of next tuple field
    IF
      open
        SEQ
          tag[addr0] := 0(INT16)
          addr0 := INT tail[addr0]
          INT16 t IS tag[addr0]:
          SEQ
            open0 := (t /\ #0010(INT16)) > 0(INT16) -- cell open?
            avHd  := (t /\ #0008(INT16)) > 0(INT16) -- head available?
            ptrHd := (t /\ #0004(INT16)) > 0(INT16) -- head pointer?
      TRUE
        SKIP

---------------------------------------------------------------------
-- handle tail field of tuple cell
---------------------------------------------------------------------

-- extract tuple tags
INT16 t IS tag[addr0]:
SEQ
  avTl  := (t /\ #0002(INT16)) > 0(INT16)  -- tail available?
  ptrTl := (t /\ #0001(INT16)) > 0(INT16)  -- tail pointer?

-- set reference count to 0
tag[addr0] := 0(INT16)
```

```

      -- check that tail available
      Assert(avTl, dealloc.err)

      -- actual pointer
      INT32 ptr IS tail[addr0]:

      IF
        ptrTl
          SEQ

            -- actual error bit
            error := BOOL (ptr /\ #00000001(INT32))

            IF
              error
                SKIP

              TRUE
                SEQ

                  -- extract module address
                  [4]BYTE p RETYPES ptr:
                  mod := p[3]

                  IF  -- tuple on other module or stack full
                    (mod <> act.mod) OR (top = Stack.Len)
                      SEQ

                        -- construct decrease reference message
                        [4]BYTE t RETYPES token[2]:
                        SEQ
                          t[2] := mod
                          t[1] := DecRef.MToken
                        token[0] := ptr

                        -- send token into network
                        out ! net.token; token

                    TRUE -- tuple on actual module and stack not full
                      SEQ

                        -- push reference on stack
                        INT32 next IS stack[top]:
                        SEQ
                          next := ptr
                          [4]BYTE n RETYPES next:
                          n[3] := 0(BYTE)          -- delete mod address
                          next := next >> 1        -- delete error flag
                          wght :=                  -- weight field
                            INT (next /\ #00000003(INT32))
                          next := next >> 2        -- tuple address
                          weight := 1              -- reference weight
                          SEQ i = 1 FOR wght
                            weight := weight * 2
```

```
                        [4]BYTE n RETYPES next:
                        n[3] := BYTE weight

                      -- increase stack pointer
                      top := top+1

      TRUE
        SKIP

    ---------------------------------------------------------------------
    -- link tuple cells to free list
    ---------------------------------------------------------------------

    tail[addr0] := INT32 free
    free := addr

:

-------------------------------------------------------------------------
-- Garbage Collection
--
--   Handle garbage collection using the weighted reference counting
--   method. Recursive deallocation of tuples is done by using an
--   explicit deallocation stack.
--
-- IN tuple: address of tuple to that reference has been deleted
-- IN wght:  weight field of reference deleted
-------------------------------------------------------------------------
PROC Garbage.Collection(VAL INT tuple, wght)

  INT weight: -- actual reference weight
  INT count:  -- reference count of tuple
  INT addr:   -- address of actual tuple cell

  -----------------------------------------------------------------------
  -- Body of Garbage.Collection
  -----------------------------------------------------------------------
  SEQ

    -- compute actual weight of reference
    weight := 1
    SEQ i = 1 FOR wght
      weight := weight * 2

    -- push reference weight and tuple address on stack
    stack[0] := INT32 tuple
    [4]BYTE s RETYPES stack[0]:
    s[3] := BYTE weight
    top := 1

    -- recursively free tuples
    WHILE top > 0
      SEQ

        -- pop reference weight and tuple address from stack
```

```
          top := top-1
          INT32 act IS stack[top]:
          SEQ
            [4]BYTE a RETYPES act:
            SEQ
              weight := INT a[3]
              a[3] := 0(BYTE)
            addr := INT act

          -- compute reference count of tuple
          [2]BYTE tags RETYPES tag[addr]:
          count := INT tags[1]

          -- update reference count
          IF
            count < weight
              Assert(FALSE, weight.err)
            count = (Range.Count-1)      -- infinite reference count
              SKIP
            count = weight               -- reference count decreases to 0
              Dealloc.Tuple(addr)
            TRUE                         -- reference count stays positive
              [2]BYTE tags RETYPES tag[addr]:
              tags[1] := BYTE (count - weight)

      -- update load
      mLoad := Load(heap.load, exp.load)
  :

  -------------------------------------------------------------------------
  -- Copy Values
  --
  --   Copies a tuple reference. If the weight field of the reference is 0,
  --   the value remains unchanged and done is set to FALSE. Otherwise,
  --   the weight field of the reference is updated.
  --
  -- IO  data: tuple reference to be copied
  -- OUT done: TRUE iff copy could be done
  -------------------------------------------------------------------------
  PROC Copy(INT32 data, BOOL done)

    BOOL error: -- error bit of reference
    INT  wght:  -- weight field of reference

    -----------------------------------------------------------------------
    -- Body of Copy
    -----------------------------------------------------------------------
    SEQ

      -- error bit and actual weight field
      error := BOOL(data /\ #00000001(INT32))
      wght := (INT (data >> 1)) /\ #00000003

      IF
        data = 0(INT32) -- nil pointer
```

```
          done := TRUE
        error           -- error bit set
          done := TRUE
        wght = 0        -- weight field 0
          done := FALSE
        TRUE            -- decrease weight field
          SEQ
            done := TRUE
            data := (data /\ #FFFFFFF9(INT32)) \/ (INT32 ((wght - 1) << 1))
:


------------------------------------------------------------------------
-- Handle Allocation Requests
--
--   Allocates a tuple with specified number of links by linking an
--   appropriate number of heap cells together. If the allocatin request
--   cannot be fulfilled, it is forwarded to the least loaded memory
--   unit among the neighbour modules.
--
-- IN alloc.tok: allocation request to be fulfilled
------------------------------------------------------------------------
PROC Alloc.Request(VAL [Token.Size]INT32 alloc.tok)

  INT  num: -- number of required tuple fields
  INT  new: -- address of new tuple allocated

  [Token.Size]INT32 token: -- network token

  ----------------------------------------------------------------------
  -- Body of Alloc.Request
  ----------------------------------------------------------------------
  SEQ

    -- extract number of tuple fields
    VAL [4]BYTE t RETYPES alloc.tok[2]:
    num := INT t[0]

    IF
      Avail.Tuple(num)              -- tuple may be allocated
        SEQ

          -- allocate tuple
          Alloc.Tuple(num, new)

          -- construct message token with tuple address
          [4]BYTE toks RETYPES token[2]:
          VAL [4]BYTE t RETYPES alloc.tok[0]:
          VAL BYTE mod IS t[3]:
          SEQ
            toks[2] := mod          -- module address
            toks[1] := Mess.PToken  -- message token
          token[1] :=  alloc.tok[1] -- context and offset of caller
          token[0] :=               -- reference with maximal weight
            INT32 ((new << 3) \/ #00000006) -- and error flag not set
```

```
          [4]BYTE toks RETYPES token[0]:
          toks[3] := act.mod        -- module address in reference

          -- send token into network
          out ! net.token; token

          -- update load
          mLoad := Load(heap.load, exp.load)

      TRUE                          -- forward allocation request
        SEQ

          -- token is to be forwarded
          token := alloc.tok

          -- replace module address by address of least loaded neighbour
          [4]BYTE toks RETYPES token[2]:
          toks[2] := Min.Loaded()

          -- send token into network
          out ! net.token; token

:

-------------------------------------------------------------------------
-- Handle Read Requests
--
--   Reads some particular field of a specified tuple and returns its
--   contents to the caller. However, if no value has yet been written
--   into the tuple field, the read request is queued into a waiting
--   list. As soon as the desired value has arrived, the read request is
--   fulfilled.
--
-- IN read.tok: read request to be fulfilled
-------------------------------------------------------------------------
PROC Read.Request(VAL [Token.Size]INT32 read.tok)

  INT   num:     -- number of referenced tuple field
  INT32 contoff: -- context and offset of caller
  BYTE  mod:     -- module address of caller
  INT   addr:    -- address of tuple
  INT   wght:    -- weight field of tuple reference
  INT   addr0:   -- address of actual tuple field
  BOOL  error:   -- error flag of tuple reference
  BOOL  open:    -- is actual tuple cell not the last?
  INT   new:     -- address of new cell allocated
  BOOL  avHd:    -- head value available
  BOOL  avTl:    -- tail value available
  INT16 t:       -- actual tag
  BOOL  gc:      -- do garbage collection
  BOOL  ptr:     -- is head/tail value pointer?
  INT32 data:    -- data of actual tuple field
  BOOL  done:    -- tuple reference could be copied

  [Token.Size]INT32 token: -- network token
```

```
-----------------------------------------------------------------------
-- Body of Read.Request
-----------------------------------------------------------------------
SEQ

  ---------------------------------------------------------------------
  -- extract token components
  ---------------------------------------------------------------------

  VAL [4]BYTE t RETYPES read.tok[2]:
  num := INT t[0]                   -- field number
  contoff := read.tok[1]            -- activation context
  VAL [4]BYTE t RETYPES read.tok[0]:
  mod := t[3]                       -- module address
  addr := INT read.tok[0]
  [4]BYTE a RETYPES addr:
  a[3] := 0(BYTE)
  error := BOOL (addr /\ #00000001) -- error flag
  addr := addr >> 1
  wght := addr /\ #00000003         -- weight field
  addr := addr >> 2                 -- tuple address

  -- check tuple reference for validity
  Assert((NOT error) AND (addr > 0) AND (tag[addr]>0(INT16)), ref.err)

  ---------------------------------------------------------------
  -- find specified tuple field
  ---------------------------------------------------------------

  -- actual tuple field
  addr0 := addr

  -- traverse tuple fields
  open := (tag[addr0] /\ #0010(INT16)) > 0(INT16)
  WHILE open AND (num > 0)
    SEQ
      num := num - 1
      addr0 := INT tail[addr0]
      open := (tag[addr0] /\ #0010(INT16)) > 0(INT16)

  -- denoted tuple field found?
  Assert(num < 2, field.err)

  -- fetch actual tag
  t := tag[addr0]

  -- do garbage collection
  gc := TRUE

  IF
    (num = 0) OR open
      SEQ

        ---------------------------------------------------------------
```

```
-- value stored in head field
---------------------------------------------------------------

-- check if head value available and ptr
avHd := (t /\ #0008(INT16)) > 0(INT16)
ptr := (t /\ #0004(INT16)) > 0(INT16)

IF
  avHd AND (NOT ptr)       -- value available and not pointer
    SEQ

      -- construct message token
      [4]BYTE toks RETYPES token[2]:
      SEQ
        toks[2] := mod          -- module address
        toks[1] := Mess.PToken  -- message token
      token[1] :=  contoff      -- context and offset
      token[0] :=  head[addr0]  -- result value

      -- send token into network
      out ! net.token; token

      -- collect unreferenced cells
      Garbage.Collection(addr, wght)

  avHd AND ptr                   -- value available and pointer
    SEQ

      -- copy pointer value
      data := head[addr0]
      Copy(data, done)

      IF
        done                    -- pointer could be copied
          SEQ

            -- recopy data
            head[addr0] := data

            -- construct message token
            [4]BYTE toks RETYPES token[2]:
            SEQ
              toks[2] := mod          -- module address
              toks[1] := Mess.PToken  -- message token
            token[1] :=  contoff      -- context and offset
            token[0] :=  data         -- result value

            -- send token into network
            out ! net.token; token

            -- collect unreferenced cells
            Garbage.Collection(addr, wght)

        TRUE                 -- increase message neccessary
          SEQ
```

```

            -- construct message
            [4]BYTE toks RETYPES token[2]:
            [4]BYTE d RETYPES data:
            SEQ
              toks[2] := d[3]          -- module address
              toks[1] := CopRef.MToken -- increase reference
            [4]BYTE toks RETYPES token[1]:
            VAL [4]BYTE r RETYPES read.tok[0]:
            VAL [4]BYTE t RETYPES read.tok[2]:
            SEQ                        -- node address
              [toks FROM 0 FOR 3] := [r FROM 0 FOR 3]
              toks[3] := t[0]          -- field number
            [4]BYTE toks RETYPES token[0]:
            [4]BYTE d RETYPES data:
            SEQ
              [toks FROM 0 FOR 3] := [d FROM 0 FOR 3]
              toks[3] := act.mod

            -- send message into network
            out ! net.token; token

            -- head not available
            INT16 t IS tag[addr0]:
            t := t /\ #FFF7(INT16)
            head[addr0] := 0(INT32)
            avHd := FALSE

            -- prohibit garbage collection
            gc := FALSE

  TRUE
    SKIP


IF
  NOT avHd                   -- head not available

    IF
      free > 0               -- cell may be allocated
        SEQ

          -- allocate new cell for deferred read request
          new := free
          free := INT tail[free]
          heap.load := heap.load+1

          -- link request into list of deferred reads
          [2]BYTE t RETYPES tag[new]:
          t[0] := mod                 -- module address
          head[new] := contoff        -- context and offset
          INT32 hd IS head[addr0]:
          SEQ
            tail[new] := hd
            hd := INT32 new
```

```
              -- collect unreferenced cells
              IF
                gc
                  Garbage.Collection(addr, wght)
                TRUE
                  SKIP

          TRUE                   -- no heap space available
            Assert(FALSE, space.err)

      TRUE
        SKIP

TRUE
  SEQ

    ---------------------------------------------------------------
    -- value stored in tail field
    ---------------------------------------------------------------

    -- check if tail value available
    avTl := (t /\ #0002(INT16)) > 0(INT16)
    ptr := (t /\ #0001(INT16)) > 0(INT16)

    IF
      avTl AND (NOT ptr)       -- value available and not pointer
        SEQ

          -- construct message token
          [4]BYTE toks RETYPES token[2]:
          SEQ
            toks[2] := mod          -- module address
            toks[1] := Mess.PToken  -- message token
          token[1] :=  contoff      -- context and offset
          token[0] :=  tail[addr0]  -- result value

          -- send token into network
          out ! net.token; token

          -- collect unreferenced cells
          Garbage.Collection(addr, wght)

      avTl AND ptr                   -- value available and pointer
        SEQ

          -- copy pointer value
          data := tail[addr0]
          Copy(data, done)

          IF
            done                    -- pointer could be copied
              SEQ

                -- recopy data
```

```
            tail[addr0] := data

            -- construct message token
            [4]BYTE toks RETYPES token[2]:
            SEQ
              toks[2] := mod          -- module address
              toks[1] := Mess.PToken  -- message token
            token[1] :=  contoff      -- context and offset
            token[0] :=  data         -- result value

            -- send token into network
            out ! net.token; token

            -- collect unreferenced cells
            Garbage.Collection(addr, wght)

        TRUE                 -- increase message neccessary
          SEQ

            -- construct message
            [4]BYTE toks RETYPES token[2]:
            [4]BYTE d RETYPES data:
            SEQ
              toks[2] := d[3]          -- module address
              toks[1] := CopRef.MToken -- increase reference
            [4]BYTE toks RETYPES token[1]:
            VAL [4]BYTE r RETYPES read.tok[0]:
            VAL [4]BYTE t RETYPES read.tok[2]:
            SEQ                        -- node address
              [toks FROM 0 FOR 3] := [r FROM 0 FOR 3]
              toks[3] := t[0]          -- field number
            [4]BYTE toks RETYPES token[0]:
            [4]BYTE d RETYPES data:
            SEQ
              [toks FROM 0 FOR 3] := [d FROM 0 FOR 3]
              toks[3] := act.mod

            -- send message into network
            out ! net.token; token

            -- tail not available
            INT16 t IS tag[addr0]:
            t := t /\ #FFFD(INT16)
            tail[addr0] := 0(INT32)
            avTl := FALSE

            -- prohibit garbage collection
            gc := FALSE

  TRUE
    SKIP

IF
  NOT avTl                   -- tail not available

```

```
              IF
                free > 0               -- cell may be allocated
                  SEQ

                    -- allocate new cell for deferred read request
                    new := free
                    free := INT tail[free]
                    heap.load := heap.load+1

                    -- link request into list of deferred reads
                    [2]BYTE t RETYPES tag[new]:
                    t[0] := mod                 -- module address
                    head[new] := contoff        -- context and offset
                    tail[new] := tail[addr0]
                    tail[addr0] := INT32 new

                    -- collect unreferenced cells
                    IF
                      gc
                        Garbage.Collection(addr, wght)
                      TRUE
                        SKIP

                TRUE                   -- no heap space available
                  Assert(FALSE, space.err)

            TRUE
              SKIP

:

-------------------------------------------------------------------------
-- Handle Submission Requests
--
--   Submits the lambda-lifted arguments of some higher-order function
--   stored in a tuple to a specific activation of that function.
--
-- IN sub.tok: submission request token to be fulfilled
-------------------------------------------------------------------------
PROC Submit.Request(VAL [Token.Size]INT32 sub.tok)

  BYTE  mod:     -- module address
  INT16 context: -- context of function activation to submit arguments to
  INT   addr:    -- tuple address
  INT   wght:    -- weight of tuple reference
  INT   addr0:   -- actual field address
  INT   num:     -- number of tuple field referenced
  BOOL  error:   -- error flag of reference
  BOOL  open:    -- last cell of tuple not encountered?
  BOOL  avHd:    -- head of cell available?
  BOOL  avTl:    -- tail of cell available?
  INT   new:     -- reference to new heap cell
  BOOL  ptr:     -- head/tail value pointer?
  INT32 data:    -- data of actual tuple field
  INT   count:   -- reference count of tuple
```

```
BOOL  done:    -- tuple reference was copied
INT   field:   -- actual number of tuple field

[Token.Size]INT32 token: -- network token

----------------------------------------------------------------------
-- Body of Submit.Request
----------------------------------------------------------------------
SEQ

  --------------------------------------------------------------------
  -- extract token components
  --------------------------------------------------------------------

  VAL [2]INT16 t RETYPES sub.tok[1]:
  context := t[1]                   -- activation context
  VAL [4]BYTE t RETYPES sub.tok[0]:
  mod := t[3]                       -- module address
  addr := INT sub.tok[0]
  [4]BYTE a RETYPES addr:
  a[3] := 0(BYTE)
  error := BOOL (addr /\ #00000001) -- error flag
  addr := addr >> 1
  wght := addr /\ #00000003         -- weight field
  addr := addr >> 2                 -- tuple address

  -- check tuple reference for validity
  Assert((NOT error) AND (addr > 0) AND (tag[addr]>0(INT16)), ref.err)

  --------------------------------------------------------------
  -- submit arguments in head fields
  --------------------------------------------------------------

  -- save tuple address
  addr0 := addr

  -- extract tuple flags
  INT16 t IS tag[addr0]:
  SEQ
    open := (t /\ #0010(INT16)) > 0(INT16)  -- tuple not closed?
    avHd := (t /\ #0008(INT16)) > 0(INT16)  -- head available?
    [2]BYTE ts RETYPES t:
    count := INT ts[1]                      -- reference count

  -- error flag and parameter number of function descriptor
  INT32 fun IS head[addr0]:
  SEQ
    error := BOOL (fun /\ #00000001(INT32)) -- error flag
    VAL INT32 funs IS fun >> 1:
    VAL [4]BYTE f RETYPES funs:
    num := INT f[2]                         -- parameter number

  -- check if function descriptor available and not error
  Assert(avHd AND (NOT error), submit.err)
```

```
-- actual number of tuple field
field := 1

-- submit arguments or store deferred read request
WHILE open
  SEQ

    -- address of next tuple field
    addr0 := INT (tail[addr0])

    -- extract field flags
    INT16 t IS tag[addr0]:
    SEQ
      open := (t /\ #0010(INT16)) > 0(INT16) -- last cell?
      avHd := (t /\ #0008(INT16)) > 0(INT16) -- head available?
      ptr  := (t /\ #0004(INT16)) > 0(INT16) -- head pointer?

    IF

      avHd AND (NOT ptr)     -- argument available and not pointer
        SEQ

          -- construct message token
          [4]BYTE toks RETYPES token[2]:
          SEQ
            toks[2] := mod          -- module address
            toks[1] := Mess.PToken  -- message token
          [2]INT16 toks RETYPES token[1]:
          SEQ
            toks[1] := context      -- context
            toks[0] := INT16 num    -- offset (parameter number)
          token[0] :=  head[addr0]  -- result value

          -- send token into network
          out ! net.token; token

      avHd AND ptr        -- value available and pointer
        SEQ

          -- copy pointer value
          data := head[addr0]
          Copy(data, done)

          IF
            done                    -- pointer could be copied
              SEQ

                -- recopy data
                head[addr0] := data

                -- construct message token
                [4]BYTE toks RETYPES token[2]:
                SEQ
                  toks[2] := mod          -- module address
                  toks[1] := Mess.PToken  -- message token
```

```
          [2]INT16 toks RETYPES token[1]:
          SEQ
            toks[1] := context      -- context
            toks[0] := INT16 num    -- offset (parameter)
          token[0] :=  data         -- result value

          -- send token into network
          out ! net.token; token

      TRUE                 -- increase message neccessary
        SEQ

          -- construct message
          [4]BYTE toks RETYPES token[2]:
          [4]BYTE d RETYPES data:
          SEQ
            toks[2] := d[3]          -- module address
            toks[1] := CopRef.MToken -- increase reference
          [4]BYTE toks RETYPES token[1]:
          VAL [4]BYTE s RETYPES sub.tok[0]:
          SEQ                        -- node address
            [toks FROM 0 FOR 3] := [s FROM 0 FOR 3]
            toks[3] := BYTE field    -- field number
          [4]BYTE toks RETYPES token[0]:
          [4]BYTE d RETYPES data:
          SEQ
            [toks FROM 0 FOR 3] := [d FROM 0 FOR 3]
            toks[3] := act.mod

          -- send message into network
          out ! net.token; token

          -- head not available
          INT16 t IS tag[addr0]:
          t := t /\ #FFF7(INT16)
          head[addr0] := 0(INT32)
          avHd := FALSE

          -- update reference count
          count := count+1

TRUE
  SKIP


IF
  NOT avHd                   -- head not available

    IF
      free > 0               -- cell may be allocated
        SEQ

          -- allocate new cell for deferred read request
          new := free
          free := INT tail[free]
```

```
                heap.load := heap.load+1

                -- link request into list of deferred reads
                [2]BYTE t RETYPES tag[new]:
                t[0] := mod                 -- module address
                [2]INT16 heads RETYPES head[new]:
                SEQ
                  heads[1] := context       -- context
                  heads[0] := INT16 num     -- offset (parameter)
                INT32 hd IS head[addr0]:
                SEQ
                  tail[new] := hd
                  hd := INT32 new

            TRUE                   -- no heap space available
              Assert(FALSE, space.err)

        TRUE
          SKIP

    -- parameter and field number
    num := num+1
    field := field+1

---------------------------------------------------------------------
-- submit last argument in tail field
---------------------------------------------------------------------

-- check if tail value available
INT16 t IS tag[addr0]:
SEQ
  avTl := (t /\ #0002(INT16)) > 0(INT16)
  ptr := (t /\ #0001(INT16)) > 0(INT16)

IF
  avTl AND (NOT ptr)       -- value available and not pointer
    SEQ

      -- construct message token
      [4]BYTE toks RETYPES token[2]:
      SEQ
        toks[2] := mod          -- module address
        toks[1] := Mess.PToken  -- message token
      [2]INT16 toks RETYPES token[1]:
      SEQ
        toks[1] := context      -- context
        toks[0] := INT16 num    -- offset (parameter)
      token[0] :=  tail[addr0]  -- result value

      -- send token into network
      out ! net.token; token

  avTl AND ptr                   -- value available and pointer
    SEQ
```

```
-- copy pointer value
data := tail[addr0]
Copy(data, done)

IF
  done                   -- pointer could be copied
    SEQ

      -- recopy data
      tail[addr0] := data

      -- construct message token
      [4]BYTE toks RETYPES token[2]:
      SEQ
        toks[2] := mod          -- module address
        toks[1] := Mess.PToken  -- message token
      [2]INT16 toks RETYPES token[1]:
      SEQ
        toks[1] := context      -- context
        toks[0] := INT16 num    -- offset (parameter)
      token[0] :=  data         -- result value

      -- send token into network
      out ! net.token; token

  TRUE                -- increase message neccessary
    SEQ

      -- construct message
      [4]BYTE toks RETYPES token[2]:
      [4]BYTE d RETYPES data:
      SEQ
        toks[2] := d[3]          -- module address
        toks[1] := CopRef.MToken -- increase reference
      [4]BYTE toks RETYPES token[1]:
      VAL [4]BYTE s RETYPES sub.tok[0]:
      SEQ                        -- node address
        [toks FROM 0 FOR 3] := [s FROM 0 FOR 3]
        toks[3] := BYTE field    -- field number
      [4]BYTE toks RETYPES token[0]:
      [4]BYTE d RETYPES data:
      SEQ
        [toks FROM 0 FOR 3] := [d FROM 0 FOR 3]
        toks[3] := act.mod

      -- send message into network
      out ! net.token; token

      -- tail not available
      INT16 t IS tag[addr0]:
      t := t /\ #FFFD(INT16)
      tail[addr0] := 0(INT32)
      avTl := FALSE

      -- update reference count
```

```
                count := count+1

      TRUE
        SKIP

    IF
      NOT avTl                   -- head not available

        IF
          free > 0               -- cell may be allocated
            SEQ

              -- allocate new cell for deferred read request
              new := free
              free := INT tail[free]
              heap.load := heap.load+1

              -- link request into list of deferred reads
              [2]BYTE t RETYPES tag[new]:
              t[0] := mod                 -- module address
              [2]INT16 heads RETYPES head[new]:
              SEQ
                heads[1] := context       -- context
                heads[0] := INT16 num     -- offset (parameter)
              tail[new] := tail[addr0]
              tail[addr0] := INT32 new

          TRUE                            -- no heap space available
            Assert(FALSE, space.err)

      TRUE
        SKIP

    -- update reference count field
    [2]BYTE t RETYPES tag[addr]:
    BYTE c IS t[1]:
    c := BYTE count

    -- collect unreferenced cells
    Garbage.Collection(addr, wght)

:

-------------------------------------------------------------------------
-- Handle Expensive Requests
--
--   Handles incoming allocation requests, read requests and
--   submission requests.
--
-- ATTENTION:
--
--   This procedure REFERENCES and ALTERS global variables defined
--   in the preamble of Manager.
-------------------------------------------------------------------------
PROC Exp.Request()
```

```

    [Token.Size]INT32 token: -- network token
    BYTE type:               -- token tag

    -----------------------------------------------------------------------
    -- Body of Exp.Request
    -----------------------------------------------------------------------
    SEQ

      -- receive cheap request and update store load
      demand.exp ! demand.tag
      from.exp ? exp.load; token

      IF
        exp.load = 0

          -- store is empty
          SKIP

        TRUE
          SEQ

            -- extract token tag
            [4]BYTE toks RETYPES token[2]:
            type := toks[1]

            -- perform specified action
            CASE type
              Alloc.MToken                 -- allocation requests
                Alloc.Request(token)
              Read.MToken                  -- read requests
                Read.Request(token)
              Submit.MToken                -- submit requests
                Submit.Request(token)
              Stop.Token                   -- stop token
                WHILE going
                  SEQ
                    demand.cheap ! demand.tag
                    from.cheap ? cheap.load; token
                    [4]BYTE toks RETYPES token[2]:
                    BYTE type IS toks[1]:
                    going := type <> Stop.Token

  :

  -------------------------------------------------------------------------
  -- Handle Write Requests
  --
  --   Writes message value into particular tuple field. All deferred
  --   read requests waiting for this value are fulfilled by sending
  --   value messages to the specified operations.
  --
  -- IN write.tok: write request token to be fulfilled
  -------------------------------------------------------------------------
  PROC Write.Request(VAL [Token.Size]INT32 write.tok)
```

```
INT   num:    -- number of tuple field referenced
INT32 data:   -- value to be written
INT   addr:   -- tuple address
INT   wght:   -- weight of tuple reference
INT   addr0:  -- actual field address
BOOL  error:  -- error flag of reference
BOOL  open:   -- last cell of tuple not yet encountered?
INT   next:   -- address of next tuple cell
INT   next0:  -- auxiliary address
BYTE  type:   -- type of write request
BOOL  isHd:   -- head field contains data
BOOL  isAtom: -- data is atomic
BOOL  going:  -- auxiliary variable
BOOL  done:   -- tuple reference could be copied

[Token.Size]INT32 token: -- network token

----------------------------------------------------------------------
-- Body of Write.Request
----------------------------------------------------------------------
SEQ

  --------------------------------------------------------------------
  -- extract token components
  --------------------------------------------------------------------

  VAL [4]BYTE toks RETYPES write.tok[2]:
  SEQ
    type := toks[1]                 -- type of write request
    num  := INT toks[0]             -- number of tuple field
  data := write.tok[1]              -- value to be written
  addr := INT write.tok[0]          -- tuple reference
  [4]BYTE a RETYPES addr:
  a[3] := 0(BYTE)
  error := BOOL (addr /\ #00000001) -- error flag
  addr := addr >> 1
  wght := addr /\ #00000003         -- reference weight
  addr := addr >> 2                 -- tuple address

  -- check tuple reference for validity
  Assert((NOT error) AND (addr > 0) AND (tag[addr]>0(INT16)), ref.err)

  --------------------------------------------------------------------
  -- find specified tuple field
  --------------------------------------------------------------------

  -- find specified tuple field
  addr0 := addr
  INT16 t IS tag[addr0]:
  open := (t /\ #0010(INT16)) > 0(INT16)
  WHILE open AND (num > 0)
    SEQ
      num := num - 1
      addr0 := INT tail[addr0]
```

```
    INT16 t IS tag[addr0]:
    open := (t /\ #0010(INT16)) > 0(INT16)

-- check field number
Assert(num < 2, field.err)

---------------------------------------------------------------------
-- find deferred read requests and write value into tuple field
---------------------------------------------------------------------

-- head or data field, atomic value or pointer?
isHd := (num = 0) OR open
isAtom := (type = WrInt.MToken) OR (data = 0(INT32))

INT16 t IS tag[addr0]:            -- tag field
IF
  isHd                            -- head field specified
    SEQ
      Assert((t/\#0008(INT16))=0(INT16), mult.write)
      next := INT head[addr0]
      head[addr0] := data
      IF
        isAtom                    -- head available flag
          t := t \/ #0008(INT16)
        TRUE                      -- head available and pointer
          t := t \/ #000C(INT16)
  TRUE                            -- tail field specified
    SEQ
      Assert((t/\#0002(INT16))=0(INT16), mult.write)
      next := INT tail[addr0]
      tail[addr0] := data
      IF
        isAtom                    -- tail available flag
          t := t \/ #0002(INT16)
        TRUE                      -- tail available and pointer
          t := t \/ #0003(INT16)

---------------------------------------------------------------------
-- fulfill deferred reads
---------------------------------------------------------------------

-- traverse list of deferred reads
going := TRUE
WHILE (next > 0) AND going
  IF

    isAtom
      SEQ

        -- construct token
        [2]BYTE tags RETYPES tag[next]:
        [4]BYTE toks RETYPES token[2]:
        SEQ
          toks[2] := tags[0]            -- module address
          toks[1] := Mess.PToken        -- message token
```

```
          token[1] := head[next]           -- context and offset
          token[0] := data                 -- message value

          -- send token to network
          out ! net.token; token

          -- free heap cell
          next0 := INT tail[next]
          tag[next] := 0(INT16)
          tail[next] := INT32 free
          free := next

          -- update heap load
          heap.load := heap.load-1

          -- next deferred read
          next := next0

      TRUE
        SEQ

          -- copy pointer value
          Copy(data, done)

          IF
            done                    -- pointer could be copied
              SEQ

                -- recopy data
                IF
                  isHd
                    head[addr0] := data
                  TRUE
                    tail[addr0] := data

                -- construct token
                [2]BYTE tags RETYPES tag[next]:
                [4]BYTE toks RETYPES token[2]:
                SEQ
                  toks[2] := tags[0]           -- module address
                  toks[1] := Mess.PToken       -- message token
                token[1] := head[next]         -- context and offset
                token[0] := data               -- message value

                -- send token to network
                out ! net.token; token

                -- free heap cell
                next0 := INT tail[next]
                tag[next] := 0(INT16)
                tail[next] := INT32 free
                free := next

                -- update heap load
                heap.load := heap.load-1
```

```

                  -- next deferred read
                  next := next0

              TRUE                 -- increase message neccessary
                SEQ

                  -- construct message
                  [4]BYTE toks RETYPES token[2]:
                  [4]BYTE d RETYPES data:
                  SEQ
                    toks[2] := d[3]          -- module address
                    toks[1] := CopRef.MToken -- increase reference
                  [4]BYTE toks RETYPES token[1]:
                  VAL [4]BYTE w RETYPES write.tok[0]:
                  VAL [4]BYTE t RETYPES write.tok[2]:
                  SEQ                        -- node address
                    [toks FROM 0 FOR 3] := [w FROM 0 FOR 3]
                    toks[3] := t[0]          -- field number
                  [4]BYTE toks RETYPES token[0]:
                  [4]BYTE d RETYPES data:
                  SEQ
                    [toks FROM 0 FOR 3] := [d FROM 0 FOR 3]
                    toks[3] := act.mod

                  -- send message into network
                  out ! net.token; token

                  -- head/tail not available
                  INT16 t IS tag[addr0]:
                  IF
                    isHd
                      SEQ
                        t := t /\ #FFF7(INT16)
                        head[addr0] := INT32 next
                    TRUE
                      SEQ
                        t := t /\ #FFFD(INT16)
                        tail[addr0] := INT32 next

                  -- terminate loop
                  going := FALSE

    -- collect unreferenced cells
    IF
      next = 0
        Garbage.Collection(addr, wght)
      TRUE
        SKIP

:

-------------------------------------------------------------------------
-- Handle Reference Requests
--
```

```
--   Increase respectively decrease reference count field of some
--   particular tuple. If the reference count is increased, a
--   reference with an adapted weight field is sent to the caller.
--   If the reference count decreases to zero, the tuple is reclaimed
--   to the free list.
--
-- IN ref.tok: reference request token to be handled
-------------------------------------------------------------------------
PROC Ref.Request(VAL [Token.Size]INT32 ref.tok)

  BYTE  type:    -- request type (increase or decrease)
  INT   addr:    -- tuple address
  INT   wght:    -- reference weight field
  BOOL  error:   -- tuple reference error bit

  [Token.Size]INT32 token: -- network token

  -----------------------------------------------------------------------
  -- Body of Ref.Request
  -----------------------------------------------------------------------
  SEQ

    -- extract token components
    VAL [4]BYTE toks RETYPES ref.tok[2]:
    type := toks[1]
    addr := INT ref.tok[0]
    [4]BYTE a RETYPES addr:
    a[3] := 0(BYTE)
    error := BOOL (addr /\ #00000001) -- error flag
    addr := addr >> 1
    wght := addr /\ #00000003         -- reference weight
    addr := addr >> 2                 -- tuple address

    -- check tuple reference for validity
    Assert((NOT error) AND (addr > 0) AND (tag[addr]>0(INT16)), ref.err)

    -- increase or decrease reference count
    CASE type

      -- increase reference count of tuple, send result to processor unit
      IncRef.MToken
        SEQ

          -- update reference count
          [2]BYTE tags RETYPES tag[addr]:
          BYTE count IS tags[1]:
          SEQ
            IF                       -- update reference count
              (Range.Count-7) > (INT count)
                count := BYTE((INT count) + 7)
              TRUE                   -- reference count infinity
                count := BYTE (Range.Count-1)

          -- construct token
          VAL [4]BYTE t RETYPES ref.tok[0]:
```

```
          [4]BYTE toks RETYPES token[2]:
          SEQ
            toks[2] := t[3]         -- module address
            toks[1] := Mess.PToken  -- message token
          token[1] :=  ref.tok[1]   -- context and offset of caller
          token[0] :=               -- reference with maximal weight
            INT32 ((addr << 3) \/ #00000006) -- and error bit not set
          [4]BYTE toks RETYPES token[0]:
          toks[3] := act.mod

          -- send token into network
          out ! net.token; token

      -- increase reference count of tuple, send result to memory unit
      CopRef.MToken
        SEQ

          -- update reference count
          [2]BYTE tags RETYPES tag[addr]:
          BYTE count IS tags[1]:
          SEQ
            IF                          -- update reference count
              (Range.Count-7) > (INT count)
                count := BYTE((INT count) + 7)
              TRUE                      -- reference count infinity
                count := BYTE (Range.Count-1)

          -- construct token
          VAL [4]BYTE t1 RETYPES ref.tok[1]:
          VAL [4]BYTE t0 RETYPES ref.tok[0]:
          [4]BYTE toks RETYPES token[2]:
          SEQ
            toks[2] := t0[3]        -- module address
            toks[1] := WrPtr.MToken -- write token
            toks[0] := t1[3]        -- field number
          token[1] :=               -- reference with maximal weight
            INT32 ((addr << 3) \/ #00000006) -- and error bit not set
          [4]BYTE toks RETYPES token[1]:
          toks[3] := act.mod        -- actual module address
          token[0] :=  ref.tok[1]   -- tuple address of caller

          -- send token into network
          out ! net.token; token

      -- decrease reference count of tuple and perform garbage collection
      DecRef.MToken
        Garbage.Collection(addr, wght)

:

-------------------------------------------------------------------------
-- Receive Memory Load Report
--
--   Update load description of neighbour memory.
--
```

```
-- IN token: load report token
------------------------------------------------------------------------
PROC Load.Message(VAL [Token.Size]INT32 token)

  ----------------------------------------------------------------------
  -- Body of Load.Message
  ----------------------------------------------------------------------
  SEQ

    VAL [4]BYTE toks RETYPES token[2]:
    VAL BYTE link      IS      toks[0]:  -- link address of module
    VAL INT  heap.load RETYPES token[1]: -- its heap load
    VAL INT  exp.load  RETYPES token[0]: -- its frame load

    -- update load description
    mem.load[INT link] := Load(heap.load, exp.load)

:

------------------------------------------------------------------------
-- Handle Cheap Requests
--
--   Handles incoming write requests and memory load reports.
--
-- ATTENTION:
--
--   This procedure REFERENCES and ALTERS global variables defined
--   in the preamble of Manager.
------------------------------------------------------------------------
PROC Cheap.Request()

  [Token.Size]INT32 token: -- network token
  BYTE type:               -- token tag

  ----------------------------------------------------------------------
  -- Body of Cheap.Request
  ----------------------------------------------------------------------
  SEQ

    -- receive cheap request and update store load
    demand.cheap ! demand.tag
    from.cheap ? cheap.load; token

    IF
      cheap.load = 0

        -- store is empty
        SKIP

      TRUE
        SEQ

          -- extract token tag
          [4]BYTE toks RETYPES token[2]:
          type := toks[1]
```

```
          -- perform specified action
          CASE type
            WrInt.MToken, WrPtr.MToken    -- write requests
              Write.Request(token)
            IncRef.MToken, DecRef.MToken,
            CopRef.MToken                 -- reference requests
              Ref.Request(token)
            MLoad.MToken                  -- load report messages
              Load.Message(token)
            Stop.Token                    -- stop token
              WHILE going
                SEQ
                  demand.exp ! demand.tag
                  from.exp ? exp.load; token
                  [4]BYTE toks RETYPES token[2]:
                  BYTE type IS toks[1]:
                  going := type <> Stop.Token

:

-------------------------------------------------------------------------
-- Report Memory Load:
--
--   Tells the memories on all neighbour modules and the master
--   module about the current workload of heap and expensive store.
--
-- ATTENTION:
--
--   This procedure REFERENCES and ALTERS global variables defined
--   in the preamble of Manager.
-------------------------------------------------------------------------
PROC Load.Report()

  [Token.Size]INT32 token: -- network token

  -----------------------------------------------------------------------
  -- Body of Load.Report
  -----------------------------------------------------------------------
  SEQ

    -- store load to be reported
    lLoad := mLoad

    -- construct token
    [4]BYTE toks RETYPES token[2]:
    toks[1] := MLoad.MToken        -- memory load token
    token [1] := INT32 heap.load   -- heap load
    token [0] := INT32 exp.load    -- exprensive store load

    -- send report to master
    [4]BYTE toks RETYPES token[2]:
    SEQ
      toks[2] := 0(BYTE)           -- address of master
      toks[0] := act.mod           -- actual module address
```

```
        out ! net.token; token

        -- send report to neighbours
        SEQ i=0 FOR Link.Num
          SEQ
            [4]BYTE toks RETYPES token[2]:
            SEQ
              toks[2] := mod.addr[i]   -- address of neighbour
              toks[0] := link.addr[i]  -- unique memory idenitifier
            out ! net.token; token

    :

    ------------------------------------------------------------------------
    -- Body of Manager
    ------------------------------------------------------------------------
    SEQ

      WHILE TRUE
        SEQ

          -- create free list (cell 0 is unused since 0 denotes NIL)
          free := 1
          tag[0] := 0(INT16)
          SEQ i = 1 FOR Heap.Len-1
            SEQ
              tag[i] := 0(INT16)
              tail[i] := INT32 (i+1)
          tag[Heap.Len-1] := 0(INT16)
          tail[Heap.Len-1] := 0(INT32)

          -- reset load variables
          heap.load, exp.load, cheap.load := 0, 0, 0

          -- reset actual and last reported load
          mLoad, lLoad := 0, 0

          -- reset load of all neighbour modules
          SEQ i = 0 FOR Link.Num
            mem.load[i] := 0

          -- send acknowledge to arbiter
          out ! net.ack

          -- manager main loop
          going := TRUE
          WHILE going
            SEQ
              Mem.Action(action)  -- determine next action
              CASE action
                Exp.Act
                  Exp.Request()   -- fulfill expensive request
                Cheap.Act
                  Cheap.Request() -- fulfill cheap request
                Report.Act
```

```
                  Load.Report()   -- report actual load

          -- send acknowledge to arbiter
          out ! net.ack

      -- never done (just to avoid compiler warnings)
      debug ! 0 :: ""

  :

  ----------------------------------------------------------------------------
  -- Body of Adam.Memory
  ---------------------------------------------------------------------------
  PAR

    -- routes incoming tokens to expensive or cheap store
    Router(in, to.exp, to.cheap, error[0])

    -- buffers incoming expensive requests
    Buffer(to.exp, demand.exp, exp.store, from.exp, error[1])

    -- buffers incoming cheap requests
    Buffer(to.cheap, demand.cheap, cheap.store, from.cheap, error[2])

    -- manages memory unit
    Manager(from.exp, from.cheap, demand.exp, demand.cheap, error[3])

    -- multiplexor for error messages
    INT len:               -- length of message
    [Max.Debug]BYTE mess:  -- error message
    ALT i=0 FOR 4
      error[i] ? len :: mess
        debug ! len :: mess

:

------------------------------------------------------------------------------
-- End of ADMEMORY.OCC
------------------------------------------------------------------------------
```

ADPROC.OCC Page 1

```
-----------------------------------------------------------------------------
--
-- ADPROC.OCC                                        Wolfgang Schreiner
-- The Adam Processor Unit                           (c) RISC-Linz 1990
--
-- Interpreter for all components of the Adam processor unit.
--
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- External References
-----------------------------------------------------------------------------

-- include files
#INCLUDE "adglobal.inc" -- global constants
#INCLUDE "adnet.inc"    -- network constants
#INCLUDE "aderror.inc"  -- error messages
#INCLUDE "adproc.inc"   -- processor constants

-----------------------------------------------------------------------------
-- The Adam Processor Unit
--
--   The processor unit receives
--     1. the executable program,
--     2. function call requests that are handled by the module,
--     3. a stop signal that breaks the execution of the unit.
--   The sequence 1-3 loops forever, i.e. the unit does not terminate.
--
--   IN  act.mod:   address of actual module
--   IN  mod.addr:  addresses of neighbour modules
--   IN  link.addr: link addresses of actual module from view of neighbour
--   IN  in:        input channels from arbiter
--   OUT out:       output channels to arbiter
--   OUT debug:     debug channel to output process
-----------------------------------------------------------------------------
PROC Adam.Processor(VAL BYTE act.mod,
  VAL [Link.Num]BYTE mod.addr, VAL [Link.Num]BYTE link.addr,
  CHAN OF Mod.Link in, CHAN OF Net.Link out, CHAN OF Debug debug)

  -- internal channels
  CHAN OF Cluster from.rout:                 -- transmit program clusters
  CHAN OF Proc.Token to.requ, to.mess:       -- transmit tokens to buffers
  CHAN OF Buffer.Token from.requ, from.mess: -- transmit tokens to execution
  CHAN OF Demand demand.requ, demand.mess:   -- transmit demands

  -- buffers for message and request store
  [Token.Size*Mess.Len]INT32 mess.store: -- message store
  [Token.Size*Requ.Len]INT32 requ.store: -- request store

  [4]CHAN OF Debug error: -- internal debugging channels

  ---------------------------------------------------------------------------
  -- The Processor Unit Router:
  --
  --   The Router receives incoming data from the arbiter and routes them
```

```
--   as follows:
--     1. Programs to the execution unit,
--     2. Message tokens to the message store,
--     3. Request tokens to the request store.
--
--   IN  in:      channel from arbiter
--   OUT to.mess: channel to message store
--   OUT to.requ: channel to request store
--   OUT to.exec: channel to execution unit
--   OUT debug:   debugging channel
-------------------------------------------------------------------------
PROC Router(CHAN OF Mod.Link in,
  CHAN OF Proc.Token to.mess, CHAN OF Proc.Token to.requ,
  CHAN OF Cluster to.exec, CHAN OF Debug debug)

  [Cluster.Size]INT32 prog:    -- program cluster
  [Token.Size]INT32   token:   -- network token
  BYTE                tag:     -- token tag
  BOOL                going:   -- auxiliary variable

  ----------------------------------------------------------------------
  -- Body of Router
  ----------------------------------------------------------------------
  SEQ

    -- loop forever
    WHILE TRUE
      SEQ

        -- receive program and forward it to execution unit
        going := TRUE
        WHILE going
          in ? CASE
            mod.startprog
              to.exec ! start.cluster
            mod.appendprog
              to.exec ! append.cluster
            mod.program; prog
              to.exec ! cluster; prog
            mod.endprog
              SEQ
                to.exec ! end.cluster
                going := FALSE

        -- route tokens to appropriate stores until stop signal encountered
        going := TRUE
        WHILE going
          SEQ
            in ? CASE mod.token; token
            [4]BYTE toks RETYPES token[2]:
            tag := toks[1]
            IF
              tag = Stop.Token         -- stop token
                SEQ
                  to.mess ! token
```

```
                  to.requ ! token
                  going := FALSE
              ((INT tag) /\ #F0) = 0   -- message token
                to.mess ! token
              TRUE                     -- request token
                to.requ ! token

    -- never done (just to avoid compiler warnings)
    debug ! 0 :: ""
:

--------------------------------------------------------------------------
-- The Processor Unit Buffer
--
--   Buffers incoming tokens in a queue and forwards them to the
--   execution unit on demand.
--
--   IN  in:     input channel from router
--   IN  demand: demand channel from execution unit
--   IN  buffer: store to hold tokens
--   OUT out:    output channel to execution unit
--   OUT debug:  debugging channel
--------------------------------------------------------------------------
PROC Buffer(CHAN OF Proc.Token in, CHAN OF Demand demand, []INT32 buffer,
  CHAN OF Buffer.Token out, CHAN OF Debug debug)

  VAL buffer.size IS SIZE buffer:

  INT  first, last: -- indizes of the first resp. last queue elements
  INT  num:         -- number of elements in queue
  BOOL full, empty: -- denote the actual status of the queue
  BOOL going:       -- termination flag
  BYTE tag:         -- tag of actual token

  [Token.Size]INT32 token: -- token value

  ------------------------------------------------------------------------
  -- Assert Condition
  --
  --   Checks condition and forwards error message to debugging channel,
  --   if neccessary.
  --
  -- IN cond: test condition
  -- IN mess: error message
  ------------------------------------------------------------------------
  PROC Assert(VAL BOOL cond, VAL []BYTE mess)
    SEQ
      IF
        cond
          SKIP
        TRUE
          SEQ
            debug ! (SIZE mess) :: mess
            STOP
  :
```

```
------------------------------------------------------------------------
-- Body of Buffer
------------------------------------------------------------------------
WHILE TRUE
  SEQ

    -- buffer length must be a multiple of Token.Size
    Assert((buffer.size REM Token.Size) = 0, buff.err)

    -- initialize status variables
    num, first, last := 0, 0, 0
    full, empty := FALSE, TRUE

    -- loop until stop token encountered
    going := TRUE
    WHILE going

      -- full equals not empty
      -- if first <> last
      --   then first refers to the next free element
      --        last refers to the last stored element
      --   else if full
      --          then first is not valid
      --          else last is not valid

      -- listen on both input channels
      ALT

        -- demand received
        demand ? CASE demand.tag
          SEQ

            -- submit token
            [Token.Size]INT32 token IS [buffer FROM last FOR Token.Size]:
            out ! num; token

            -- extract tag
            [Token.Size]INT32 token IS [buffer FROM last FOR Token.Size]:
            [4]BYTE toks RETYPES token[2]:
            tag := toks[1]

            IF
              tag = Stop.Token  -- stop token submitted, empty buffer
                going := FALSE
              empty             -- submitted token was undefined
                SKIP
              TRUE              -- update state variables
                SEQ
                  num  := num-1
                  last := (last + Token.Size) REM buffer.size
                  full := FALSE
                  empty := first = last

        -- token received
```

```
          in ? token
            SEQ

              -- buffer must not be full
              Assert(NOT full, buff.full)

              -- store token
              [buffer FROM first FOR Token.Size] := token
              num   := num+1
              first := (first + Token.Size) REM buffer.size
              empty := FALSE
              full  := first = last

:

--------------------------------------------------------------------------
-- The Processor Execution Unit
--
--   Receives the addresses of the neighbour modules and the machine code
--   of the Adam program to be executed, then handles the incoming
--   messages and requests. Program execution is terminated by a stop
--   request that resets the execution unit to its initial state. The unit
--   then is ready to receive a new program.
--
--   The execution unit itself does not terminate.
--
--   IN from.rout:    channel from router transmitting the machine code
--   IN from.requ:    channel from request store
--   IN from.mess:    channel from message store
--   OUT demand.requ: demand channel to request store
--   OUT demand.mess: demand channel to message store
--   OUT debug:       debugging channel
---------------------------------------------------------------------------
PROC Execution.Unit(CHAN OF Cluster from.rout,
  CHAN OF Buffer.Token from.requ, from.mess,
  CHAN OF Demand demand.requ, demand.mess, CHAN OF Debug debug)

  -- the basic stores
  [Prog.Len]INT32  prog.store:    -- the program store
  [Frame.Len]INT32 frame.store:   -- the frame store
  [Cont.Len]INT32  cont.store.f:  -- the context store (frame pointer)
  [Cont.Len]INT32  cont.store.c:  -- the context store (code pointer)
  [Trig.Len]INT32  trig.store:    -- the trigger store
  [Used.Len]INT32  used.store:    -- used bits for frame store

  -- the load parameters
  INT frame.load: -- load of frame store
  INT cont.load:  -- load of context store
  INT trig.load:  -- load of trigger store
  INT requ.load:  -- load of request store
  INT mess.load:  -- load of message store

  INT pLoad:   -- relative load of actual processor
  INT lLoad:   -- last load reported to neighbours

```

```
-- heap organizing variables
[Free.Step]INT frame.free:  -- heads of the frame store free lists
INT cont.free:              -- head of context store free list

-- network descriptors
[Link.Num]INT proc.load: -- relative load of processors

-- the following quantities are currently not used
-- [Link.Num]INT  proc.frame.load: -- load of neighbours' frame stores
-- [Link.Num]INT  proc.requ.load:  -- load of neighbours' request stores

-- auxiliary variables
INT32 null:    -- conversion auxiliary
INT   action:  -- next action to be executed
INT   pos:     -- actual program store position
INT   max.pos: -- actual length of program without appended fragment
BOOL  append:  -- program fragment is to be appended
BOOL  going:   -- termination flag

------------------------------------------------------------------------
-- Auxiliary Procedures in Execution Unit
------------------------------------------------------------------------

------------------------------------------------------------------------
-- Assert Condition
--
--   Checks condition and forwards error message to debugging channel,
--   if neccessary.
--
-- IN cond: test condition
-- IN mess: error message
------------------------------------------------------------------------
PROC Assert(VAL BOOL cond, VAL []BYTE mess)

  ----------------------------------------------------------------------
  -- Body of Assert
  ----------------------------------------------------------------------
  SEQ
    IF
      cond
        SKIP
      TRUE
        SEQ
          debug ! (SIZE mess) :: mess
          STOP
:

------------------------------------------------------------------------
-- Put Trigger into Store
--
--   Pushes trigger on stack-organized trigger store
--
-- IN trigger: instruction trigger
--
-- ATTENTION:
```

```
--
--   Alters global variables trig.load and trig.store!
------------------------------------------------------------------------
PROC Trig.Put(VAL INT32 trigger)

  ----------------------------------------------------------------------
  -- Body of Trig.Put
  ----------------------------------------------------------------------
  SEQ

    -- stack must not be full
    Assert(trig.load < Trig.Len, trig.full)

    -- push trigger on stack
    trig.store[trig.load] := trigger
    trig.load := trig.load+1
:

------------------------------------------------------------------------
-- Fetch Trigger from Store
--
--   Pops trigger from stack-organized trigger store
--
-- OUT trigger: instruction trigger
--
-- ATTENTION:
--
--   Alters global variables trig.load and trig.store!
------------------------------------------------------------------------
PROC Trig.Fetch(INT32 trigger)

  ----------------------------------------------------------------------
  -- Body of Trig.Fetch
  ----------------------------------------------------------------------
  SEQ

    -- store must not be empty
    Assert(trig.load>0, trig.empt)

    -- pop trigger from stack
    trig.load := trig.load-1
    trigger := trig.store[trig.load]
:

-------------------------------------------------------------------------
-- Compute Relative Load
--
--   Takes frame load and request load of some processor unit and
--   computes its relative load, i.e. a value between 0 and 10000
--   that (hopefully adequately) describes the overall working load of
--   the processor: a value of 0 should denote "the processor is
--   totally idle" and a value of 10000 should denote "the processor is
--   not able to fulfill any further call request". Any value between
--   should in some way express a rough measure for the load of the
--   processor.
```

```
--
-- IN frame: the frame load of the processor
-- IN requ:  the request load of the processor
----------------------------------------------------------------------
INT FUNCTION Load(VAL INT frame, requ)

  --------------------------------------------------------------------
  -- Body of Load
  --------------------------------------------------------------------
  VALOF
    SKIP
    RESULT ((100 * frame) / (Frame.Len / 100)) + (0 * requ)
:

----------------------------------------------------------------------
-- Return Address of Minimal Loaded Processor
--
--   Returns the address of that processor that shall be considered as
--   minimal loaded. The local processor is preferred in the sense that
--   a neighbour processor must be significantly less loaded than the
--   local one in order to forward call requests to that processor.
--   The actual measure of 'significantly' is given by the constant
--   'Significance'.
--
-- ATTENTION: References global load descriptors!
----------------------------------------------------------------------
BYTE FUNCTION Min.Loaded()

  BYTE lmod: -- address of least loaded processor
  INT  pl:   -- load of this processor

  --------------------------------------------------------------------
  -- Body of Min.Loaded
  --------------------------------------------------------------------
  VALOF

    SEQ

      -- default is actual module
      lmod, pl := act.mod, (pLoad*Significance)/100

      -- search for less loaded module among neighbours
      SEQ i = 0 FOR Link.Num
        IF
          proc.load[i] <= pl
            SEQ
              lmod := mod.addr[i]
              pl := proc.load[i]
          TRUE
            SKIP

    RESULT lmod
:

----------------------------------------------------------------------
```

```
-- Determine Type of Processor Action
--
--   Using some strategy, this procedure returns the kind of processor
--   action to be executed next:
--   * execute instruction,
--   * receive message,
--   * fulfill request,
--   * report load.
--
-- IO action: last and next action to be executed
--
-- ATTENTION: references the global load descriptors!
------------------------------------------------------------------------
PROC Proc.Action(INT action)

  ----------------------------------------------------------------------
  -- Body of Proc.Action
  ----------------------------------------------------------------------
  IF
    trig.load > 0
      -- prefer instruction execution, if possible
      action := Execute.Act
    (pLoad > Proc.Trash) AND
      ((((100*lLoad) < (Rep.Delay*pLoad)) OR
        ((100*pLoad) < (Rep.Delay*lLoad))))
      -- let neighbours know, if load has significantly changed
      action := Report.Act
    mess.load = 0
      -- fulfill requests, if message store empty but
      -- give message store a chance for the next time
      SEQ
        action := Request.Act
        mess.load := 1
    requ.load = 0
      -- receive message, if request store empty but
      -- give request store a chance for the next time
      SEQ
        action := Receive.Act
        requ.load := 1
    (frame.load > Frame.Bound)
      -- receive message, if frame load too high
      action := Receive.Act
    (frame.load < Frame.Trash) OR
      ((frame.load < Frame.Bound) AND (requ.load > Requ.Bound))
      -- fulfill requests, if frame load too low or
      -- request store is quite filled
      action := Request.Act
    TRUE
      -- receiving a message is always possible
      action := Receive.Act

:

-----------------------------------------------------------------------
-- Context Heap:
```

```
--
--   The following procedures manage the heap-organized context store
--   consisting of the arrays 'cont.store.f' and 'cont.store.c':
--
--     * 'cont.store.f' holds the address of the function's frame
--     * 'cont.store.c' holds the address of the function's code block
--
--   The two corresponding entries of each store represent one context,
--   i.e. one node on the heap. All free nodes on the heap are linked by
--   their 'cont.store.f' fields, the variable 'cont.free' references
--   the first element in the free list.
------------------------------------------------------------------------

------------------------------------------------------------------------
-- Allocate New Context
--
--   Allocates a new context in context store and returns its address.
--
-- OUT context: address of new context
-- OUT done:    context could be allocated
--
-- ATTENTION: references and alters the global variables
--            cont.store, cont.free, cont.load
------------------------------------------------------------------------
PROC Alloc.Context(INT16 context, BOOL done)

  ----------------------------------------------------------------------
  -- Body of Alloc.Context
  ----------------------------------------------------------------------
  SEQ

    -- context store must not be full
    done := cont.free > (-1)

    IF
      done
        SEQ
          SEQ
            -- take first context in free list and update load
            context := INT16 cont.free
            cont.free := INT cont.store.f[cont.free]
            cont.load := cont.load+1
      TRUE
        SKIP
:

------------------------------------------------------------------------
-- Deallocate Context
--
--   Dellocates a context and stores it into a free list.
--
-- IN context: address of context to deallocate
--
-- ATTENTION: references and alters the global variables
--            cont.store, cont.free, cont.load
```

```
----------------------------------------------------------------------
PROC Dealloc.Context(VAL INT16 context)

  --------------------------------------------------------------------
  -- Body of Dealloc.Context
  --------------------------------------------------------------------
  SEQ
    -- store context in free list and update load
    cont.store.f[INT context] := INT32 cont.free
    cont.free := INT context
    cont.load := cont.load-1
:

----------------------------------------------------------------------
-- Frame Heap
--
--   The following procedures manage the heap-organized frame store
--   consisting of activation frames of variable size.
--
--   For heap management, the "buddy" algorithm is used that guarantees
--   a very fast allocation and deallocation of frames for the price that
--   every frame has a length that is a power of 2. In this particular
--   implementation, frame sizes may vary from at least 'Free.Min' words
--   to at most 'Free.Max' words.
--
--   Initially, the heap is split into blocks of 'Free.Max' words.
--   Vice versa, the reunification of deallocated blocks with their
--   buddies may lead to blocks of at most 'Free.Max' size.
----------------------------------------------------------------------

-----------------------------------------------------------------------
-- Allocate Activation Frame
--
--   Allocates a new activation frame for a specified number of
--   synchronization events and registers.
--
-- IN events: number of words neccessary to represent the event bits
-- IN regs:   number of registers required
-- OUT frame: address of activation frame (only if done = TRUE)
-- OUT done:  activation frame could be allocated
-----------------------------------------------------------------------
PROC Alloc.Frame(VAL INT events, regs, INT frame, BOOL done)

  INT i:  -- auxiliary index
  INT i0: -- index of list of best fitting blocks
  INT l:  -- required length of frame
  INT n:  -- actual length of block
  INT f:  -- frame address

  INT  buddy: -- index of buddy block
  INT  next:  -- index of successor of buddy block

  ---------------------------------------------------------------------
  -- Body of Alloc.Frame
  ---------------------------------------------------------------------
```

```
      SEQ

        -- required frame length (add two result address registers)
        l := (events+regs)+2

        -- length must not be to big
        IF
          l > Free.Max
            Assert(FALSE, frame.size)
          TRUE
            SKIP

        -- find frame of appropriate size
        i, n := 0, Free.Min
        WHILE (i < Free.Step) AND (n < l)
          i, n := i+1, n*2

        -- find fitting and free frame
        i0 := i
        WHILE (i < Free.Step) AND (frame.free[i] = (-1))
          i, n := i+1, n*2

        done := (i < Free.Step) AND (frame.free[i] > (-1))
        IF
          done

            -- free frame found
            SEQ

              -- extract block from doubly linked list
              frame := frame.free[i]
              f := INT frame.store[frame]
              frame.free[i] := f
              IF
                f > (-1)
                  frame.store[f+1] := -1(INT32)
                TRUE
                  SKIP

              WHILE i > i0
                SEQ

                  -- split block
                  i := i-1
                  n := n/2

                  -- compute address of second part
                  buddy := frame + n

                  -- link second part of block in free list
                  next := frame.free[i]
                  frame.free[i]        := buddy
                  frame.store[buddy]   := INT32 next
                  frame.store[buddy+1] := -1(INT32)
                  frame.store[buddy+2] := INT32 n
```

```
            IF
              next > (-1)
                frame.store[next+1] := INT32 buddy
              TRUE
                SKIP

        -- mark block as used (32 bits/word, addresses multiples of 8)
        INT32     slot IS used.store[frame / 256]:
        VAL INT32 bit  IS bits[(frame / 8) REM 32]:
        IF
          (slot /\ bit) = 0(INT32)
            slot := slot \/ bit
          TRUE
            STOP

        -- reset synchronization bits
        SEQ i = frame FOR events
          frame.store[i] := 0(INT32)

        -- update load
        frame.load := frame.load+n
        pLoad := Load(frame.load, requ.load)

        -- activation frame address is address of first register
        frame := (frame+events)+2

    TRUE

      -- no fitting frame available
      SKIP

:

------------------------------------------------------------------------
-- Deallocate Activation Frame
--
--   Deallocates activation frame with specified address, number of
--   synchronization events and registers.
--
-- IN frame:  -- address of frame (= address of first register of frame)
-- IN events: -- number of words reserved for synchronization events
-- IN regs:   -- number of registers reserved in frame
------------------------------------------------------------------------
PROC Dealloc.Frame(VAL INT frame, events, regs)

  INT  block: -- base address of block
  INT  buddy: -- address of buddy block
  INT  next:  -- successor of buddy
  INT  prev:  -- predecessor of buddy
  INT  len:   -- length of frame
  INT  i:     -- free list index
  INT  n:     -- length of block
  BOOL used:  -- buddy used?

  ----------------------------------------------------------------------
```

```
-- Body of Dealloc.Frame
----------------------------------------------------------------------
SEQ

  block := (frame-2)-events -- basis address of frame
  len   := (events+regs)+2  -- length of frame

  -- check if all synchronization events really occurred
  SEQ i = block FOR events
    IF
      frame.store[i] = 0(INT32)
        SKIP
      TRUE
        Assert(FALSE, synch.err)

  -- compute actual length of block
  i, n := 0, Free.Min
  WHILE (i < Free.Step) AND (n < len)
    i, n := i+1, n*2

  -- update load
  frame.load := frame.load - n
  pLoad := Load(frame.load, requ.load)

  -- block base address of block, n its length, i index in free list

  -- mark block as unused
  INT32     slot IS used.store[block / 256]:
  VAL INT32 bit  IS bits[(block / 8) REM 32]:
  IF
    (slot /\ bit) = 0(INT32)
      STOP
    TRUE
      slot := slot /\ (~bit)

  used := FALSE
  WHILE NOT used
    SEQ

      -- if buddy used, block has to be linked into free list
      IF
        i = (Free.Step-1)
          -- there is no buddy any more
          used := TRUE
        TRUE
          SEQ
            -- address of buddy
            buddy := INT ((INT32 block) >< bits[i+Free.Min.B])
            -- buddy used?
            INT32     slot IS used.store[buddy / 256]:
            VAL INT32 bit  IS bits[(buddy / 8) REM 32]:
            used := ((slot /\ bit) <> 0(INT32)) OR
              (frame.store[buddy+2] <> (INT32 n))

      IF
```

```
          used
            SEQ

              -- link block into free list
              next := frame.free[i]
              frame.free[i]     := block
              frame.store[block] := INT32 next
              frame.store[block+1] := -1(INT32)
              frame.store[block+2] := (INT32 n)
              IF
                next > (-1)
                  frame.store[next+1] := INT32 block
                TRUE
                  SKIP

          TRUE
            SEQ

              -- unlink buddy from free list
              next := INT frame.store[buddy]
              prev := INT frame.store[buddy+1]
              IF
                prev > (-1)
                  frame.store[prev] := INT32 next
                TRUE
                  frame.free[i] := next
              IF
                next > (-1)
                  frame.store[next+1] := INT32 prev
                TRUE
                  SKIP

              -- join block with buddy
              IF
                buddy < block
                  block := buddy
                TRUE
                  SKIP

              -- next index in free list and length of block
              i := i+1
              n := n*2

:

------------------------------------------------------------------------
-- Execute Instructions
--
--   This is the core of the interpreter. It fetches the actual
--   instruction from the program store, splits the instruction into
--   its operation code and its operand word and executes the denoted
--   operation. Then it continues with the successor instruction and
--   so on. The instruction sequence is interrupted, if a synchronization
--   event has not yet occurred or a request to the memory unit has
--   been submitted.
```

```
--
-- ATTENTION:
--
--   Although REFERENCES and also ALTERS the global variables defined
--   in the preamble of 'Execution.Unit'!
-------------------------------------------------------------------------
PROC Instruction()

  INT32 frame.base:  -- the base address of the actual activation frame
  INT32 code.base:   -- the base address of the actual code block
  INT32 trigger:     -- the actual trigger
  INT32 operation:   -- the actual instruction word
  INT   instruction: -- the actual instruction address
  BOOL  going:       -- termination variable

  -----------------------------------------------------------------------
  -- Body of Instruction
  -----------------------------------------------------------------------
  SEQ

    -- fetch trigger from trigger store
    Trig.Fetch(trigger)

    -- split it into its components context and offset
    [2]INT16 trig RETYPES trigger:
    INT16 context RETYPES trig[1]:
    INT16 offset  RETYPES trig[0]:

    SEQ

      -- convert context field to signed integer
      [2]INT16 n RETYPES null:
      n[0] := context

      -- fetch activation descriptors from context store
      frame.base := cont.store.f[INT null]
      code.base  := cont.store.c[INT null]

      -- compute actual instruction address
      instruction := (INT code.base) + (INT offset)

      going := TRUE
      WHILE going
        SEQ

          -- fetch instruction from program store
          operation := prog.store[instruction]

          -- split instruction into its components
          [4]BYTE op      RETYPES operation:
          BYTE    op.code IS op[3]:
          [3]BYTE op.word IS [op FROM 0 FOR 3]:

          -- execute instruction
          #INCLUDE "adexec.inc"
```

```

:

------------------------------------------------------------------------
-- Receive Messages
--
--   Receives
--   * value messages, stores the transmitted values in the denoted
---    activation frame registers and triggers the associated
--     instructions,
--   * context messages, stores the transmitted result addresses in the
--     denoted activation frame and triggers the first instruction of
--     the associated code block,
--   * load reports and updates the associated load variables.
--
-- ATTENTION:
--
--   This procedure REFERENCES and ALTERS the global variables defined
--   in the preamble of ‘Execution.Unit’!
------------------------------------------------------------------------
PROC Reception()

  [Token.Size]INT32 token:    -- token value

  INT32 instruct:   -- instruction
  INT32 frame.base: -- basis address of activation frame
  INT32 code.base:  -- basis address of code block
  INT32 trigger:    -- instruction trigger

  ----------------------------------------------------------------------
  -- Body of Reception
  ----------------------------------------------------------------------
  SEQ

    -- fetch message from buffer
    demand.mess ! demand.tag
    from.mess ? mess.load; token

    IF
      mess.load = 0

        -- buffer empty, no message received
        SKIP

      TRUE
        SEQ

          -- select token type
          [4]BYTE  toks  RETYPES token[2]:
          BYTE     mType  IS toks[1]:              -- message type
          [2]INT32 mData IS [token FROM 0 FOR 2]:  -- message data

          CASE mType

            ------------------------------------------------------------
```

```
-- stop token received
------------------------------------------------------------
Stop.Token

  -- empty request buffer and terminate
  [Token.Size]INT32 token0:   -- auxiliary token
  WHILE going
    SEQ
      demand.requ ! demand.tag
      from.requ ? mess.load; token0
      [4]BYTE toks RETYPES token0[2]:
      BYTE    type IS toks[1]:
      going := type <> Stop.Token

------------------------------------------------------------
-- value message received
------------------------------------------------------------
Mess.PToken
  SEQ

    -- split token value into components
    [2]INT16 toks RETYPES mData[1]:
    INT16 context IS toks[1]:
    INT16 offset  IS toks[0]:
    INT32 data    IS mData[0]:
    SEQ

      -- convert context field to signed integer
      [2]INT16 n RETYPES null:
      n[0] := context

      -- fetch base register values
      frame.base := cont.store.f[INT null]
      code.base  := cont.store.c[INT null]

      -- convert offset field to signed integer
      [2]INT16 n RETYPES null:
      n[0] := offset

      -- fetch instruction word
      instruct := prog.store[(INT code.base)+(INT null)]

      -- write message value into register
      [4]BYTE inst RETYPES instruct:
      BYTE    op   IS inst[1]:
      frame.store[(INT frame.base) + (INT op)] := data

    -- put trigger into store
    Trig.Put(mData[1])

------------------------------------------------------------
-- context message received
------------------------------------------------------------
Cont.PToken
  SEQ
```

```

                -- split token value into components
                INT32    con.off IS mData[1]:
                [2]INT16 toks    RETYPES mData[0]:
                [2]BYTE  tok     RETYPES toks[1]:
                BYTE     mod     IS tok[1]:
                INT16    context IS toks[0]:

                SEQ

                  -- convert context field to signed integer
                  [2]INT16 n RETYPES null:
                  n[0] := context

                  -- fetch description of activation
                  frame.base := cont.store.f[INT null]
                  code.base  := cont.store.c[INT null]

                  -- store return address in activation frame
                  frame.store[(INT frame.base)-1] := con.off
                  frame.store[(INT frame.base)-2] := INT32 mod

                  -- put trigger into store
                  [2]INT16 trig RETYPES trigger:
                  SEQ
                    trig[1] := context
                    trig[0] := 0(INT16)
                  Trig.Put(trigger)

            -------------------------------------------------------------
            -- processor load message received
            -------------------------------------------------------------
            PLoad.PToken

              -- update load variables
              BYTE link IS toks[0]:
              INT32 fload IS mData[1]: -- frame load
              INT32 rload IS mData[0]: -- request load
              SEQ

                -- store relative processor load
                proc.load[INT link] := Load(INT fload, INT rload)

                -- currently not used
                -- proc.frame.load[link] := fload
                -- proc.requ.load[link]  := rload

:

-------------------------------------------------------------------------
-- Fulfill Call Requests
--
--   Fulfills the next call request in request store, if possible.
--   Allocates activation frame and context and triggers start of
--   the execution, if required.
```

```
--
--   If a stop request is encountered, the termination variable 'going'
--   is set to FALSE. By this, the Execution.Unit is reset to its
--   initial state ready to receive a new program.
--
-- ATTENTION:
--
--   This procedure REFERENCES and ALTERS variables that have been
--   globally defined in the preamble of Execution.Unit!
-----------------------------------------------------------------------
PROC Request()

  INT32 descr:   -- function descriptor
  INT32 code:    -- code block address
  INT32 ret:     -- return address
  INT32 trig:    -- instruction trigger
  INT16 context: -- context address
  INT   frame:   -- activation frame address
  BOOL  done:    -- allocation successful

  [Token.Size]INT32 token:  -- input token
  [Token.Size]INT32 token0: -- output token

  ---------------------------------------------------------------------
  -- Body of Request
  ---------------------------------------------------------------------
  SEQ

    -- fetch request from buffer
    demand.requ ! demand.tag
    from.requ ? requ.load; token
    token0 := token

    -- update processor load
    pLoad := Load(frame.load, requ.load)

    IF

      requ.load = 0

        -- buffer empty, no request received
        SKIP

      TRUE

        -- request received
        SEQ

          -- select token type
          [4]BYTE  toks  RETYPES token[2]:
          BYTE     rType IS toks[1]:              -- request type
          [2]INT32 rData IS [token FROM 0 FOR 2]:  -- request data

          IF

```

```
-- stop request encountered
rType = Stop.Token

  -- empty message buffer and terminate
  WHILE going
    SEQ
      demand.mess ! demand.tag
      from.mess ? requ.load; token0
      [4]BYTE toks RETYPES token0[2]:
      BYTE    type IS toks[1]:
      going := type <> Stop.Token

-- call request encountered
TRUE

  -- select arguments
  INT32   contoff IS rData[1]:              -- return address
  [4]BYTE modfun  RETYPES rData[0]:
  BYTE    mod     IS modfun[3]:             -- return module
  [2]BYTE fun     IS [modfun FROM 0 FOR 2]: -- function

  SEQ

    -- construct function descriptor
    descr := 0(INT32)
    [4]BYTE d RETYPES descr:
    [d FROM 0 FOR 2] := fun

    -- fetch code block address
    code := prog.store[INT descr]

    -- fetch resource descriptors
    [4]BYTE evreg  RETYPES prog.store[(INT code)-1]:
    VAL INT events IS INT evreg[1]:
    VAL INT regs   IS INT evreg[0]:

    -- Allocate context and frame
    SEQ
      Alloc.Context(context, done)
      IF
        done
          SEQ
            Alloc.Frame(events, regs, frame, done)
            IF
              done
                SKIP
              TRUE
                Dealloc.Context(context)
        TRUE
          SKIP

    IF

      -- allocation okay
      done
```

```
-- activate function
SEQ

  -- initialize context
  cont.store.f[INT context] := INT32 frame
  cont.store.c[INT context] := code

  CASE rType

    -- fast call
    FCall.PToken

      -- result address is available
      SEQ

        -- construct result address
        [2]INT16 conts RETYPES contoff:
        [2]INT16 rets  RETYPES ret:
        SEQ
          rets[1] := conts[1]           -- context
          rets[0] := conts[0]+1(INT16)  -- offset

        -- store result address
        frame.store[(INT frame)-2] := INT32 mod
        frame.store[(INT frame)-1] := ret

        -- construct trigger
        [2]INT16 trigs RETYPES trig:
        SEQ
          trigs[1] := context          -- context
          trigs[0] := 0(INT16)         -- offset

        -- release trigger
        Trig.Put(trig)

    -- tail call
    TCall.PToken

      -- result address will be sent later
      SKIP

    -- error in token type
    ELSE

      Assert(FALSE, requ.err)

  -- return context address to caller

  -- construct token
  [4]BYTE toks RETYPES token0[2]:
  SEQ
    toks[2] := mod                -- module address
    toks[1] := Mess.PToken        -- message token
  token0[1] := contoff            -- return address
```

```
                      [4]BYTE toks RETYPES token0[0]:
                      toks[2] := act.mod              -- actual module
                      [2]INT16 toks RETYPES token0[0]:
                      toks[0] := context              -- context address
                      token0[0] := token0[0] << 1     -- error flag

                      -- send token to processor unit
                      out ! net.token; token0

                  -- too little memory for function activation
                  TRUE

                    -- forward call request
                    SEQ

                      -- enter address of minimal loaded module
                      [4]BYTE toks RETYPES token0[2]:
                      toks[2] := Min.Loaded()

                      -- send token to processor unit
                      out ! net.token; token0

:

------------------------------------------------------------------------
-- Report Processor Load:
--
--   Tells the processors on all neighbour modules and the master
--   module about the current workload of frame store and request store.
--
-- ATTENTION:
--
--   This procedure REFERENCES and ALTERS global variables defined
--   in the preamble of Execution.Unit
------------------------------------------------------------------------
PROC Report()

  [Token.Size]INT32 token: -- network token

  ----------------------------------------------------------------------
  -- Body of Report
  ----------------------------------------------------------------------
  SEQ

    -- store load to be reported
    lLoad := pLoad

    -- construct token
    [4]BYTE toks RETYPES token[2]:
    toks[1] := PLoad.PToken        -- processor load token
    token [1] := INT32 frame.load  -- frame store load
    token [0] := INT32 requ.load   -- request store load

    -- send report to master
    [4]BYTE toks RETYPES token[2]:
```

```
    SEQ
      toks[2] := 0(BYTE)           -- address of neighbour
      toks[0] := act.mod           -- actual module address
    out ! net.token; token

    -- send report to neighbours
    SEQ i=0 FOR Link.Num
      SEQ
        [4]BYTE toks RETYPES token[2]:
        SEQ
          toks[2] := mod.addr[i]   -- address of neighbour
          toks[0] := link.addr[i]  -- unique processor idenitifier
        out ! net.token; token

:

------------------------------------------------------------------------
-- Body of Execution.Unit
------------------------------------------------------------------------
SEQ

  -- initialize global conversion auxiliary
  null := 0(INT32)

  -- loop forever
  WHILE TRUE
    SEQ

      --------------------------------------------------------------------
      -- initializations
      --------------------------------------------------------------------

      -- initialize load descriptors of neighbour processors
      SEQ i=0 FOR Link.Num
        SEQ
          proc.load[i] := 0
          -- currently not used
          -- proc.frame.load[i], proc.requ.load[i] := 0, 0

      -- initialize local load descriptors
      frame.load, cont.load, trig.load, requ.load, mess.load :=
        0, 0, 0, 0, 0

      -- initialize relative processor load
      pLoad, lLoad := 0, 0

      -- initialize context store heap
      cont.free := 0
      SEQ i = 0 FOR Cont.Len
        cont.store.f[i] := INT32(i+1)
      cont.store.f[Cont.Len-1] := -1(INT32)

      -- initialize frame store heap
      SEQ i = 0 FOR Used.Len
        used.store[i] := 0(INT32)         -- no activation frame used
```

```
SEQ i = 0 FOR Free.Step-1
  frame.free[i] := -1               -- all free lists empty
frame.free[Free.Step-1] := 0        --   except the last one
SEQ i = 0 FOR Frame.Len / Free.Max  -- split store in frames
  VAL j IS i*Free.Max:              --   and link them in
  SEQ                               --   both directions
    frame.store[j]   := INT32 (j+Free.Max)
    frame.store[j+1] := INT32 (j-Free.Max)
    frame.store[j+2] := INT32 Free.Max
frame.store[0+1] := -1(INT32)       -- handle bounds of frame store
frame.store[Frame.Len-Free.Max] := -1(INT32)

-- assume start signal
pos, max.pos, append := 0, 0, FALSE

--------------------------------------------------------------------
-- receive program
--------------------------------------------------------------------

-- receive program clusters
going := TRUE
WHILE going
  from.rout ? CASE
    start.cluster            -- a new program will arrive
      SEQ
        pos := 0
        max.pos := 0
        append := FALSE
    append.cluster           -- a program part is to be appended
      SEQ                    -- store address of appended part
        prog.store[0] := INT32 (max.pos+1)
        pos := max.pos
        append := TRUE       -- receive program cluster
    cluster; [prog.store FROM pos FOR Cluster.Size]
      SEQ
        pos := pos+Cluster.Size
        Assert(pos<Prog.Len, prog.full)
        IF
          append
            SKIP
          TRUE
            max.pos := max.pos+Cluster.Size
    end.cluster              -- end of program signalled
      going := FALSE

-- send acknowledge signal
out ! net.ack

--------------------------------------------------------------------
-- interpretation loop
--------------------------------------------------------------------
going := TRUE
WHILE going
  SEQ

```

```
            ------------------------------------------------------------
            -- ATTENTION:
            --
            --   Although the following procedures are not given explicit
            --   arguments, they REFERENCE the variables defined in the
            --   preamble of procedure 'Execution.Unit' and also ALTER
            --   their values!
            ------------------------------------------------------------

            -- choose next action
            Proc.Action(action)

            CASE action
              Execute.Act  -- Execute Instruction
                Instruction()
              Receive.Act  -- Receive Message
                Reception()
              Request.Act  -- Fulfill Request
                Request()
              Report.Act   -- Report Processor Load
                Report()

        -- send acknowledge signal
        out ! net.ack

:

--------------------------------------------------------------------------
-- Body of Adam.Processor
--------------------------------------------------------------------------
PAR

  -- routes incoming tokens to request or to message store,
  -- the incoming program to execution unit
  Router(in, to.mess, to.requ, from.rout, error[0])

  -- buffers incoming processor requests and forwards them
  -- on demand to execution unit
  Buffer(to.requ, demand.requ, requ.store, from.requ, error[1])

  -- buffers incoming processor messages and forwards them
  -- on demand to execution unit
  Buffer(to.mess, demand.mess, mess.store, from.mess, error[2])

  -- receives program and executes it,
  -- handles requests and messages
  Execution.Unit(from.rout, from.requ, from.mess,
    demand.requ, demand.mess, error[3])

  -- multiplexor for error messages
  INT len:               -- length of message (< Max.Debug!)
  [Max.Debug]BYTE mess:  -- error message
  ALT i=0 FOR 4
    error[i] ? len :: mess
      debug ! len :: mess
```

```

:
--------------------------------------------------------------------------------
-- End of ADPROC.OCC
--------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
--
-- ADMODULE.OCC                                         Wolfgang Schreiner
-- A Module of the AdaM Network                         (c) RISC-Linz 1990
--
-- Connect the units within an Adam module
--
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- External References
-------------------------------------------------------------------------------

-- include files
#INCLUDE "aderror.inc"  -- error messages
#INCLUDE "adglobal.inc" -- global constants and protocols
#INCLUDE "adnet.inc"    -- network constants and protocols

-- compilation units
#USE "adarbit.t8h"  -- arbiter unit
#USE "adproc.t8h"   -- processor unit
#USE "admemory.t8h" -- memory unit

-------------------------------------------------------------------------------
-- The Adam Module:
--
--   The arbiter routes an incoming network token (with respect to its
--   destination address) either to the processor respectively the memory
--   unit or forwards it to one of the neighbour modules. It is also
--   responsible for the correct distribution of the program before the
--   start of the execution.
--
--   The processor performs the actual program execution by receiving
--   call requests, activating the desired functions, computing the results
--   of this activation and delivering them to the sender of the request.
--
--   The memory holds all structured data in a heap organized store. The
--   processor has no direct access to this unit. It has to send requests
--   to the store and will then receive the desired results.
--
--   IN  act.mod:   address of actual module
--   IN  mod.addr:  addresses of neighbour modules
--   IN  link.addr: link addresses of actual module from view of neighbours
--   IN  route:     link addresses of all modules in the network (inc. master)
--   IN  boot.from: link address from which program is received
--   IN  boot.to:   signals to which links program is to be forwarded
--   IN  in:        input channels from neighbour modules
--   OUT out:       output channels to neighbour modules
-------------------------------------------------------------------------------
PROC Adam.Module(VAL BYTE act.mod, VAL [Link.Num]BYTE mod.addr,
  VAL [Link.Num]BYTE link.addr, VAL [Mod.Num+1]INT route,
  VAL INT boot.from, VAL[Link.Num]BOOL boot.to,
  [Link.Num]CHAN OF Net.Link in, out)

  CHAN OF Net.Link from.proc, from.mem: -- arbiter channels from local units
```

```
  CHAN OF Mod.Link to.proc, to.mem:     -- arbiter channels to local units

  [3]CHAN OF Debug debug: -- debug channels

  ---------------------------------------------------------------------------
  -- Body of Adam.Module
  ---------------------------------------------------------------------------
  PAR

    -- the arbiter
    Adam.Arbiter(act.mod, boot.from, boot.to, route,
      in, out, from.proc, to.proc, from.mem, to.mem, debug[0])

    -- the processor unit
    Adam.Processor(act.mod, mod.addr, link.addr,
      to.proc, from.proc, debug[1])

    -- the memory unit
    Adam.Memory(act.mod, mod.addr, link.addr,
      to.mem, from.mem, debug[2])

    -- debug process
    INT len:
    [80]BYTE error:
    ALT i=0 FOR 3
      debug[i] ? len :: error
        STOP

:

-----------------------------------------------------------------------------
-- End of ADMODULE.OCC
-----------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------
--
-- ADARBIT.INC                                            Wolfgang Schreiner
-- Arbiter Constants and Protocols                        (c) RISC-Linz 1990
--
-- Defines constants and protocols that are used in the arbiter
-- of the Adam module.
--
----------------------------------------------------------------------------

--constants
VAL INT Arbit.Size IS 5000: -- length of arbiter buffer in tokens

-- protocols
PROTOCOL Permit
  CASE
    write.permit          -- write permission signal
:

----------------------------------------------------------------------------
-- End of ADARBIT.INC
----------------------------------------------------------------------------
```

ADEXEC.INC Page 1

```
-----------------------------------------------------------------------------
--
-- ADEXEC.INC                                          Wolfgang Schreiner
-- Execution of Adam Machine Code                      (c) RISC-Linz 1990
--
-- Contains the piece of code that interprets the various instructions of
-- the ADAM machine language.
--
-- For reasons of efficiency, this piece of code is textually included into
-- the procedure `Instruction' (file "ADPROC.OCC") and not compiled as a
-- seperate compilation unit.
--
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- Integer Absolute Value (only works for x values > MOSTNEG!)
-----------------------------------------------------------------------------
INT FUNCTION IABS(VAL INT x)
  INT y:
  VALOF
    IF
      x < 0
        y := - x  -- no overflow, since only called for x > MOSTNEG
      TRUE
        y := x
    RESULT y
:

-----------------------------------------------------------------------------
-- Select Instruction
-----------------------------------------------------------------------------
CASE op.code


  ---------------------------------------------------------------------------
  -- Instruction ADD
  ---------------------------------------------------------------------------
  Add.OpC

    -- components of operand word
    BYTE regL IS op.word[2]: -- left register operand
    BYTE regR IS op.word[1]: -- right register operand
    BYTE succ IS op.word[0]: -- successor instruction

    -- local variables
    INT   opL, opR:   -- operand adresses
    INT32 infL, infR: -- operand values
    INT   datL, datR: -- data parts of operands
    BOOL  errL, errR: -- error flags of operands
    INT32 result:     -- result of operation

    SEQ -- instruction ADD

      -- select operands
      opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
```

```
    infL, infR := frame.store[opL], frame.store[opR]
    datL, datR := ASHIFTRIGHT(INT infL, 1), ASHIFTRIGHT(INT infR, 1)
    errL, errR := BOOL(infL /\ 1(INT32)), BOOL(infR /\ 1(INT32))

    -- compute results
    IF
      errL
        -- left operand invalid, keep error code
        result := infL
      errR
        -- right operand invalid, keep error code
        result := infR
      TRUE
        SEQ
          -- add values (no overflow may occur!)
          result := INT32 (datL + datR)
          IF
            (result < Min.Int) OR (result > Max.Int)
              -- integer overflow
              result := Error.Overflow
            TRUE
              -- generate error bit 0
              result := result << 1

    -- store results
    frame.store[opL] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)

---------------------------------------------------------------------------
-- Instruction ADDC
---------------------------------------------------------------------------
AddC.OpC

  -- components of operand word
  BYTE con  IS op.word[2]: -- constant operand
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   op:     -- operand adress
  INT32 inf:    -- operand value
  INT   dat:    -- data parts of operand
  BOOL  err:    -- error flag of operand
  INT32 result: -- result of operation
  INT   scon:   -- signed constant operand

  SEQ -- instruction ADDC

    -- select register operand
    op  := (INT frame.base)+(INT reg)
    inf := frame.store[op]
    dat := ASHIFTRIGHT(INT inf, 1)
    err := BOOL(inf /\ 1(INT32))
```

```

    -- compute results
    IF
      err
        -- operand invalid, keep error code
        result := inf
      TRUE
        SEQ
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
          -- add values (no overflow may occur!)
          result := INT32 (dat + scon)
          IF
            (result < Min.Int) OR (result > Max.Int)
              -- integer overflow
              result := Error.Overflow
            TRUE
              -- generate error bit 0
              result := result << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


----------------------------------------------------------------------------
-- Instruction SUB
----------------------------------------------------------------------------
Sub.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR:   -- operand adresses
  INT32 infL, infR: -- operand values
  INT   datL, datR: -- data parts of operands
  BOOL  errL, errR: -- error flags of operands
  INT32 result:     -- result of operation

  SEQ -- instruction SUB

    -- select operands
    opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
    infL, infR := frame.store[opL], frame.store[opR]
    datL, datR := ASHIFTRIGHT(INT infL, 1), ASHIFTRIGHT(INT infR, 1)
    errL, errR := BOOL(infL /\ 1(INT32)), BOOL(infR /\ 1(INT32))
```

```

    -- compute results
    IF
      errL
        -- left operand invalid, keep error code
        result := infL
      errR
        -- right operand invalid, keep error code
        result := infR
      TRUE
        SEQ
          -- subtract values (no overflow may occur!)
          result := INT32 (datL - datR)
          IF
            (result < Min.Int) OR (result > Max.Int)
              -- integer overflow
              result := Error.Overflow
            TRUE
              -- generate error bit 0
              result := result << 1

    -- store results
    frame.store[opL] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


----------------------------------------------------------------------------
-- Instruction SUBC
----------------------------------------------------------------------------
SubC.OpC

  -- components of operand word
  BYTE con  IS op.word[2]: -- constant operand
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   op:     -- operand adress
  INT32 inf:    -- operand value
  INT   dat:    -- data parts of operand
  BOOL  err:    -- error flag of operand
  INT   scon:   -- signed constant operand
  INT32 result: -- result of operation

  SEQ -- instruction SUBC

    -- select register operand
    op  := (INT frame.base)+(INT reg)
    inf := frame.store[op]
    dat := ASHIFTRIGHT(INT inf, 1)
    err := BOOL(inf /\ 1(INT32))

    -- compute results
```

```
    IF
      err
        -- operand invalid, keep error code
        result := inf
      TRUE
        SEQ
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
          -- subtract values (no overflow may occur!)
          result := INT32 (dat - scon)
          IF
            (result < Min.Int) OR (result > Max.Int)
              -- integer overflow
              result := Error.Overflow
            TRUE
              -- generate error bit 0
              result := result << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


---------------------------------------------------------------------------
-- Instruction SUBRC
---------------------------------------------------------------------------
SubRC.OpC

  -- components of operand word
  BYTE con  IS op.word[2]: -- constant operand
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   op:     -- operand adress
  INT32 inf:    -- operand value
  INT   dat:    -- data parts of operand
  BOOL  err:    -- error flag of operand
  INT   scon:   -- signed constant operand
  INT32 result: -- result of operation

  SEQ -- instruction SUBRC

    -- select register operand
    op  := (INT frame.base)+(INT reg)
    inf := frame.store[op]
    dat := ASHIFTRIGHT(INT inf, 1)
    err := BOOL(inf /\ 1(INT32))

```

```
    -- compute results
    IF
      err
        -- operand invalid, keep error code
        result := inf
      TRUE
        SEQ
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
          -- subtract values (no overflow may occur!)
          result := INT32 (scon - dat)
          IF
            (result < Min.Int) OR (result > Max.Int)
              -- integer overflow
              result := Error.Overflow
            TRUE
              -- generate error bit 0
              result := result << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


---------------------------------------------------------------------------
-- Instruction MUL
---------------------------------------------------------------------------
Mul.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR:   -- operand adresses
  INT32 infL, infR: -- operand values
  INT   datL, datR: -- data parts of operands
  BOOL  errL, errR: -- error flags of operands
  INT   high, low:  -- result words of multiplication
  INT32 result:     -- result of operation

  SEQ -- instruction MUL

    -- select operands
    opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
    infL, infR := frame.store[opL], frame.store[opR]
    datL, datR := ASHIFTRIGHT(INT infL, 1), ASHIFTRIGHT(INT infR, 1)
    errL, errR := BOOL(infL /\ 1(INT32)), BOOL(infR /\ 1(INT32))
```

```

    -- compute results
    IF
      errL
        -- left operand invalid, keep error code
        result := infL
      errR
        -- right operand invalid, keep error code
        result := infR
      TRUE
        SEQ
          -- multiply operands
          high, low := LONGPROD(IABS(datL), IABS(datR), 0)
          IF
            high <> 0
              -- product out of range
              result := Error.Overflow
            ((datL > 0) AND (datR > 0)) OR ((datL < 0) AND (datR < 0))
              -- product is positive
              IF
                (INT32 low) > Max.Int
                  result := Error.Overflow
                TRUE
                  result := (INT32 low) << 1
            TRUE
              -- product is negative
              IF
                (INT32 low) > Min.Int.Abs
                  result := Error.Overflow
                TRUE
                  result := (INT32 (- low)) << 1

    -- store results
    frame.store[opL] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


------------------------------------------------------------------------
-- Instruction MULC
------------------------------------------------------------------------
MulC.OpC

  -- components of operand word
  BYTE con  IS op.word[2]: -- constant operand
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   op:         -- operand adress
  INT32 inf:        -- operand value
  INT   dat:        -- data part of operand
  BOOL  err:        -- error flag of operand
  INT   scon:       -- signed constant operand
```

```
  INT   high, low:  -- result words of multiplication
  INT32 result:     -- result of operation

  SEQ -- instruction MULC

    -- select operands
    op  := (INT frame.base)+(INT reg)
    inf := frame.store[op]
    dat := ASHIFTRIGHT(INT inf, 1)
    err := BOOL(inf /\ 1(INT32))

    -- compute results
    IF
      err
        -- operand invalid, keep error code
        result := inf
      TRUE
        SEQ
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
          -- multiply operands
          high, low := LONGPROD(IABS(dat), IABS(scon), 0)
          IF
            high <> 0
              -- product out of range
              result := Error.Overflow
            ((dat > 0) AND (scon > 0)) OR ((dat < 0) AND (scon < 0))
              -- product is positive
              IF
                (INT32 low) > Max.Int
                  result := Error.Overflow
                TRUE
                  result := (INT32 low) << 1
            TRUE
              -- product is negative
              IF
                (INT32 low) > Min.Int.Abs
                  result := Error.Overflow
                TRUE
                  result := (INT32 (- low)) << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


---------------------------------------------------------------------------
-- Instruction DIV
---------------------------------------------------------------------------
```

```
Div.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR:   -- operand adresses
  INT32 infL, infR: -- operand values
  INT   datL, datR: -- data parts of operands
  BOOL  errL, errR: -- error flags of operands
  INT32 result:     -- result of operation

  SEQ -- instruction DIV

    -- select operands
    opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
    infL, infR := frame.store[opL], frame.store[opR]
    datL, datR := ASHIFTRIGHT(INT infL, 1), ASHIFTRIGHT(INT infR, 1)
    errL, errR := BOOL(infL /\ 1(INT32)), BOOL(infR /\ 1(INT32))

    -- compute results
    IF
      errL
        -- left operand invalid, keep error code
        result := infL
      errR
        -- right operand invalid, keep error code
        result := infR
      datR = 0
        -- division by zero
        result := Error.ZeroDiv
      TRUE
        SEQ
          -- divide operands (no overflow may occur!)
          result := INT32 (datL / datR)
          IF
            (result < Min.Int) OR (result > Max.Int)
              -- integer overflow
              result := Error.Overflow
            TRUE
              -- generate error bit 0
              result := result << 1

    -- store results
    frame.store[opL] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


---------------------------------------------------------------------------
-- Instruction DIVC
---------------------------------------------------------------------------
```

```
DivC.OpC

  -- components of operand word
  BYTE con  IS op.word[2]: -- constant operand
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   op:          -- operand adress
  INT32 inf:         -- operand value
  INT   dat:         -- data part of operand
  BOOL  err:         -- error flag of operand
  INT   scon:        -- signed constant operand
  INT32 result:      -- result of operation

  SEQ -- instruction DIVC

    -- select operands
    op  := (INT frame.base)+(INT reg)
    inf := frame.store[op]
    dat := ASHIFTRIGHT(INT inf, 1)
    err := BOOL(inf /\ 1(INT32))

    -- compute results
    IF
      err
        -- operand invalid, keep error code
        result := inf
      TRUE
        SEQ
          -- constant must not be zero
          Assert(con > 0(BYTE), const.zero)
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
          -- divide operands (no overflow may occur!)
          result := INT32 (dat / scon)
          IF
            (result > Max.Int)
              -- integer overflow (can only happen in divisions by -1)
              result := Error.Overflow
            TRUE
              -- generate error bit 0
              result := result << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)
```

```
------------------------------------------------------------------------
-- Instruction DIVRC
------------------------------------------------------------------------
DivRC.OpC

  -- components of operand word
  BYTE con  IS op.word[2]: -- constant operand
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   op:         -- operand adress
  INT32 inf:        -- operand value
  INT   dat:        -- data part of operand
  BOOL  err:        -- error flag of operand
  INT   scon:       -- signed constant operand
  INT32 result:     -- result of operation

  SEQ -- instruction DIVRC

    -- select operands
    op  := (INT frame.base)+(INT reg)
    inf := frame.store[op]
    dat := ASHIFTRIGHT(INT inf, 1)
    err := BOOL(inf /\ 1(INT32))

    -- compute results
    IF
      err
        -- operand invalid, keep error code
        result := inf
      dat = 0
        -- division by zero
        result := Error.ZeroDiv
      TRUE
        SEQ
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
          -- divide operands (no overflow may occur!)
          result := INT32 (scon / dat)
          -- no integer overflow may occur
          -- generate error bit 0
          result := result << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


```

```
---------------------------------------------------------------------------
-- Instruction MOD
---------------------------------------------------------------------------
Mod.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR:   -- operand adresses
  INT32 infL, infR: -- operand values
  INT   datL, datR: -- data parts of operands
  BOOL  errL, errR: -- error flags of operands
  INT32 result:     -- result of operation

  SEQ -- instruction MOD

    -- select operands
    opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
    infL, infR := frame.store[opL], frame.store[opR]
    datL, datR := ASHIFTRIGHT(INT infL, 1), ASHIFTRIGHT(INT infR, 1)
    errL, errR := BOOL(infL /\ 1(INT32)), BOOL(infR /\ 1(INT32))

    -- compute results
    IF
      errL
        -- left operand invalid, keep error code
        result := infL
      errR
        -- right operand invalid, keep error code
        result := infR
      datR = 0
        -- remainder modulo zero
        result := Error.ZeroDiv
      TRUE
        -- compute remainder and generate error bit 0
        result := (INT32 (infL REM infR)) << 1

    -- store results
    frame.store[opL] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


---------------------------------------------------------------------------
-- Instruction MODC
---------------------------------------------------------------------------
ModC.OpC

  -- components of operand word
  BYTE con  IS op.word[2]: -- constant operand
  BYTE reg  IS op.word[1]: -- register operand
```

```
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   op:     -- operand adress
  INT32 inf:    -- operand value
  INT   dat:    -- data part of operand
  BOOL  err:    -- error flag of operand
  INT   scon:   -- signed constant operand
  INT32 result: -- result of operation

  SEQ -- instruction MODC

    -- select operands
    op  := (INT frame.base)+(INT reg)
    inf := frame.store[op]
    dat := ASHIFTRIGHT(INT inf, 1)
    err := BOOL(inf /\ 1(INT32))

    -- compute results
    IF
      err
        -- register operand invalid, keep error code
        result := inf
      TRUE
        SEQ
          -- constant must not be zero
          Assert(con > 0(BYTE), const.zero)
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
          -- compute remainder and generate error bit 0
          result := (INT32 (dat REM scon)) << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


---------------------------------------------------------------------------
-- Instruction MODRC
---------------------------------------------------------------------------
ModRC.OpC

  -- components of operand word
  BYTE con  IS op.word[2]: -- constant operand
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   op:     -- operand adress
```

```
  INT32 inf:    -- operand value
  INT   dat:    -- data part of operand
  BOOL  err:    -- error flag of operand
  INT   scon:   -- signed constant operand
  INT32 result: -- result of operation

  SEQ -- instruction MODRC

    -- select operands
    op  := (INT frame.base)+(INT reg)
    inf := frame.store[op]
    dat := ASHIFTRIGHT(INT inf, 1)
    err := BOOL(inf /\ 1(INT32))

    -- compute results
    IF
      err
        -- register operand invalid, keep error code
        result := inf
      dat = 0
        -- remainder by zero
        result := Error.ZeroDiv
      TRUE
        SEQ
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
          -- compute remainder and generate error bit 0
          result := (INT32 (scon REM dat)) << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


---------------------------------------------------------------------------
-- Instruction AND
---------------------------------------------------------------------------
And.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR:   -- operand adresses
  INT32 infL, infR: -- operand values
  INT   datL, datR: -- data parts of operands
  BOOL  errL, errR: -- error flags of operands
```

```
    INT32 result:     -- result of operation

    SEQ -- instruction AND

      -- select operands
      opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
      infL, infR := frame.store[opL], frame.store[opR]
      datL, datR := ASHIFTRIGHT(INT infL, 1), ASHIFTRIGHT(INT infR, 1)
      errL, errR := BOOL(infL /\ 1(INT32)), BOOL(infR /\ 1(INT32))

      -- compute results
      IF
        errL
          -- left operand invalid, keep error code
          result := infL
        errR
          -- right operand invalid, keep error code
          result := infR
        TRUE
          -- compute conjunction and generate error bit 0
          result := (INT32 (datL /\ datR)) << 1

      -- store results
      frame.store[opL] := result

      -- go to successor instruction
      instruction := instruction + (INT succ)


  --------------------------------------------------------------------------
  -- Instruction OR
  --------------------------------------------------------------------------
  Or.OpC

    -- components of operand word
    BYTE regL IS op.word[2]: -- left register operand
    BYTE regR IS op.word[1]: -- right register operand
    BYTE succ IS op.word[0]: -- successor instruction

    -- local variables
    INT   opL, opR:   -- operand adresses
    INT32 infL, infR: -- operand values
    INT   datL, datR: -- data parts of operands
    BOOL  errL, errR: -- error flags of operands
    INT32 result:     -- result of operation

    SEQ -- instruction OR

      -- select operands
      opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
      infL, infR := frame.store[opL], frame.store[opR]
      datL, datR := ASHIFTRIGHT(INT infL, 1), ASHIFTRIGHT(INT infR, 1)
      errL, errR := BOOL(infL /\ 1(INT32)), BOOL(infR /\ 1(INT32))

      -- compute results
```

```
    IF
      errL
        -- left operand invalid, keep error code
        result := infL
      errR
        -- right operand invalid, keep error code
        result := infR
      TRUE
        -- compute disjunction and generate error bit 0
        result := (INT32 (datL \/ datR)) << 1

    -- store results
    frame.store[opL] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


---------------------------------------------------------------------------
-- Instruction EQUAL
---------------------------------------------------------------------------
Equal.OpC -- Instruction EQUAL

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR:   -- operand adresses
  INT32 infL, infR: -- operand values
  INT   datL, datR: -- data parts of operands
  BOOL  errL, errR: -- error flags of operands
  INT32 result:     -- result of operation

  SEQ

    -- select operands
    opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
    infL, infR := frame.store[opL], frame.store[opR]
    datL, datR := ASHIFTRIGHT(INT infL, 1), ASHIFTRIGHT(INT infR, 1)
    errL, errR := BOOL(infL /\ 1(INT32)), BOOL(infR /\ 1(INT32))

    -- compute results
    IF
      errL
        -- left operand invalid, keep error code
        result := infL
      errR
        -- right operand invalid, keep error code
        result := infR
      TRUE
        -- test equality and generate error bit 0
        result := (INT32 (datL = datR)) << 1

```

```
      -- store results
      frame.store[opL] := result

      -- go to successor instruction
      instruction := instruction + (INT succ)


  --------------------------------------------------------------------------
  -- Instruction EQUALC
  --------------------------------------------------------------------------
  EqualC.OpC -- Instruction EQUALC

    -- components of operand word
    BYTE con  IS op.word[2]: -- constant operand
    BYTE reg  IS op.word[1]: -- register operand
    BYTE succ IS op.word[0]: -- successor instruction

    -- local variables
    INT   op:     -- operand adress
    INT32 inf:    -- operand value
    INT   dat:    -- data part of operand
    BOOL  err:    -- error flag of operand
    INT   scon:   -- signed constant operand
    INT32 result: -- result of operation

    SEQ

      -- select operands
      op  := (INT frame.base)+(INT reg)
      inf := frame.store[op]
      dat := ASHIFTRIGHT(INT inf, 1)
      err := BOOL(inf /\ 1(INT32))

      -- compute results
      IF
        err
          -- register operand invalid, keep error code
          result := inf
        TRUE
          SEQ
            -- sign constant
            IF
              con > (BYTE Max.Con)
                scon := (INT con) - Range.Con
              TRUE
                scon := INT con
            -- test equality and generate error bit 0
            result := (INT32 (dat = scon)) << 1

      -- store results
      frame.store[op] := result

      -- go to successor instruction
      instruction := instruction + (INT succ)
```

```

------------------------------------------------------------------------
-- Instruction LESS
------------------------------------------------------------------------
Less.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR:   -- operand adresses
  INT32 infL, infR: -- operand values
  INT   datL, datR: -- data parts of operands
  BOOL  errL, errR: -- error flags of operands
  INT32 result:     -- result of operation

  SEQ -- instruction LESS

    -- select operands
    opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
    infL, infR := frame.store[opL], frame.store[opR]
    datL, datR := ASHIFTRIGHT(INT infL, 1), ASHIFTRIGHT(INT infR, 1)
    errL, errR := BOOL(infL /\ 1(INT32)), BOOL(infR /\ 1(INT32))

    -- compute results
    IF
      errL
        -- left operand invalid, keep error code
        result := infL
      errR
        -- right operand invalid, keep error code
        result := infR
      TRUE
        -- test order and generate error bit 0
        result := (INT32 (datL < datR)) << 1

    -- store results
    frame.store[opL] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


------------------------------------------------------------------------
-- Instruction LESSC
------------------------------------------------------------------------
LessC.OpC

   -- components of operand word
  BYTE con  IS op.word[2]: -- constant operand
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

```

```
    -- local variables
    INT   op:     -- operand adress
    INT32 inf:    -- operand value
    INT   dat:    -- data part of operand
    BOOL  err:    -- error flag of operand
    INT   scon:   -- signed constant operand
    INT32 result: -- result of operation

    SEQ -- instruction LESSC

      -- select operands
      op  := (INT frame.base)+(INT reg)
      inf := frame.store[op]
      dat := ASHIFTRIGHT(INT inf, 1)
      err := BOOL(inf /\ 1(INT32))

      -- compute results
      IF
        err
          -- register operand invalid, keep error code
          result := inf
        TRUE
          SEQ
            -- sign constant
            IF
              con > (BYTE Max.Con)
                scon := (INT con) - Range.Con
              TRUE
                scon := INT con
            -- test order and generate error bit 0
            result := (INT32 (dat < scon)) << 1

      -- store results
      frame.store[op] := result

      -- go to successor instruction
      instruction := instruction + (INT succ)


  ---------------------------------------------------------------------------
  -- Instruction LESSRC
  ---------------------------------------------------------------------------
  LessRC.OpC

     -- components of operand word
    BYTE con  IS op.word[2]: -- constant operand
    BYTE reg  IS op.word[1]: -- register operand
    BYTE succ IS op.word[0]: -- successor instruction

    -- local variables
    INT   op:     -- operand adress
    INT32 inf:    -- operand value
    INT   dat:    -- data part of operand
    BOOL  err:    -- error flag of operand
    INT   scon:   -- signed constant operand
```

```
  INT32 result: -- result of operation

  SEQ -- instruction LESSRC

    -- select operands
    op  := (INT frame.base)+(INT reg)
    inf := frame.store[op]
    dat := ASHIFTRIGHT(INT inf, 1)
    err := BOOL(inf /\ 1(INT32))

    -- compute results
    IF
      err
        -- register operand invalid, keep error code
        result := inf
      TRUE
        SEQ
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
          -- test order and generate error bit 0
          result := (INT32 (scon < dat)) << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


----------------------------------------------------------------------------
-- Instruction LESSEQ
----------------------------------------------------------------------------
LessEq.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR:   -- operand adresses
  INT32 infL, infR: -- operand values
  INT   datL, datR: -- data parts of operands
  BOOL  errL, errR: -- error flags of operands
  INT32 result:     -- result of operation

  SEQ -- instruction LESSEQ

    -- select operands
    opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
    infL, infR := frame.store[opL], frame.store[opR]
```

```
      datL, datR := ASHIFTRIGHT(INT infL, 1), ASHIFTRIGHT(INT infR, 1)
      errL, errR := BOOL(infL /\ 1(INT32)), BOOL(infR /\ 1(INT32))

      -- compute results
      IF
        errL
          -- left operand invalid, keep error code
          result := infL
        errR
          -- right operand invalid, keep error code
          result := infR
        TRUE
          -- test order and generate error bit 0
          result := (INT32 (datL <= datR)) << 1

      -- store results
      frame.store[opL] := result

      -- go to successor instruction
      instruction := instruction + (INT succ)


  ---------------------------------------------------------------------------
  -- Instruction LESSEQC
  ---------------------------------------------------------------------------
  LessEqC.OpC

    -- components of operand word
    BYTE con  IS op.word[2]: -- constant operand
    BYTE reg  IS op.word[1]: -- register operand
    BYTE succ IS op.word[0]: -- successor instruction

    -- local variables
    INT   op:     -- operand adress
    INT32 inf:    -- operand value
    INT   dat:    -- data part of operand
    BOOL  err:    -- error flag of operand
    INT   scon:   -- signed constant operand
    INT32 result: -- result of operation

    SEQ -- instruction LESSEQC

      -- select operands
      op  := (INT frame.base)+(INT reg)
      inf := frame.store[op]
      dat := ASHIFTRIGHT(INT inf, 1)
      err := BOOL(inf /\ 1(INT32))

      -- compute results
      IF
        err
          -- register operand invalid, keep error code
          result := inf
        TRUE
          SEQ
```

```
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
          -- test order and generate error bit 0
          result := (INT32 (dat <= scon)) << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


--------------------------------------------------------------------------
-- Instruction LESSEQRC
--------------------------------------------------------------------------
LessEqRC.OpC

  -- components of operand word
  BYTE con  IS op.word[2]: -- constant operand
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   op:     -- operand adress
  INT32 inf:    -- operand value
  INT   dat:    -- data part of operand
  BOOL  err:    -- error flag of operand
  INT   scon:   -- signed constant operand
  INT32 result: -- result of operation

  SEQ -- instruction LESSEQRC

    -- select operands
    op  := (INT frame.base)+(INT reg)
    inf := frame.store[op]
    dat := ASHIFTRIGHT(INT inf, 1)
    err := BOOL(inf /\ 1(INT32))

    -- compute results
    IF
      err
        -- register operand invalid, keep error code
        result := inf
      TRUE
        SEQ
          -- sign constant
          IF
            con > (BYTE Max.Con)
              scon := (INT con) - Range.Con
            TRUE
              scon := INT con
```

ADEXEC.INC Page 23

```
          -- test order and generate error bit 0
          result := (INT32 (scon <= dat)) << 1

    -- store results
    frame.store[op] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


------------------------------------------------------------------------
-- Instruction NEG
------------------------------------------------------------------------
Neg.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR: -- operand adresses
  INT32 infR:     -- value of right operand
  INT   datR:     -- data part of right operand
  BOOL  errR:     -- error flag of right operands
  INT32 result:   -- result of operation

  SEQ -- instruction NEG

    -- select operands
    opL, opR := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
    infR     := frame.store[opR]
    datR     := ASHIFTRIGHT(INT infR, 1)
    errR     := BOOL(infR /\ 1(INT32))

    -- compute results
    IF
      errR
        -- right operand invalid, keep error code
        result := infR
      TRUE
        SEQ
          -- invert sign
          result := INT32 (- datR)
          IF
            (result > Max.Int)
              -- only positive integer overflow possible
              result := Error.Overflow
            TRUE
              result := result << 1

    -- store results
    frame.store[opL] := result

    -- go to successor instruction
```

```
    instruction := instruction + (INT succ)


--------------------------------------------------------------------------
-- Instruction NOT
--------------------------------------------------------------------------
Not.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR: -- operand adresses
  INT32 infR:     -- value of right operand
  INT   datR:     -- data part of right operand
  BOOL  errR:     -- error flag of right operands
  INT32 result:   -- result of operation

  SEQ -- instruction NOT

    -- select operands
    opL, opR := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
    infR     := frame.store[opR]
    datR     := ASHIFTRIGHT(INT infR, 1)
    errR     := BOOL(infR /\ 1(INT32))

    -- compute results
    IF
      errR
        -- right operand invalid, keep error code
        result := infR
      TRUE
        -- compute negation
        result := (INT32 (~ datR)) << 1

    -- store results
    frame.store[opL] := result

    -- go to successor instruction
    instruction := instruction + (INT succ)


--------------------------------------------------------------------------
-- Instruction FAILED
--------------------------------------------------------------------------
Failed.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
```

```
    INT   opL, opR: -- operand adresses
    INT32 infR:     -- value of right operand
    BOOL  errR:     -- error flag of right operands

    SEQ -- instruction FAILED

      -- select operands
      opL, opR := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
      infR     := frame.store[opR]
      errR     := BOOL(infR /\ 1(INT32))

      -- compute and store result
      frame.store[opL] := (INT32 errR) << 1

      -- go to successor instruction
      instruction := instruction + (INT succ)


  ---------------------------------------------------------------------------
  -- Instruction COPYI
  ---------------------------------------------------------------------------
  CopyI.OpC

    -- components of operand word
    BYTE regL IS op.word[2]: -- left register operand
    BYTE regR IS op.word[1]: -- right register operand
    BYTE succ IS op.word[0]: -- successor instruction

    SEQ -- instruction COPYI

      -- copy integer value
      frame.store[(INT frame.base)+(INT regL)] :=
        frame.store[(INT frame.base)+(INT regR)]

      -- go to successor instruction
      instruction := instruction + (INT succ)


  ---------------------------------------------------------------------------
  -- Instruction COPYP
  ---------------------------------------------------------------------------
  CopyP.OpC

    -- components of operand word
    BYTE regL IS op.word[2]: -- left register operand
    BYTE regR IS op.word[1]: -- right register operand
    BYTE succ IS op.word[0]: -- successor instruction

    -- local variables
    INT   op:   -- operand address
    INT32 ptr:  -- pointer value
    BOOL  err:  -- error flag
    INT   wght: -- reference weigth

    [Token.Size]INT32 token: -- network token
```

```
SEQ -- instruction COPYP

  -- fetch pointer value
  op  := (INT frame.base) + (INT regR)
  ptr := frame.store[op]

  -- decompose pointer into its components
  err  := BOOL (ptr /\ 1(INT32))       -- error flag
  wght := INT ((ptr >> 1) /\ 3(INT32)) -- reference weight

  IF
    (ptr = 0(INT32)) OR err
      SEQ

        -- nil or error flag set, pointer may be copied
        frame.store[(INT frame.base)+(INT regL)] := ptr

        -- go to successor instruction
        instruction := instruction + (INT succ)

    wght > 0
      SEQ

        -- decrease weight
        wght := (wght - 1) << 1
        ptr := (ptr /\ #FFFFFFF9(INT32)) \/ (INT32 wght)

        -- copy pointer
        frame.store[op] := ptr
        frame.store[(INT frame.base)+(INT regL)] := ptr

        -- go to successor instruction
        instruction := instruction + (INT succ)

    TRUE
      SEQ

        -- construct token with reference request
        [4]BYTE ptrs RETYPES ptr:
        [4]BYTE toks RETYPES token[2]:
        SEQ
          toks[2] := ptrs[3]                         -- module address
          toks[1] := IncRef.MToken                   -- increase reference
        [2]INT16 toks RETYPES token[1]:
        SEQ
          toks[1] := context                         -- actual context
          toks[0] :=                                 -- actual offset
            INT16 (instruction - (INT code.base))
        [4]BYTE ptrs RETYPES ptr:
        [4]BYTE toks RETYPES token[0]:
        SEQ
          toks[3] := act.mod                         -- actual module
          [toks FROM 0 FOR 3] := [ptrs FROM 0 FOR 3] -- node reference
```

```
          -- send token to memory unit
          out ! net.token; token

          -- break instruction sequence
          going := FALSE


----------------------------------------------------------------------------
-- Instruction DELP
----------------------------------------------------------------------------
DelP.OpC

  -- components of operand word
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- instruction offset

  -- local variables
  INT32 ptr: -- pointer value
  BOOL  err: -- error flag

  [Token.Size]INT32 token: -- network token

  SEQ -- instruction DELP

    -- fetch pointer value and extract error flag
    ptr := frame.store[(INT frame.base) + (INT reg)]
    err := BOOL (ptr /\ 1(INT32))

    IF
      (ptr = 0(INT32)) OR err

        -- nil or error flag set, go to successor instruction
        instruction := instruction + (INT succ)

      TRUE
        SEQ

          -- send decrease reference request
          [4]BYTE ptrs RETYPES ptr:
          SEQ
            -- construct token
            [4]BYTE toks RETYPES token[2]:
            SEQ
              toks[2] := ptrs[3]            -- module address
              toks[1] := DecRef.MToken      -- decrease reference
            [4]BYTE toks RETYPES token[0]:
            [toks FROM 0 FOR 3] :=          -- node reference
              [ptrs FROM 0 FOR 3]

          -- send token to memory unit
          out ! net.token; token

          -- go to successor instruction
          instruction := instruction + (INT succ)

```

```
--------------------------------------------------------------------------
-- Instruction SPLIT
--------------------------------------------------------------------------
Split.OpC

  -- components of operand word
  BYTE succ1 IS op.word[1]: -- first successor instruction
  BYTE succ2 IS op.word[0]: -- second successor instruction

  -- local variables
  INT32 trigger: -- instruction trigger

  SEQ -- instruction SPLIT

    -- construct trigger with first successor instruction
    [2]INT16 i RETYPES trigger:
    INT16 cont IS i[1]: -- context field
    INT16 off  IS i[0]: -- offset field
    SEQ
      cont := context
      off  := (INT16 (instruction - (INT code.base))) + (INT16 succ1)

    -- put it into trigger store
    Trig.Put(trigger)

    -- go to second successor instruction
    instruction := instruction + (INT succ2)


--------------------------------------------------------------------------
-- Instruction SWITCH
--------------------------------------------------------------------------
Switch.OpC

  -- components of operand word
  BYTE reg   IS op.word[2]: -- register operand
  BYTE succ1 IS op.word[1]: -- first successor instruction
  BYTE succ2 IS op.word[0]: -- second successor instruction

  -- local variables
  INT32 inf:  -- test operand
  INT32 dat: -- value of operand
  BOOL  err: -- error flag of operand

  SEQ -- instruction SWITCH

    -- select operands
    inf := frame.store[(INT frame.base)+(INT reg)]
    dat := inf >> 1
    err := BOOL(inf /\ 1(INT32))

    IF
      err
        -- test value invalid, go to next instruction
        instruction := instruction + 1
```

```
      dat = (INT32 TRUE)
        -- test value true, go to first successor
        instruction := instruction + (INT succ1)
      dat = (INT32 FALSE)
        -- test value false, go to second successor
        instruction := instruction + (INT succ2)
      TRUE
        -- fatal error occurred
        Assert(FALSE, true.false)


--------------------------------------------------------------------------
-- Instruction JUMP
--------------------------------------------------------------------------
Jump.OpC

  -- operand word denotes SIGNED offset to successor instruction

  -- local variables
  INT32 off: -- instruction offset

  SEQ -- instruction JUMP

    -- increase instruction by SIGNED integer equivalent of op.word
    [4]BYTE o RETYPES off:
    [o FROM 1 FOR 3] := op.word
    instruction := instruction + ASHIFTRIGHT(INT off,8)


--------------------------------------------------------------------------
-- Instruction WAIT
--------------------------------------------------------------------------
Wait.OpC

  -- components of operand word
  [2]BYTE ev IS [op.word FROM 1 FOR 2]: -- event number
  BYTE  succ IS op.word[0]:             -- successor instruction

  -- local variables
  INT  slot.num: -- number of event slot
  INT  bit.num:  -- number of event bit
  INT  addr:     -- address of slot
  BOOL synch:    -- synchronization neccessary

  SEQ -- instruction WAIT

    -- convert event number to integer
    [4]BYTE e RETYPES null:
    [e FROM 0 FOR 2] := ev

    -- compute slot number and bit number of event
    slot.num, bit.num := (INT null) / 32, (INT null) REM 32

    -- compute effective slot address
    addr := ((INT frame.base) - 3) - slot.num
```

```

      -- get bit and invert it
      INT32     slot IS frame.store[addr]:
      VAL INT32 bit  IS bits[bit.num]:
      SEQ
        synch := (slot /\ bit) = 0(INT32)
        slot  := slot >< bit

      IF
        synch
          -- event bit not set, break execution
          going := FALSE
        TRUE
          -- event bit set, continue execution
          instruction := instruction + (INT succ)


----------------------------------------------------------------------------
-- Instruction CONST
----------------------------------------------------------------------------
Const.OpC

  -- components of operand word
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  SEQ -- instruction CONST

    -- copy long constant from program store into register
    frame.store[(INT frame.base)+(INT reg)] := prog.store[instruction+1]

    -- go to successor instruction
    instruction := instruction + (INT succ)


----------------------------------------------------------------------------
-- Instruction FASTCALL
----------------------------------------------------------------------------
FastCall.OpC

  -- components of operand word
  [2]BYTE fun  IS [op.word FROM 1 FOR 2]: -- function descriptor
  BYTE    succ IS op.word[0]:             -- successor instruction

  -- local variables
  [Token.Size]INT32 token: -- network token

  SEQ -- instruction FASTCALL

    -- construct token
    [4]BYTE toks RETYPES token[2]:
    SEQ
      toks[2] := Min.Loaded()       -- address of minimal loaded module
      toks[1] := FCall.PToken       -- fast call
    [2]INT16 toks RETYPES token[1]:
```

```
    SEQ
      toks[1] := context            -- actual context
      toks[0] :=                    -- successor offset
        (INT16 (instruction - (INT code.base))) + (INT16 succ)
    [4]BYTE toks RETYPES token[0]:
    SEQ
      toks[3] := act.mod            -- actual module
      [toks FROM 0 FOR 2] := fun    -- function descriptor

    -- send token to processor unit
    out ! net.token; token

    -- break instruction sequence
    going := FALSE


--------------------------------------------------------------------------
-- Instruction TAILCALL
--------------------------------------------------------------------------
TailCall.OpC

  -- components of operand word
  [2]BYTE fun  IS [op.word FROM 1 FOR 2]: -- function descriptor
  BYTE    succ IS op.word[0]:             -- successor instruction

  -- local variables
  [Token.Size]INT32 token: -- network token

  SEQ -- instruction TAILCALL

    -- construct token
    [4]BYTE toks RETYPES token[2]:
    SEQ
      toks[2] := Min.Loaded()       -- address of minimal loaded module
      toks[1] := TCall.PToken       -- fast call
    [2]INT16 toks RETYPES token[1]:
    SEQ
      toks[1] := context            -- actual context
      toks[0] :=                    -- successor offset
        (INT16 (instruction - (INT code.base))) + (INT16 succ)
    [4]BYTE toks RETYPES token[0]:
    SEQ
      toks[3] := act.mod            -- actual module
      [toks FROM 0 FOR 2] := fun    -- function descriptor

    -- send token to processor unit
    out ! net.token; token

    -- break instruction sequence
    going := FALSE


--------------------------------------------------------------------------
-- Instruction CONTEXT
--------------------------------------------------------------------------
```

```
Context.OpC

  -- components of operand word
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT32 dat:               -- register operand
  [Token.Size]INT32 token: -- network token

  SEQ -- instruction CONTEXT

    -- select operand
    dat := frame.store[(INT frame.base)+(INT reg)] >> 1

    -- components of activation address
    [2]INT16 dati RETYPES dat:
    INT16    cont IS dati[0]:
    [2]BYTE  datb RETYPES dati[1]:
    BYTE     mod  IS datb[0]:

    -- components of result address
    [2]INT16 addr0 RETYPES frame.store[(INT frame.base)-1]:
    INT16 res.cont IS addr0[1]:
    INT16 res.off  IS addr0[0]:
    [4]BYTE addr1 RETYPES frame.store[(INT frame.base)-2]:
    BYTE  res.mod  IS addr1[0]:

    -- construct token
    SEQ
      [4]BYTE toks RETYPES token[2]:
      SEQ
        toks[2] := mod                -- module address
        toks[1] := Cont.PToken        -- context message
      [2]INT16 toks RETYPES token[1]:
      SEQ
        toks[1] := res.cont           -- result context
        toks[0] := res.off            -- result offset
      [4]BYTE toks RETYPES token[0]:
      toks[3] := res.mod              -- result module
      [2]INT16 toks RETYPES token[0]:
      toks[0] := cont                 -- context

    -- send token to processor unit
    out ! net.token; token

    -- jump to successor instruction
    instruction := instruction + (INT succ)


---------------------------------------------------------------------------
-- Instruction APPLY
---------------------------------------------------------------------------
Apply.OpC

```

```
    -- components of operand word
    BYTE reg  IS op.word[1]: -- register operand
    BYTE succ IS op.word[0]: -- successor instruction

    -- local variables
    INT32 inf: -- operand
    INT16 fun: -- function descriptor of operand
    BOOL  err: -- error flag of operand

    [Token.Size]INT32 token: -- network token

    SEQ -- instruction APPLY

      -- select operand
      inf := frame.store[(INT frame.base)+(INT reg)]
      fun := INT16 ((inf >> 1) /\ #FFFF(INT32))
      err := BOOL(inf /\ 1(INT32))

      IF
        err
          SEQ
            -- error flag of function descriptor set

            -- replace activation address by error value
            [4]BYTE next RETYPES prog.store[instruction+(INT succ)]:
            BYTE    nreg IS next[1]:
            frame.store[(INT frame.base)+(INT nreg)] := inf

            -- replace result value by error value
            [4]BYTE next RETYPES prog.store[(instruction+(INT succ))+1]:
            BYTE    nreg IS next[1]:
            frame.store[(INT frame.base)+(INT nreg)] := inf

        TRUE
          SEQ
            -- function descriptor valid

            -- construct token
            [4]BYTE toks RETYPES token[2]:
            SEQ
              toks[2] := Min.Loaded()       -- minimal loaded module
              toks[1] := FCall.PToken       -- fast call
            [2]INT16 toks RETYPES token[1]:
            SEQ
              toks[1] := context            -- actual context
              toks[0] :=                    -- successor offset
                (INT16 (instruction - (INT code.base))) + (INT16 succ)
            [2]INT16 toks RETYPES token[0]:
            [2]BYTE  tokb RETYPES toks[1]:
            SEQ
              tokb[1] := act.mod -- actual module
              toks[0] := fun     -- function descriptor

            -- send token to processor unit
            out ! net.token; token
```

```

          -- break instruction sequence
          going := FALSE


--------------------------------------------------------------------------
-- Instruction SEND
--------------------------------------------------------------------------
Send.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE par  IS op.word[0]: -- parameter number

  -- local variables
  INT32 inf: -- operand
  INT32 dat: -- data field of operand
  BOOL  err: -- error flag of operand

  [Token.Size]INT32 token: -- network token

  SEQ -- instruction SEND

    -- select operand
    inf := frame.store[(INT frame.base)+(INT regR)]
    dat := inf >> 1
    err := BOOL(inf /\ 1(INT32))

    IF
      err

        -- error flag of activation adress set, do nothing
        SKIP

      TRUE
        SEQ

          -- components of activation address
          [2]INT16 dati RETYPES dat:
          INT16    cont IS dati[0]:
          [2]BYTE  datb RETYPES dati[1]:
          BYTE     mod  IS datb[0]:

          -- construct token
          SEQ
            [4]BYTE toks RETYPES token[2]:
            SEQ
              toks[2] := mod               -- module address of callee
              toks[1] := Mess.PToken       -- message token
            [2]INT16 toks RETYPES token[1]:
            SEQ
              toks[1] := cont              -- context of callee
              toks[0] := (INT16 par)       -- parameter number
            token[0] :=                    -- argument value
```

```
              frame.store[(INT frame.base) + (INT regL)]

          -- send token to processor unit
          out ! net.token; token

    -- go to immediate next instruction
    instruction := instruction + 1


--------------------------------------------------------------------------
-- Instruction SENDC
--------------------------------------------------------------------------
SendC.OpC

  -- components of operand word
  BYTE con IS op.word[2]: -- constant argument
  BYTE reg IS op.word[1]: -- register operand
  BYTE par IS op.word[0]: -- parameter number

  -- local variables
  INT32 inf: -- operand
  INT32 dat: -- data field of operand
  BOOL  err: -- error flag of operand
  INT   arg: -- argument value

  [Token.Size]INT32 token: -- network token

  SEQ -- instruction SENDC

    -- select operand
    inf := frame.store[(INT frame.base)+(INT reg)]
    dat := inf >> 1
    err := BOOL(inf /\ 1(INT32))

    IF
      err

        -- error flag of activation adress set, do nothing
        SKIP

      TRUE
        SEQ

          -- argument value is signed constant with error flag
          IF
            con > (BYTE Max.Con)
              arg := ((INT con) - Range.Con) << 1
            TRUE
              arg := (INT con) << 1

          -- components of activation address
          [2]INT16 dati RETYPES dat:
          INT16    cont IS dati[0]:
          [2]BYTE  datb RETYPES dati[1]:
          BYTE     mod  IS datb[0]:
```

```
          -- construct token
          SEQ
            [4]BYTE toks RETYPES token[2]:
            SEQ
              toks[2] := mod                -- module address of callee
              toks[1] := Mess.PToken        -- message token
            [2]INT16 toks RETYPES token[1]:
            SEQ
              toks[1] := cont               -- context of callee
              toks[0] := (INT16 par)        -- parameter number
            token[0] := INT32 arg           -- argument value

          -- send token to processor unit
          out ! net.token; token

    -- go to immediate next instruction
    instruction := instruction + 1


----------------------------------------------------------------------------
-- Instruction SUBMIT
----------------------------------------------------------------------------
Submit.OpC

  -- components of operand word
  BYTE regL IS op.word[2]: -- left register operand
  BYTE regR IS op.word[1]: -- right register operand
  BYTE succ IS op.word[0]: -- successor instruction

  -- local variables
  INT   opL, opR:   -- operand adresses
  INT32 infL, infR: -- operand values
  BOOL  errL, errR: -- error flags of operands
  INT32 datR:       -- data value of operand

  [Token.Size]INT32 token: -- network token

  SEQ -- instruction SUBMIT

    -- select operands
    opL,  opR  := (INT frame.base)+(INT regL), (INT frame.base)+(INT regR)
    infL, infR := frame.store[opL], frame.store[opR]
    errL, errR := BOOL(infL /\1(INT32)), BOOL(infR /\ 1(INT32))

    IF
      errL OR errR

        -- error flag set, do nothing
        SKIP

      TRUE
        SEQ

          -- data value of right operand
```

```
          datR := infR >> 1

          -- components of pointer to curried function
          [4]BYTE  infs RETYPES infL:
          BYTE     rmod IS infs[3]:             -- module address of tuple
          [3]BYTE  ref  IS [infs FROM 0 FOR 3]: -- tuple reference

          -- components of activation address
          [2]INT16  dati RETYPES datR:
          INT16     cont IS dati[0]:            -- context of activation
          [2]BYTE   datb RETYPES dati[1]:
          BYTE      mod  IS datb[0]:            -- module address

          -- construct token
          SEQ
            [4]BYTE toks RETYPES token[2]:
            SEQ
              toks[2] := rmod               -- module address of tuple
              toks[1] := Submit.MToken      -- submission token
            [2]INT16 toks RETYPES token[1]:
            toks[1] := cont                 -- context of activation
            [4]BYTE toks RETYPES token[0]:
            SEQ
              toks[3] := mod                 -- module of activation
              [toks FROM 0 FOR 3] := ref     -- tuple reference

          -- send token to processor unit
          out ! net.token; token

    -- go to successor instruction
    instruction := instruction + (INT succ)


---------------------------------------------------------------------------
-- Instruction RETURN
---------------------------------------------------------------------------
Return.OpC

  -- components of operand word
  BYTE reg  IS op.word[1]: -- register operand
  BYTE succ IS op.word[0]: -- successor instruction

  [Token.Size]INT32 token: -- network token

  SEQ -- instruction RETURN

    -- select return address
    [2]INT16 addr0 RETYPES frame.store[(INT frame.base)-1]:
    INT16    cont  IS addr0[1]:
    INT16    off   IS addr0[0]:
    [4]BYTE  addr1 RETYPES frame.store[(INT frame.base)-2]:
    BYTE     mod   IS addr1[0]:

    -- construct token
    SEQ
```

```
        [4]BYTE toks RETYPES token[2]:
        SEQ
          toks[2] := mod                -- module address
          toks[1] := Mess.PToken        -- message token
        [2]INT16 toks RETYPES token[1]:
        SEQ
          toks[1] := cont               -- return context
          toks[0] := off                -- return offset
        token[0] :=                     -- result value
          frame.store[(INT frame.base)+(INT reg)]

      -- send token to processor unit
      out ! net.token; token

      -- go to successor instruction
      instruction := instruction + (INT succ)


----------------------------------------------------------------------------
-- Instruction RETURNC
----------------------------------------------------------------------------
ReturnC.OpC

  -- components of operand word
  BYTE con  IS op.word[1]: -- constant operand
  BYTE succ IS op.word[0]: -- successor instruction

  INT res:                 -- result value

  [Token.Size]INT32 token: -- network token

  SEQ -- instruction RETURNC

    -- argument value is signed constant with error flag
    IF
      con > (BYTE Max.Con)
        res := ((INT con) - Range.Con) << 1
      TRUE
        res := (INT con) << 1

    -- select return address
    [2]INT16 addr0 RETYPES frame.store[(INT frame.base)-1]:
    INT16    cont  IS addr0[1]:
    INT16    off   IS addr0[0]:
    [4]BYTE  addr1 RETYPES frame.store[(INT frame.base)-2]:
    BYTE     mod   IS addr1[0]:

    -- construct token
    SEQ
      [4]BYTE toks RETYPES token[2]:
      SEQ
        toks[2] := mod                -- module address
        toks[1] := Mess.PToken        -- message token
      [2]INT16 toks RETYPES token[1]:
      SEQ
```

```
        toks[1] := cont               -- return context
        toks[0] := off                -- return offset
      token[0] := INT32 res           -- result value

    -- send token to processor unit
    out ! net.token; token

    -- go to successor instruction
    instruction := instruction + (INT succ)


--------------------------------------------------------------------------
-- Instruction TERMINATE
--------------------------------------------------------------------------
Terminate.OpC

  [4]BYTE inst RETYPES prog.store[(INT code.base)-1]:
  BYTE    evts IS inst[1]:
  BYTE    regs IS inst[0]:

  SEQ -- instruction TERMINATE

    -- deallocate frame
    Dealloc.Frame((INT frame.base), (INT evts), (INT regs))

    -- deallocate context
    Dealloc.Context(context)

    -- break execution sequence
    going := FALSE


--------------------------------------------------------------------------
-- Instruction ALLOC
--------------------------------------------------------------------------
Alloc.OpC

  -- components of operand word
  BYTE const IS op.word[1]: -- constant operand
  BYTE succ  IS op.word[0]: -- successor instruction

  [Token.Size]INT32 token: -- network token

  SEQ -- instruction ALLOC

    -- construct token
    SEQ
      [4]BYTE toks RETYPES token[2]:
      SEQ
        toks[2] := act.mod            -- module address (actual module)
        toks[1] := Alloc.MToken       -- allocation token
        toks[0] := const              -- number of tuple fields
      [2]INT16 toks RETYPES token[1]:
      SEQ
        toks[1] := context            -- return context
```

```
        toks[0] :=                    -- return offset
          (INT16 (instruction - (INT code.base))) + (INT16 succ)
      [4]BYTE toks RETYPES token[0]:
      toks[3] := act.mod              -- return module

    -- send token to memory unit on actual module
    out ! net.token; token

    -- break execution sequence
    going := FALSE


--------------------------------------------------------------------------
-- Instruction READ
--------------------------------------------------------------------------
Read.OpC

  -- components of operand word
  BYTE const IS op.word[2]: -- constant operand
  BYTE reg   IS op.word[1]: -- register operand
  BYTE succ  IS op.word[0]: -- successor instruction

  INT32 dat: -- tuple reference

  [Token.Size]INT32 token: -- network token

  SEQ -- instruction READ

    dat := frame.store[(INT frame.base)+(INT reg)]

    IF

      -- reference is nil
      dat = 0(INT32)
        SEQ

          -- replace result value by error value
          [4]BYTE next RETYPES prog.store[instruction+(INT succ)]:
          BYTE    nreg IS next[1]:
          frame.store[(INT frame.base)+(INT nreg)] := Error.Read

          -- go to successor instruction
          instruction := instruction + (INT succ)

      -- reference is error
      (dat /\ 1(INT32)) > 0(INT32)
        SEQ

          -- replace result value by propagated error value
          [4]BYTE next RETYPES prog.store[instruction+(INT succ)]:
          BYTE    nreg IS next[1]:
          frame.store[(INT frame.base)+(INT nreg)] := dat

          -- go to successor instruction
          instruction := instruction + (INT succ)
```

```

      -- reference is valid
      TRUE

        -- select components of tuple reference
        [4]BYTE dats RETYPES dat:
        BYTE    mod  IS dats[3]:
        [3]BYTE ref  IS [dats FROM 0 FOR 3]:

        SEQ

          -- construct token
          [4]BYTE toks RETYPES token[2]:
          SEQ
            toks[2] := mod                -- module address of tuple
            toks[1] := Read.MToken        -- read token
            toks[0] := const              -- number of tuple field to read
          [2]INT16 toks RETYPES token[1]:
          SEQ
            toks[1] := context            -- return context
            toks[0] :=                    -- return offset
              (INT16 (instruction - (INT code.base))) + (INT16 succ)
          [4]BYTE toks RETYPES token[0]:
          SEQ
            toks[3] := act.mod            -- return module
            [toks FROM 0 FOR 3] := ref    -- tuple reference

          -- send token to memory unit
          out ! net.token; token

          -- break execution sequence
          going := FALSE


-----------------------------------------------------------------------------
-- Instruction WRITEI
-----------------------------------------------------------------------------
WriteI.OpC

  -- components of operand word
  BYTE regL  IS op.word[2]: -- left register operand (integer value)
  BYTE regR  IS op.word[1]: -- right register operand (tuple reference)
  BYTE const IS op.word[0]: -- constant value (number of tuple field)

  INT32 ptr: -- tuple reference
  BOOL  err: -- error flag

  [Token.Size]INT32 token: -- network token

  SEQ -- instruction WRITEI

    -- fetch tuple reference
    ptr := frame.store[(INT frame.base)+(INT regR)]
    err := (ptr /\ 1(INT32)) <> 0(INT32)

```

```
    -- select components of tuple reference
    [4]BYTE  ptrs RETYPES ptr:
    BYTE     mod  IS ptrs[3]:             -- module address
    [3]BYTE  ref  IS [ptrs FROM 0 FOR 3]: -- tuple address

    -- construct token
    SEQ

      -- tuple reference must be valid
      Assert(NOT err, ref.err)

      [4]BYTE toks RETYPES token[2]:
      SEQ
        toks[2] := mod                -- module address of tuple
        toks[1] := WrInt.MToken       -- write integer request
        toks[0] := const              -- number of tuple field
      token[1] :=                     -- value to be written
        frame.store[(INT frame.base) + (INT regL)]
      [4]BYTE toks RETYPES token[0]:
      [toks FROM 0 FOR 3] := ref      -- tuple reference

    -- send token to memory unit
    out ! net.token; token

    -- continue execution with immediately next instruction
    instruction := instruction + 1


---------------------------------------------------------------------------
-- Instruction WRITEP
---------------------------------------------------------------------------
WriteP.OpC

  -- components of operand word
  BYTE regL  IS op.word[2]: -- left register operand (pointer value)
  BYTE regR  IS op.word[1]: -- right register operand (tuple reference)
  BYTE const IS op.word[0]: -- constant value (number of tuple field)

  INT32 ptr: -- tuple reference
  BOOL  err: -- error flag

  [Token.Size]INT32 token: -- network token

  SEQ -- instruction WRITEP

    -- fetch tuple reference
    ptr := frame.store[(INT frame.base)+(INT regR)]
    err := (ptr /\ 1(INT32)) <> 0(INT32)

    -- select components of tuple reference
    [4]BYTE  ptrs RETYPES ptr:
    BYTE     mod  IS ptrs[3]:             -- module address
    [3]BYTE  ref  IS [ptrs FROM 0 FOR 3]: -- tuple address

    -- construct token
```

```
      SEQ

        -- tuple reference must be valid
        Assert(NOT err, ref.err)

        [4]BYTE toks RETYPES token[2]:
        SEQ
          toks[2] := mod                -- module address of tuple
          toks[1] := WrPtr.MToken       -- write pointer request
          toks[0] := const              -- number of tuple field
        token[1] :=                     -- value to be written
          frame.store[(INT frame.base) + (INT regL)]
        [4]BYTE toks RETYPES token[0]:
        [toks FROM 0 FOR 3] := ref      -- tuple reference

      -- send token to memory unit
      out ! net.token; token

      -- continue execution with immediately next instruction
      instruction := instruction + 1


  --------------------------------------------------------------------------
  -- Instruction WRITEC
  --------------------------------------------------------------------------
  WriteC.OpC

    -- components of operand word
    BYTE con   IS op.word[2]: -- constant argument value
    BYTE reg   IS op.word[1]: -- register operand (tuple reference)
    BYTE const IS op.word[0]: -- constant value (number of tuple field)

    -- local variables
    INT32 ptr: -- tuple reference
    BOOL  err: -- error flag
    INT   arg: -- signed integer value

    [Token.Size]INT32 token: -- network token

    SEQ -- instruction WRITEC

      -- argument value is signed constant with error flag
      IF
        con > (BYTE Max.Con)
          arg := ((INT con) - Range.Con) << 1
        TRUE
          arg := (INT con) << 1

      -- fetch tuple reference
      ptr := frame.store[(INT frame.base)+(INT reg)]
      err := (ptr /\ 1(INT32)) <> 0(INT32)

      -- select components of tuple reference
      [4]BYTE  ptrs RETYPES ptr:
      BYTE     mod  IS ptrs[3]:             -- module address
```

```
      [3]BYTE  ref  IS [ptrs FROM 0 FOR 3]: -- tuple address

      SEQ

        -- tuple reference must be valid
        Assert(NOT err, ref.err)

        -- construct token
        [4]BYTE toks RETYPES token[2]:
        SEQ
          toks[2] := mod                -- module address of tuple
          toks[1] := WrInt.MToken       -- write integer request
          toks[0] := const              -- number of tuple field
        token[1] := INT32 arg           -- value to be written
        [4]BYTE toks RETYPES token[0]:
        [toks FROM 0 FOR 3] := ref      -- tuple reference

      -- send token to memory unit
      out ! net.token; token

      -- continue execution with immediately next instruction
      instruction := instruction + 1


  ---------------------------------------------------------------------------
  -- Instruction Invalid
  ---------------------------------------------------------------------------
  ELSE
    Assert(FALSE, inst.err)


-----------------------------------------------------------------------------
-- End of ADEXEC.INC
-----------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------
--
-- ADMEMORY.INC                                         Wolfgang Schreiner
-- Memory Constants and Protocols                       (c) RISC-Linz 1990
--
-- Defines all the constants and protocols that are used within the
-- memory unit of an ADAM module.
--
--------------------------------------------------------------------------

VAL INT K IS 1024: -- unit of measure

VAL INT Heap.Len  IS  64*K:  -- length of heap in cells
VAL INT Exp.Len   IS   4*K:  -- length of expensive request store in tokens
VAL INT Cheap.Len IS   4*K:  -- length of cheap request store in tokens
VAL INT Stack.Len IS   4*K:  -- length of deallocation stack in tokens

VAL INT Range.Count IS 256: -- range of reference count field (excl.)

--------------------------------------------------------------------------
-- protocols (within memory unit)
--------------------------------------------------------------------------

-- token protocol
PROTOCOL Memory.Token IS [Token.Size]INT32:

-- demand protocol (just a tag for synchronization)
PROTOCOL Demand
  CASE
    demand.tag
:

-- buffer protocol (number of buffered elements and delivered token)
PROTOCOL Buffer.Token IS INT; [Token.Size]INT32:

--------------------------------------------------------------------------
-- action codes
--------------------------------------------------------------------------

VAL INT Exp.Act    IS 0: -- fulfill expensive request
VAL INT Cheap.Act  IS 1: -- fulfill cheap request
VAL INT Report.Act IS 2: -- report actual load

--------------------------------------------------------------------------
-- The following variables control the behaviour of the procedure
-- 'Mem.Action' determining the next action of the memory unit:
--
--    * Mem.Trash denotes the minimal memory load for sending a load report.
--    * Mem.Delay denotes 100 minus the tolerable relative difference between
--      the actual processor load and the last reported one.
--    * Heap.Trash denotes a lower bound for the load of the heap store.
--      If the actual load falls below this bound, the memory fulfills more
--      expensive requests.
--    * Heap.Bound denotes an upper bound for the load of the bound store.
--      If the actual load exceedes this bound, the memory fulfills less
```

```
--     expensive requests.
--   * Exp.Bound denotes an upper bound for the load of the expensive store.
--     If the actual load exceedes this bound, the processor will fulfill
--     more expensive requests.
-----------------------------------------------------------------------------

VAL INT Mem.Trash  IS 1:  -- lower bound memory load (>0 <= 100)
VAL INT Mem.Delay  IS 90: -- report delay (>0 <= 100)

VAL INT Heap.Trash IS Heap.Len * (25/100): -- lower bound heap store
VAL INT Heap.Bound IS Heap.Len * (75/100): -- upper bound heap store
VAL INT Exp.Bound  IS Exp.Len  * (50/100): -- bound request store

-----------------------------------------------------------------------------
-- End of ADMEMORY.INC
-----------------------------------------------------------------------------
```

```
ADPROC.INC                                                          Page   1

---------------------------------------------------------------------------
--
-- ADPROC.INC                                           Wolfgang Schreiner
-- Processor Constants and Protocols                    (c) RISC-Linz 1990
--
-- Defines all the constants and protocols that are used within the
-- processor unit of an ADAM module.
--
---------------------------------------------------------------------------

---------------------------------------------------------------------------
-- constants
---------------------------------------------------------------------------

VAL INT K IS 1024: -- storage unit in words

VAL INT Prog.Len  IS   8*K: -- length of program store in words
VAL INT Frame.Len IS  64*K: -- length of frame store (must be a power of 2)
VAL INT Cont.Len  IS   8*K: -- length of context store in words
VAL INT Trig.Len  IS   1*K: -- length of trigger store in words
VAL INT Mess.Len  IS   4*K: -- length of message store in tokens
VAL INT Requ.Len  IS   4*K: -- length of request store in tokens

--------------------------------------------------------------------------
-- constants for managing the heap-oriented frame store (buddy method)
--------------------------------------------------------------------------

-- frame blocks may be from 8 to 256 words long (in factors of 2)
VAL INT Free.Min   IS 8:   -- length of smallest activation frame
VAL INT Free.Max   IS 256: -- length of largest activation frame
VAL INT Free.Min.B IS 3:   -- binary logarithm of Free.Min
VAL INT Free.Step  IS 6:   -- number of different sizes of activation frames

VAL INT Used.Len   IS (Frame.Len/256): -- length of array of used bits

---------------------------------------------------------------------------
-- 'Significance' must have a value between 0 (excl.) and 100 (incl).
-- It denotes the factor by that the load of a neighbour must be
-- smaller than the load of the actual processor in order to forward
-- activation requests to that processor.
---------------------------------------------------------------------------

VAL INT Significance IS 100:  -- load significance

---------------------------------------------------------------------------
-- The following variables control the behaviour of the procedure
-- 'Proc.Action' determining the next action of the processor unit:
--
--    * Proc.Trash denotes the minimal processor load for sending a load
--      report.
--    * Rep.Delay denotes 100 minus the tolerable relative difference
--      between the actual processor load and the last reported one.
--    * Frame.Trash denotes a lower bound for the load of the frame store.
--      If the actual load falls below this bound, the processor fulfills more
--      call requests.
```

```
--   * Frame.Bound denotes an upper bound for the load of the frame store.
--     If the actual load exceedes this bound, the processor fulfills less
--     call requests.
--   * Requ.Bound denotes an upper bound for the load of the request store.
--     If the actual load exceedes this bound, the processor will fulfill
--     more call requests.
-----------------------------------------------------------------------------

VAL INT Proc.Trash  IS 1:                     -- lower bound processor load
VAL INT Rep.Delay   IS 90:                    -- report delay (>0 <= 100)
VAL INT Frame.Trash IS Frame.Len * (25/100): -- lower bound frame store
VAL INT Frame.Bound IS Frame.Len * (75/100): -- upper bound frame store
VAL INT Requ.Bound  IS Requ.Len  * (50/100): -- upper bound request store

-----------------------------------------------------------------------------
-- protocols (within processor unit)
-----------------------------------------------------------------------------
-- token protocol
PROTOCOL Proc.Token IS [Token.Size]INT32:

-- cluster protocol (for transmitting the program to the execution unit)
PROTOCOL Cluster
  CASE
    start.cluster                    -- start of new program
    append.cluster                   -- append program fragment
    cluster;    [Cluster.Size]INT32  -- program cluster
    end.cluster                      -- end of cluster signal
:

-- demand protocol (just a tag for synchronization)
PROTOCOL Demand
  CASE
    demand.tag
:

-- buffer protocol (number of buffered elements and delivered token)
PROTOCOL Buffer.Token IS INT; [Token.Size]INT32:

-----------------------------------------------------------------------------
-- action codes
-----------------------------------------------------------------------------

VAL INT Execute.Act IS 0: -- execute instruction
VAL INT Receive.Act IS 1: -- receive message
VAL INT Request.Act IS 2: -- fulfill request
VAL INT Report.Act  IS 3: -- report load

-----------------------------------------------------------------------------
-- instruction codes
-----------------------------------------------------------------------------

-- arithmetical-logical instructions
VAL BYTE Add.OpC       IS #00(BYTE): -- integer addition
VAL BYTE AddC.OpC      IS #01(BYTE): -- integer addition with constant
VAL BYTE Sub.OpC       IS #02(BYTE): -- integer subtraction
```

ADPROC.INC Page 3

```
VAL BYTE SubC.OpC      IS #03(BYTE): -- integer subtraction with constant
VAL BYTE SubRC.OpC     IS #04(BYTE): -- integer subtraction reverse with c.
VAL BYTE Mul.OpC       IS #05(BYTE): -- integer multiplication
VAL BYTE MulC.OpC      IS #06(BYTE): -- integer multiplication with constant
VAL BYTE Div.OpC       IS #07(BYTE): -- integer division
VAL BYTE DivC.OpC      IS #08(BYTE): -- integer division with constant
VAL BYTE DivRC.OpC     IS #09(BYTE): -- integer division reverse with const.
VAL BYTE Mod.OpC       IS #0A(BYTE): -- integer remainder
VAL BYTE ModC.OpC      IS #0B(BYTE): -- integer remainder with constant
VAL BYTE ModRC.OpC     IS #0C(BYTE): -- integer remainder reverse with const.
VAL BYTE And.OpC       IS #0D(BYTE): -- boolean conjunction
VAL BYTE Or.OpC        IS #0E(BYTE): -- boolean disjunction
VAL BYTE Equal.OpC     IS #0F(BYTE): -- integer equality
VAL BYTE EqualC.OpC    IS #10(BYTE): -- integer equality with constant
VAL BYTE Less.OpC      IS #11(BYTE): -- integer strict order
VAL BYTE LessC.OpC     IS #12(BYTE): -- integer strict order with constant
VAL BYTE LessRC.OpC    IS #13(BYTE): -- integer strict order reverse with c.
VAL BYTE LessEq.OpC    IS #14(BYTE): -- integer reflexive order
VAL BYTE LessEqC.OpC   IS #15(BYTE): -- integer reflexive order with const.
VAL BYTE LessEqRC.OpC  IS #16(BYTE): -- integer reflexive order rev. with c.
VAL BYTE Neg.OpC       IS #17(BYTE): -- integer negation
VAL BYTE Not.OpC       IS #18(BYTE): -- boolean negation
VAL BYTE Failed.OpC    IS #19(BYTE): -- test for error value

-- control instructions
VAL BYTE CopyI.OpC     IS #20(BYTE): -- copy integer value
VAL BYTE CopyP.OpC     IS #21(BYTE): -- copy pointer value
VAL BYTE DelP.OpC      IS #22(BYTE): -- delete pointer value
VAL BYTE Split.OpC     IS #23(BYTE): -- split control flow
VAL BYTE Switch.OpC    IS #24(BYTE): -- switch control flow
VAL BYTE Jump.OpC      IS #25(BYTE): -- route control flow
VAL BYTE Wait.OpC      IS #26(BYTE): -- wait for synchronization
VAL BYTE Const.OpC     IS #27(BYTE): -- generate long constant

-- function call instructions
VAL BYTE FastCall.OpC  IS #30(BYTE): -- standard function call
VAL BYTE TailCall.OpC  IS #31(BYTE): -- tail-recursive function call
VAL BYTE Context.OpC   IS #32(BYTE): -- submit own result address
VAL BYTE Apply.OpC     IS #33(BYTE): -- request activation
VAL BYTE Send.OpC      IS #34(BYTE): -- send argument to callee
VAL BYTE SendC.OpC     IS #35(BYTE): -- send constant argument to callee
VAL BYTE Submit.OpC    IS #36(BYTE): -- request submission of curried arg.
VAL BYTE Return.OpC    IS #37(BYTE): -- return result value
VAL BYTE ReturnC.OpC   IS #38(BYTE): -- return constant result value
VAL BYTE Terminate.OpC IS #39(BYTE): -- terminate activation

-- storage instructions
VAL BYTE Alloc.OpC     IS #40(BYTE): -- allocate tuple
VAL BYTE Read.OpC      IS #41(BYTE): -- read tuple field
VAL BYTE WriteI.OpC    IS #42(BYTE): -- write integer into tuple field
VAL BYTE WriteP.OpC    IS #43(BYTE): -- write pointer into tuple field
VAL BYTE WriteC.OpC    IS #44(BYTE): -- write integer constant into tuple f.

---------------------------------------------------------------------------
-- End of ADPROC.INC
```

```
----------------------------------------------------------------------------
```

References

Bibliography

[Abelson and Sussman, 1985] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, 1985.

[Aho *et al.*, 1986] Alfred A. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Arvind and Culler, 1986] Arvind and D. E. Culler. Dataflow Architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[Arvind and Culler, 1987] Arvind and David E. Culler. Resource Requirements of Dataflow Programs. Computation Structures Group Memo 280, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December 1987.

[Arvind and Ekanadham, 1987] Arvind and Kattamuri Ekanadham. Future Scientific Programming on Parallel Machines. Computation Structures Group Memo 272, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1987. Revised February 1988.

[Arvind and Iannucci, 1987] Arvind and Robert A. Iannucci. Two Fundamental Issues in Multiprocessing. Computation Structures Group Memo 226-6, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1987. Also in: Proceedings of the DVLR-Conference on Parallel Processing in Science and Engineering, Bonn/Bad Godesberg, June 25–26, 1987.

[Arvind and Nikhil, 1988] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. Computation Structures Group Memo 271, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 1988. Also in: PARLE 87, Conference on Parallel Architectures and Languages Europe, Eindhoven, The Netherlands, June 15–19, 1987, Volume 259 of Lecture Notes in Computer Science, Springer, Berlin.

[Arvind *et al.*, 1987] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data Structures for Parallel Computing. Computation Structures Group Memo 269, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1987. Also in: Proceedings of the Workshop on Graph Reduction, Los Alamos, New Mexico, September 28 – October 1, 1986.

[Arvind *et al.*, 1988a] Arvind, David E. Culler, and Gino K. Maa. Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs. Computation Structures Group Memo 279, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1988. Revised June 1988. To appear in the *International Journal of Supercomputer Applications.*

[Arvind *et al.*, 1988b] Arvind, Steve Heller, and Rishiyur S. Nikhil. Programming Generality and Parallel Computers. Computation Structures Group Memo 287, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1988. Also: Fourth International Symposium on Biological and Artificial Intelligence Systems, Trento, Italy, September 18–22, 1988.

[Augustsson, 1987] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Backus, 1978] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bevan, 1987] D. I. Bevan. Distributed Garbage Collection Using Reference Counting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

[Bird and Wadler, 1988] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Series in Computer Science. Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

[Bjornson *et al.*, 1987] Robert Bjornson, Nicholas Carriero, David Gelernter, and Jerrold S. Leichter. Linda, the Portable Parallel. Research Report YALE/DCS/RR-520, Department of Computer Science, Yale University, New Haven, Connecticut, February 1987. Revised January 1988.

[Boyle, 1989] James M. Boyle. Deriving Parallel Programs from Pure Lisp Specifications by Program Transformation. Lecture held at RISC-Linz, Johannes Kepler University, Linz, Austria, June 1989.

[Buchberger, 1978] Bruno Buchberger. Computer-Trees and their Programming. In Dauchet, editor, *4th Colloquium "Trees in algebra and programming"*, pages 1–18, University of Lille, February 16–18, 1978. Also: Technical Report 78-11, RISC-Linz, Johannes Kepler University, Linz, Austria, 1978.

[Buchberger, 1985] Bruno Buchberger. The L-Machine: An Attempt at Parallel Hardware for Symbolic Computation. In J. Calmet, editor, *3rd International Conference on Algebraic Algorithms and Error-Correcting Codes*, volume 229 of *Lecture Notes in Computer Science*, pages 333–347, Grenoble, France, July 15–19, 1985. Springer. Also: Technical Report 85-24, RISC-Linz, Johannes Kepler University, Linz, Austria, 1985.

[Burstall *et al.*, 1980] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, Stanford University, Stanford, California, August 25–27, 1980. ACM Press. Also: Internal Report CSR-62-80, Department of Computer Science, University of Edinburgh, May 1980, Update February 1981.

[Church, 1941] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, New Jersey, 1941.

[Cousineau *et al.*, 1985] Guy Cousineau, Gèrard Huet, and Larry Paulson. *The ML Handbook.* University of Cambridge, May 1985.

[Dennis, 1984] Jack B. Dennis. Data Flow Computation. In *Control Flow and Data Flow — Concepts of Distributed Programming, International Summer School*, pages 343–398, Marktoberdorf, West Germany, July 31 – August 12, 1984. Springer, Berlin.

[Foster and Taylor, 1989] Ian Foster and Stephen Taylor. *Strand — New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Funk, 1989] Gerhard Funk. *A Verification System for Total Correctness of the Parallel L-Language.* PhD thesis, Johannes Kepler University, Linz, Austria, June 1989. Also: Technical Report 89-26, RISC-Linz, Johannes Kepler University, Linz, Austria, 1989.

[Gao *et al.*, 1988] Guang R. Gao, Rene Tio, and Herbert H. J. Hum. Design of an Efficient Dataflow Architecture without Data Flow. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 861–868, Japan, 1988. ICOT.

[Goldberg, 1988] Benjamin F. Goldberg. Multiprocessor Execution of Functional Programs. Research Report YALEU/DCS/RR-618, Department of Computer Science, Yale University, April 1988.

[Gurd *et al.*, 1985] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[Harper *et al.*, 1988] R. Harper, R. Milner, and M. Tofte. The Definition of Standard ML — Version 2. Technical Report LFCS-88-62, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, 1988.

[Herath *et al.*, 1985] Jayantha Herath, Nobuo Saito, Kenji Toda, Yoshinori Yamaguchi, and Toshitsugu Yuba. NOT(OPERATION) for High Speed Dataflow Computer Systems. In *Proceedings of the First International Conference on Supercomputing Systems*, pages 524–533, St. Petersburg, Florida, December 16–20, 1985. IEEE Computer Society.

[Hiraki *et al.*, 1984] Kei Hiraki, Toshio Shimada, and Kenji Nishida. A Hardware Design of the SIGMA-1, a Data Flow Computer for Scientific Computations. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 524–531. IEEE Computer Society, August 1984.

[Hiraki *et al.*, 1986] Kei Hiraki, Kenji Nishida, Satoshi Sekiguchi, and Toshio Shimada. Maintenance Architecture and its LSI Implementation of a Dataflow Computer with a Large Number of Processors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 584–591. IEEE Computer Society, 1986.

[Hoare, 1985] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1985.

[Hudak *et al.*, 1990] Paul Hudak, Philip Wadler, Arvind, Brian Boutel, John Fairbairn, Joseph Fasel, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, and Jonathan Young. Report on the Programming Language Haskell – A Non-strict, Purely Functional Language (Version 1.0). Research report, Department of Computing Science, University of Glasgow, Glasgow, UK, April 1990.

[Iannucci, 1988] Robert Alan Iannucci. A Dataflow / von Neumann Hybrid Architecture. Technical Report MIT/LCS/TR-418, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1988.

[Inmos, 1988a] Inmos. *Occam 2 Reference Manual.* Prentice Hall, New York, 1988.

[Inmos, 1988b] Inmos Limited, Bristol, UK. *The Transputer Databook*, November 1988.

[Jensen, 1988] Jensen & Partners International. *TopSpeed Modula-2 for IBM Personal Computer and Compatibles, User's Manual*, 1988.

[Johnson, 1985] Thomas Johnson. Lambda Lifting: Transforming Programs to Recursive Equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, France, September 16–19, 1985. Springer, Berlin.

[Johnsson, 1987] Thomas Johnsson. *Compiling Lazy Functional Languages, Part I.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Kusche, 1988] Klaus Kusche. LL User Manual. Master's thesis, Johannes Kepler University, Linz, Austria, 1988. Also: Technical Report 88-25, RISC-Linz, Johannes Kepler University, Linz, Austria, 1988.

[Landin, 1966] P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[Loidl, 1992] Hans Wolfgang Loidl. Compiling Functional Programs to Dataflow Code. In *Proceedings of the Workshop on Parallel Computers and Programming Languages*, number 10 in Communications of the German Computer Society, Group for Parallel Algorithms, Computer Architectures and System Software, pages 78–86, Dagstuhl Castle, Germany, February 26–28, 1992. Also: Technical Report ACPC/TR 92-10, ACPC Technical Report Series, Austrian Center for Parallel Computation, July 1992.

[McCarthy, 1960] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machines. *Communications of the ACM*, 3(4):184–195, April 1960.

[McGraw *et al.*, 1985] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. SISAL — Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2. Technical Report M-146, Rev. 1, Lawrence Livermore National Laboratory, University of California, March 1985.

[Nikhil, 1987] Rishiyur S. Nikhil. Id World Reference Manual (for Lisp Machines). Computation Structures Group Memo 280, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1987.

[Nikhil, 1988] Rishiyur S. Nikhil. ID (Version 88.0) Reference Manual. Computation Structures Group Memo 284, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1988.

[Papadopoulos and Culler, 1990] Gregory M. Papadopoulos and David E. Culler. Monsoon: An Explicit Token-Store Architecture. In *The 17th International Symposium on Computer Architecture*, number 18(2) in ACM SIGARCH Computer Architecture News, pages 82–91, Seattle, Washington, May 28–31, June 1990.

[Papadopoulos, 1988] Gregory Michael Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1988.

[Peyton Jones, 1987] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, New York, 1987.

[Rudalics, 1988a] Martin Rudalics. *Multiprocessor List Memory Management.* PhD thesis, Johannes Kepler University, Linz, Austria, 1988. Also: Technical Report 88-87, RISC-Linz, Johannes Kepler University, Linz, Austria, 1988.

[Rudalics, 1988b] Martin Rudalics. Multiprocessor Reference Counting. In Chris Jesshope and K. D. Reinartz, editors, *CONPAR 88*, pages 32–41, Manchester, England, September 12–16, 1988. Cambridge University Press, Cambridge, UK.

[Ruggherio and Sargeant, 1987] C. A. Ruggherio and J. Sargeant. Control of Parallelism in the Manchaster Dataflow Machine. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 1–15, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

[Schreiner, 1989] Wolfgang Schreiner. Report on the EVE Programming Language. Technical Report 89-43, RISC-Linz, Johannes Kepler University, Linz, Austria, November 1989.

[Schreiner, 1990] Wolfgang Schreiner. ADAM — An Abstract Dataflow Machine and Its Transputer Implementation (Extended Abstract). In *Workshop on Parallel Computation*, Schloß Seggau at Graz, Austria, June 24–30, 1990.

[Schreiner, 1991] Wolfgang Schreiner. ADAM — An Abstract Dataflow Machine and Its Transputer Implementation. In *Distributed Memory Computing, 2nd European Conference, EDMCC2*, pages 392–401, Munich, Germany, April 22–24, 1991. Volume 487 of Lecture Notes in Computer Science, Springer, Berlin. Also: Technical Report 90-53, RISC-Linz, Johannes Kepler University, Linz, Austria, November 1990. Also: Technical Report 91-3, Austrian Center for Parallel Computation, January 1991.

[Shapiro, 1987] Ehud Shapiro, editor. *Concurrent Prolog, Collected Papers, Volumes 1 and 2.* MIT Press, Cambridge, Massachusetts, 1987.

[Shimada *et al.*, 1986] Toshio Shimada, Kei Hiraki, Kenji Nishida, and Satosi Sekiguchi. Evaluation of a Prototype Data Flow Processor of the SIGMA-1 for Scientific Computation. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 226–234, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society.

[Siegl, 1990a] Kurt Siegl. A Parallel Version of Buchberger's Algorithm in STRAND88. Technical Report 90-17.0, RISC-Linz, Johannes Kepler University, Linz, Austria, April 1990.

[Siegl, 1990b] Kurt Siegl. Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages. Master's thesis, Johannes Kepler University, Linz, Austria, 1990. Also: Technical Report 90-54, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990.

[Tevanian Jr. *et al.*, 1987] Avadis Tevanian Jr., Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young. Mach Threads and the Unix Kernel: The Battle for Control. Technical Report CMU-CS-87-149, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA, August 1987.

[Traub, 1986] Kenneth R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report MIT/LCS/TR-370, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1986.

[Turner, 1985] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, September 16–19, 1985. Springer, Berlin.

[Yamaguchi *et al.*, 1983] Yoshinori Yamaguchi, Kenji Toda, and Toshitsugu Yuba. A Performance Evaluation of a Lisp-Based Data-Driven Machine (EM-3). In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 363–369, Stockholm, Sweden, 1983. IEEE Computer Society.

[Yamaguchi *et al.*, 1984] Yoshinori Yamaguchi, Kenji Toda, Jayanta Herath, and Toshitsugu Yuba. EM-3: A Lisp-Based Data-Driven Machine. *Future Generation Computer Systems*, pages 524–532, November 1984.

[Yuba *et al.*, 1983] Toshitsugu Yuba, Yoshinori Yamaguchi, and Toshio Shimada. A Control Mechanism of a Lisp-Based Data-Driven Machine. *Information Processing Letters*, pages 139–143, April 1983.

[Yuba *et al.*, 1985] Toshitsugu Yuba, Toshio Shimada, Kei Hiraki, and Hiroshi Kashiwagi. SIGMA-1: A Dataflow Computer for Scientific Computations. *Computer Physics Communications*, pages 141–148, 1985.

[Yuba, 1986] Toshitsugu Yuba. Research and Development Efforts on a Dataflow Computer Architecture in Japan. *Journal of Information Processing*, 9(2):51–60, September 1986.