1994

An Occam2 implementation of Prolog

Manjula Motwani

An Occam2 Implementation of Prolog

by

Manjula H. Motwani

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by: _____
Graduate Advisor - Dr. Tony Chang

_____
Department Head - Dr. Roy Czernikowski

_____
Committee Member - Dr. Andrew Kitchen

DEPARTMENT OF COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
JANUARY 1994

THESIS RELEASE PERMISSION FORM

ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF ENGINEERING

Title of Thesis: An Occam Implementation of Prolog.

I, Manjula H. Motwani, hereby grant permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part.

Signature:_____

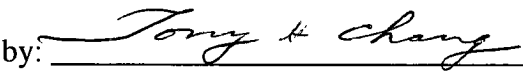
Date: 2/28/94

An Occam2 Implementation of Prolog

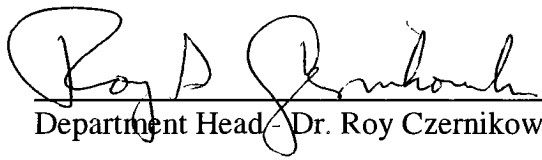
by

Manjula H. Motwani

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by: 

Graduate Advisor - Dr. Tony Chang



Department Head - Dr. Roy Czernikowski

Committee Member - Dr. Andrew Kitchen

DEPARTMENT OF COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
JANUARY 1994

THESIS RELEASE PERMISSION FORM

ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF ENGINEERING

Title of Thesis: An Occam Implementation of Prolog.

I, Manjula H. Motwani, hereby grant permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part.

Signature: MH Motwani

Date: 2/28/94

Abstract

Prolog has been widely recognized as a powerful programming language for artificial intelligence. It was also chosen as a kernel language for the Japanese Fifth Generation Project. The project is a large scale effort to initiate a new generation of computing. Due to the wide range of applications that Prolog has, many methods have been developed for extracting parallelism from standard Prolog in order to achieve faster execution on a multiprocessor.

This project designs an execution model for Prolog, which attempts to exploit the parallelism mainly at the argument level through the unification operation. The model consisting of a number of virtual machine instructions, has been implemented in Occam2 on a Transputer Development System. A few Prolog procedures have been hand compiled to the virtual machine instructions, and have been run on a Transputer Development System with a single transputer. This model of virtual machine instructions can be applied to a multiple transputer system. This project gives the details of the implementation of the virtual machine instructions.

Table of Contents

Table of Contents (Continued)

Table of Contents (Continued)

Table of Contents (Continued)

List of Figures

List of Figures (Continued)

1.0 INTRODUCTION

Prolog is used as a powerful programming language for artificial intelligence and symbolic computing. It has been chosen as the machine language of the Japanese Fifth Generation Computer System. The basic mechanisms that it uses include pattern matching, tree based data structuring and automatic backtracking.

The concept of parallel prolog has been a major topic in recent conferences on Computer Architecture. Many computational models have been developed to improve the execution speed of Prolog programs, which has been a major drawback of Prolog. Since Prolog programs are non procedural, the Von Neumann architecture is not suitable to run Prolog programs. Instead we need a multiprocessor architecture to exploit the parallelism inherent in Prolog programs.

The classic sequential model of Prolog uses pattern matching and unification operations. The unification creates a binding of variables to terms. As the program executes it forms a tree which is expanded in a depth-first manner. A prolog program consists of facts and rules. A fact like parent(tom,lisa) is always unconditionally true. A rule has a condition part (the right hand side of the rule) and a conclusion part (the left hand side of the rule). The conclusion part is called the head of a clause and the condition part the body of a clause. If the condition is true, then a logical consequence of it is the conclusion. When a particular clause is unified, bindings are produced for the head. The goals of the clause are then executed from left to right with the same bindings as those produced for the head. To execute a goal the system searches for the first clause that unifies with the goal. If any of the goals fails, the system backtracks undoing any bindings made for that clause. It then looks for another clause to match.

Example

Assume we have the following rule -

grandparent(X,Z) :- parent(X,Y),parent(Y,Z).

We now pose a query grandfather(jack,jane). This clause unifies with the clause grandparent(X,Z) and produces a binding of X to jack and Z to jane. The goals of the clause, parent(X,Y) and parent(Y,Z) will be executed with these bindings. The system will then look for a clause that unifies with parent(jack,Y). If it succeeds in finding one, it produces a binding of Y to a term. The system then looks for a goal that unifies with parent(Y,jane). If this succeeds the original query succeeds.

This simple model was used in the earlier days when Prolog did not have such a wide range of applications. But since the need for efficient AI languages has increased, we need to investigate the possibilities of parallelism in Prolog and replace the sequential model by a parallel model.

My goal is to study the possibilities of parallelism in Prolog and to design an execution model for Prolog which uses parallelism at the argument level through the unification operation. The model will consist of five virtual machine instructions, which will be implemented in Occam2 and run on a Transputer Development System. The virtual machine instructions are unify, decompose, check, instantiate and construct. All Prolog procedures are hand compiled into the virtual machine. The scheme that I have proposed to execute a Prolog procedure in parallel, consists of converting each Prolog procedure into a dataflow graph of the virtual machine instructions. The instructions are operated in a data driven fashion. The instructions on the same horizontal level in a graph can be executed in parallel. Since these instructions operate on different arguments it is

sometimes called argument parallelism. Argument parallelism is realized by processing multiple arguments in parallel.

I intend to use Occam2 as the programming language since it offers an easy way of running concurrent processes. Since prolog is a general purpose programming language, general purpose processors should be chosen for the implementation of the multiprocessor system. Transputer, an Inmos series of processor has been chosen for the implementation of the system.

2.0 PROLOG - AN OVERVIEW

2.1 INTRODUCTION

Prolog stands for programming in Logic. It has its roots in mathematical Logic or predicate calculus. It is generally used for problems involving structured objects and relations between them. In general programming in Prolog involves defining relations and asking queries about those relations.

A Prolog program consists of clauses. There are 3 types of clauses facts, rules and questions.

1) A fact is always unconditionally true. It has the form <head>.

Example

male(tom).

parent(tom,liz).

sister(jill,jack).

2) A rule specifies things that may be true if a particular condition is satisfied. Thus rules have a condition and a conclusion part. The conclusion part is called the head of a clause and the condition the body of a clause. A rule has the form <head> :- <tail>. A logical consequence of the tail is the head. The tail may be a conjunction or list of goals to be satisfied. All the goals have to be satisfied for the conclusion to be true.

Example

child(Y,X) :- parent(X,Y).

mother(X,Y) :- parent(X,Y),female(X).

3) A question is a means of querying the existing relations. A relation can be specified by facts which are objects that satisfy the relation or by rules that define the relation.

Example

If we have the following rules and facts in our database -

male(tom).

parent(tom,jack).

father(X,Y) :- parent(X,Y),male(X).

Now if we pose the following queries

| ?- male(X). Prolog answers this query by giving the value of X

X = tom;

no.

| ?- father(tom,jack). Prolog answers this query by saying

yes.

Thus facts are clauses that have an empty body, rules have a head and a body and questions have a body only.

A procedure is a set of clauses about the same relation.

Example

predecessor(X,Z) :- parent(X,Z).

predecessor(X,Z) :- parent(X,Y),predecessor(Y,Z).

A prolog program can contain variables. A variable is a literal beginning with an uppercase letter. During the execution of the program a variable can be substituted by another object. The variable then has a value and is said to be instantiated.

2.2 ANSWERING QUESTIONS IN PROLOG

A query is the key to executing programs in Prolog. A query is a sequence of goals. In order for a query to be true all the goals need to be satisfied. A goal is a logical consequence of the facts and rules present in the program. If the query contains variables

Prolog tries to find what values the variables can take on for the goals to be true. The values that the variables take on are then displayed. If the goals of the query cannot be satisfied then Prolog answers no to the query.

Example

If we have the following facts and rules in our database -

```
male(tom).
male(jack).
parent(tom,jill).
parent(jack,jane).
father(X,Y) :- parent(X,Y),male(X).
```

If we pose the following queries -

| ?- male(X). Prolog answers this query by giving the value of X

```
X = tom;
X = jack.
yes.
```

If we type a semicolon after the first result, Prolog will look for another goal to satisfy the query. If it does find one then the result is displayed or else it says no.

| ?- parent(peter,jill). Prolog answers this query by saying

```
no.
```

| ?- father(tom,jill). Prolog answers this query by saying

```
yes.
```

2.3 UNIFICATION

Prolog classifies data objects as shown in figure - 2.1. A constant is an identifier that starts with a lowercase letter, e.g. amy, 3, jill. A variable is an identifier that starts with an uppercase letter, e.g. X, Result. Prolog uses variables like any other programming

language. Generally a common programming language uses the assignment operation to assign a value to a variable. Prolog uses unification or matching to assign values to variables. A structure is an object that has several components. A structure is identified by its functor and arity. The functor is a literal starting with a lowercase letter. The arity is the number of components the structure contains. The components of a structure can be constants, variables or structures. e.g. date(1,2,3) - The functor of this structure is date and the arity is 3. The components of the structure are 1,2,3. All data objects are called terms in Prolog.

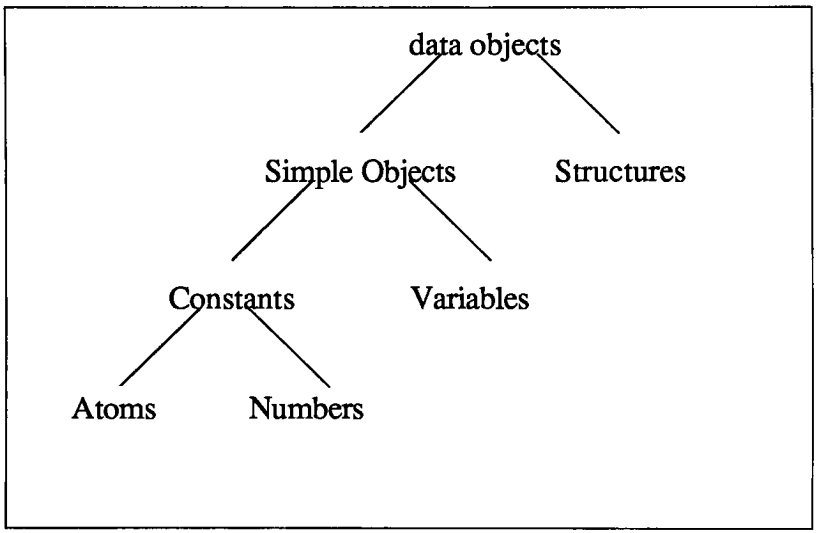


Figure 2.1 - Classification of Prolog terms

The most important operation that can be performed on terms is unification or matching. Two terms S and T can be unified if they are identical or the variables in both the terms can be instantiated to objects in such a way that they become identical after the substitution. The general rules to decide if 2 terms M and N unify are as follows-

1) If M and N are constants they unify only if they are identical or the same object.

2) If M is a variable and N is any term, then they unify and M is instantiated to N.

3) If N is a variable and M is any term, then they unify and N is instantiated to M.

4) If M and N are structures then they unify if

a) M and N have the same principal functor and arity, and

b) all the corresponding components unify or can be unified.

This is a brief overview of Prolog and it covers the most important facts of Prolog.

3.0 OCCAM - A BRIEF OVERVIEW

3.1 INTRODUCTION

Occam was designed to program parallel computers. It was developed at Inmos Ltd. as the lowest level language for programming transputers. A transputer is a powerful microprocessor which has serial input/output interfaces, with the help of which it can communicate with other transputers. The transputer was designed specifically to implement the constructs of Occam. Occam evolved from Communicating Sequential Processes (CSP) designed by C.A.R. Hoare. The major difference between Occam and any other sequential programming language is that in Occam a user can create processes that can run in parallel and can communicate with each other via channels.

There are actually 2 different languages referred to as Occam. The original language is known as Occam and a development of the language is Occam2. The language Occam2 has a better set of program structures and a means of describing the types and structures of data items used and communicated by a program. The major differences between Occam and Occam2 are -

1) Occam2 is a typed language. The standard types used are INT, BYTE and BOOL.

2) Channels may also be typed in Occam2. Channels can also carry a sequence of differently typed data. This has to be then declared with the help of a protocol.

3) In Occam2 new types can be created with the TYPE declaration.

4) A CASE construct is provided in Occam2 to allow for input selection.

5) Multi-dimensional arrays are possible in Occam2.

3.2 PROCESSES

The basic unit of an Occam program is called a process. Each process may be written as a single line which indicates a single action to be performed or a number of lines bound together by process constructors which indicate a combination of actions to be performed. Processes can be classified as follows -

3.2.1 SINGLE LINE PROCESSES

These processes indicate single actions to be performed. The simplest of these single line processes are assignment, input and output. The assignment process consists of the name of a variable, an assignment sign ":=" and an expression, e.g.- X := 3. The input and output processes involve channels. The output process consists of a channel name and a value to be output on the channel. The value could be an expression which is evaluated and the result is then placed on the channel, e.g. - cha ! 3. This process outputs the value 3 on channel cha. The input process consists of a channel name and a variable to receive the value input on the channel, e.g. cha ? i. This process assigns the value on channel cha to the variable i.

Another important one line procedure available is the procedure or subroutine call. The syntax of this process is the procedure name followed by a set of parameters enclosed in parenthesis e.g. sum(arg1,arg2,arg3). A procedure which has no parameters needs an empty pair of parenthesis.

There are two other one line processes that are available to the programmer that do nothing. They are SKIP and STOP. The process SKIP terminates as soon as it starts. In Occam the programmer is obliged to write SKIP when he does not want any action to be performed. This is contrary to other programming languages where the programmer

need not write anything if he does not want any action to be performed. The process STOP starts but it does not proceed and it does not terminate. A sequential process that is stopped cannot do anything but a parallel process that is stopped can still perform some actions. It is not used very often in programs, but it is the most ideal thing to do when unexpected errors are encountered, because it ensures that the process which has failed is brought to a standstill without affecting other processes.

3.2.2 CONSTRUCTIONS

All other processes are constructed out of the fundamental single line processes, and they are known as constructions. Constructions include sequences, conditionals, loops, parallels and alternations.

3.2.2.1 SEQUENTIAL PROCESSES

A sequential process is made up of a number of processes that are performed one after the other in the sequence in which they are written. It is written as the single word SEQ, followed on consecutive lines by all the processes to be performed. These processes are indented exactly 2 spaces to the left from the keyword SEQ. A sequence starts with the start of its first process. Each subsequent process starts when the process preceding it terminates. The sequence terminates on the termination of the last process.

Example

```
SEQ
  x := 4
  y := x + 18
  x := x + 6
```

3.2.2.2 CONDITIONALS

These processes make decisions based on the values of variables. A conditional process consists of the keyword IF followed by several components each indented 2 spaces to the right. Each component consists of a Boolean expression and is called a guarded choice. The Boolean expression can be evaluated to TRUE or FALSE. Below each condition is a process which is indented a little further to the right.

The conditional executes by evaluating the booleans in sequence until one is found which yields the value TRUE. After one is found, no more booleans are evaluated. The process which immediately follows the TRUE boolean is executed. The conditional terminates when this process terminates. If none of the boolean expressions evaluates to TRUE then the conditional behaves like a STOP i.e. it does not terminate. Thus in order to avoid this we need to use TRUE as the last boolean. Thus if none of the booleans evaluates to TRUE then the process after the last boolean TRUE will be executed. Leaving it out is syntactically correct and will not cause a compile time error, but may cause the program to stop while running.

Example

```
IF
  (i < 0)
    SEQ
      y := y + 5
      z := y + 8
  (i = 0)
    y := y + 3
  (i > 0)
    SEQ
```

```
        y := y + 4
        z := y + 6
    TRUE
      SKIP
```

3.2.2.3 LOOPS

There are 2 kinds of loops in Occam : unbounded WHILE loops and indexed bounded FOR loops. Unbounded loops have to be executed sequentially but bounded loops need not be run sequentially.

An unbounded loop is written with the keyword WHILE followed by a boolean expression. Below this is a process indented to the right. The while loop is executed by evaluating the value of the boolean expression, if the boolean expression evaluates to TRUE the process following it is started. When the process terminates the boolean expression is evaluated again, if it evaluates to TRUE the process is executed again. This goes on till the boolean expression evaluates to FALSE. Once the boolean expression evaluates to FALSE the execution of the WHILE loop terminates.

Example

```
WHILE (i >= 8)
    SEQ
      i := i + y
```

A bounded loop can be considered to be an array of processes and can be made with any of the SEQ, IF, PAR and ALT constructors by putting a replicator of the form name = base FOR count after the keyword and a process following it. The base and count are expressions of type INT. The meaning of such a FOR loop is the same as that of a construction formed with the same keyword followed by count copies of the component

with the name taking on values base, base+1,....,base+count-1 in successive copies.

Example

```
SEQ j = 30 FOR 15
    celebrate(j)
```

This stands for a sequential composition of 15 processes .

```
SEQ
    celebrate(30)
    celebrate(31)
    ....
    celebrate(44)
```

So SEQ-FOR loops are equivalent to FOR loops in any other language. The bodies of parallel PAR-FOR loops are executed concurrently and can be considered as arrays of parallel processes.

3.2.2.4 PARALLEL PROCESSES

To perform several processes simultaneously the key word PAR is written with all the processes written one after another, below the keyword and indented 2 spaces to the right.

Example

```
PAR
  ..... code for the first process
  ..... code for the second process
```

In this example process1 and process2 are run concurrently. If one of them has to have a priority over the other we write

```
PRI PAR
  ..... code for high priority process
  ..... code for low priority process
```

A process in PRI PAR will only run if all the processes above it are either stopped, terminated or waiting for input/output. Replicated PARs can also used. They have the form -

```
PAR j = 0 FOR count
    worker(j,work.to slave[j])
```

This creates count number of concurrent processes with values of j ranging from 0 to count-1.

3.2.2.5 ARBITRATING PROCESSES

An alternative is like a conditional where the choice depends on whether another process is doing an output. An alternative is written with the keyword ALT and a list of guarded processes below it, each indented 2 places to the right. Each guarded process has an input which is the guard followed by a process.

Example

```
ALT
  up ? increase
    x := x + increase
  down ? decrease
    x := x - decrease
  read ? req
    x := x * req
```

The process is executed by waiting for another process to perform an output on one of the channels up, down or read. When there is an input on any one of the channels

the corresponding process gets executed. If several inputs are ready at the same time then only one of the processes is chosen for execution. Replicated ALTs can also be used and they provide input from any one of an array of channels.

3.3 METHODS OF COMMUNICATING BETWEEN PROCESSES

There are a number of ways of communicating between processes. The most commonly uses techniques are through variables and channels. Conventional programming languages use variables to communicate with other processes whereas concurrent processes used channels to communicate.

3.3.1 VARIABLES

In conventional programming languages a process modifies the value of a variable which can then be read by another process executed in sequence with the first. A variable is a name with a type associated with it. The variable can hold a value only of the type associated with it. Communication through variables is sequential. The value of a variable can be changed by assignment. The variables can be declared as local variables, which implies that their scope is limited to the process where they are declared, or global or shared variables which implies that the variables can be used in the entire program. Processes that use local variables use parameter passing to communicate with other processes.

3.3.2 CHANNELS

Concurrent processes use channels to communicate between themselves. A channel is a means of passing messages from one process to another. In parallel programs the equivalent of a variable is a channel. Occam uses message passing between concurrent processes. Two or more processes may not access any concurrent data if any one of them

attempts to change the data. Any sharing of information between such processes is done through message passing.

A process can pass data to another concurrent process by an output process sync ! info, where sync is the name of the channel and info is the data to be passed. This process is called an output . The receiving process accepts the data with an input process - sync ? data , where sync is the name of the channel and data is the variable that receives the value of the message.

Thus a channel is a uni-directional, point-to-point and unbuffered means of communication. A channel cannot be used to broadcast messages. A message put onto a channel by an output process can be received only by one input process. If a message has to be sent to many processes then multiple copies of the same message have to be output. A channel cannot be used to send information in both directions. Since a channel is unbuffered, a momentary synchronization is required for an output action and its corresponding input. Both must start before either can terminate.

A channel has to be declared like any other variable, e.g. CHAN OF INT test - test is a channel of INT and can only pass messages which consist of a single INT. Two processes can use the channel to pass messages -

```
CHAN OF INT test:
PAR
  SEQ    --process 1
    ---
    test ! 45
    ----
```

```
INT temp:  --process 2
SEQ
  ----
  test ? temp
```

Process 1 sends a message to process 2. The message consists of a single integer 45. The process 2 inputs that message and assigns it to variable temp. The overall effect is the assignment operation - temp := 45. Processes that share data do not cause problems as long as no process alters the data, but the problem starts as soon as one process tends to change information used by another process.

3.4 TRANSPUTERS AND OCCAM

A transputer is a single chip microcomputer that contains processor, memory and communication links which can provide point-to-point connections between transputers. A transputer can be used in a single processor system or in a network of transputers to build high performance concurrent systems.

Transputers can be programmed in most high level languages. Occam is designed as the low level language of the transputer. To gain most benefit from the transputer architecture, the whole system can be programmed in Occam. This provides all the advantages of a high level language, maximum program efficiency and the ability to use the special features of the transputer. Occam provides a very good framework for designing concurrent systems using transputers. A program running on a transputer is equivalent to an Occam process, so that a network of transputers can be described directly as an Occam program. Every transputer implements the Occam concepts of concurrency and communication. The transputer supports Occam in two ways : one can run a whole Occam program on a single transputer, or one can run processes on separate processors.

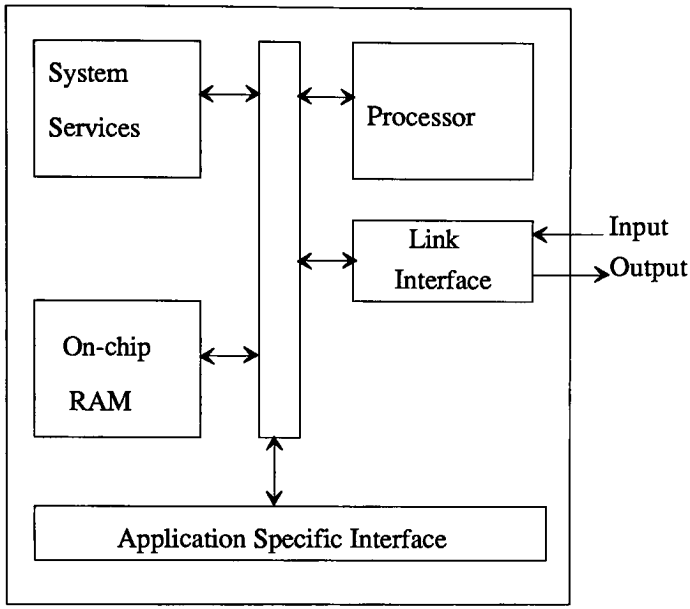


Figure 3.1 - The Transputer Architecture

When Occam is used to program an individual transputer, the transputer shares its time between the concurrent processes and channel communication is implemented by moving data within the memory. When Occam is used to program a network of transputers, each transputer executes the processes allocated to it. Communication between Occam processes on different transputers is implemented directly by transputer links. Processes running on a single processor can use as many channels as needed, but all processes on one processor have to confine their communication with extemal processes to four input and four output channels. A transputer has four links and each link can support one input and one output. The transputer links carry information serially. Passing a message in one direction involves some handshaking with a few bits passed as acknowledgments in the other direction. The transputers are joined together by connecting their links. A transputer does not have to have all its links connected. A connected pair of links can support at most one channel in each direction. Thus the same Occam program can be implemented on

a variety of transputer configurations, with one configuration optimized for cost, another for performance or another for a balance of cost and performance.

The transputers and Occam were designed together. All transputers include special instructions and hardware to provide maximum performance and optimal implementations of the Occam model of concurrency and communications. All transputer instruction sets are designed to enable simple, direct and efficient compilation of Occam. Programming of I/O interrupts and timing is standard on all transputers and conforms to the Occam model.

4.0 PARALLELISM IN PROLOG

4.1 INTRODUCTION

It was assumed that Prolog programs were to be executed sequentially on a Von Neumann machine. But the semantics of Prolog allows it to be executed on a number of processors in parallel. Sequential Prolog uses the order of goals in a clause and the order of clauses in the program to control the search for a proof. The chosen goal is always the leftmost goal. Given a goal A1, A2, ..., An and a program P, Prolog sequentially searches for the first clause in P whose head unifies with A1, and reduces the goal using this clause. It then tries from left to right to solve the reduced goal. If it ever fails to solve a goal it backtracks to the last choice of a clause made. Sequential Prolog resembles a conventional sequential programming language. In order to improve the execution speed of computers we can exploit the parallelism inherent in Prolog and execute it in parallel. Conery points out the following five types of parallelism OR parallelism, AND parallelism, stream parallelism, unification parallelism and search parallelism.

4.1.1 OR PARALLELISM

OR Parallelism is possible where a goal will unify with the head of more than one clause. Sequential Prolog uses backtracking to cover these possibilities, but a multiprocessor could start them all at once. These OR clauses do not consume values from each other so they can run with a minimum of synchronization.

Let us assume we have the following facts and rules in our database -

```
f(X,Y,Z) :- p(X,Z),q(Y,Z).
p(1,3).
p(1,4).
q(2,4).
```

Let us pose the query | ?- f(1,2,Z). In sequential Prolog the goals of the "f" clause

will be executed from left to right in the order which they appear. The following steps take place in sequential Prolog.

1) The first goal of the "f" clause matches the first "p" clause and Z is instantiated to 3.

2) The next step is to execute the goal q(Y,Z). Since Z is instantiated to 3 we have to look for a clause q(2,3) in order for the main goal to succeed.

3) Since we do not have a clause q(2,3) in our database it fails and the system backtracks undoing the instantiation for Z. The system backtracks to look for another clause to satisfy p(1,Z).

4) The system then finds another clause p(1,4) and Z will be instantiated to 4.

5) The system will now try to find a clause q(2,4) to satisfy the final goal of the "f" clause . It finds a clause q(2,4), Thus the main goal succeeds and the answer will be Z=4.

With a multiprocessor, a process is created for every alternative choice. When we start executing the query we can start two processes for p at the same time. The two processes have different values for Z. The first process has a value of 3 for Z, and the second process has a value of 4 for Z. The second goal "q" is tried with Z=3 and fails. We don't need to backtrack since we have another value of Z available. Thus no backtracking is needed in OR parallelism. It is the easiest form of Parallelism.

4.1.2 AND PARALLELISM

The concurrent execution of subgoals in the body of a clause is referred to as AND parallelism. It is the most complicated form of parallelism. The problem with AND parallelism is that the bindings of variables are often interrelated so binding conflicts may arise. This kind of parallelism involves a great deal of communication. We will discuss some of the difficulties involved in section 4.2.

Let us assume we have the following facts and rules in our database -

```
f(X,Y,Z) :- p(X,Z),q(Y,Z).
```

p(1,3).

p(1,4).

q(2,4).

Now let us pose the query | ?- f(A,B,4). We have seen in the example of OR parallelism how this will be executed in sequential Prolog. In AND parallelism the goals "p" and "q" of the clause "f" will start executing simultaneously. That is the system will start 2 processes simultaneously. The first process will look for the clause p(A,4) and the second process will look for the clause q(B,4). The first process will return an instantiation of A = 1 and the next process will return an instantiation of B = 2. This will be done in half the time it takes sequential Prolog to find an answer to a similar query.

4.1.3 STREAM PARALLELISM

Stream Parallelism involves the pipelining of structured data, like lists. One procedure creates a data structure and another procedure consumes the data structure while it is still being produced. Thus if two functions are to be applied to a list, stream parallelism would have a processor pass elements one by one after application of the first function to a second processor that applies the second function. Assume we have the following rule in our database -

f(List) :- p(List),q(List).

If we pose a query | ?- f([3,4,5,6,7]). The goal "p" will start processing the list. After it has finished processing the head of the list, i.e. 3, a processor will hand this element over to the goal "q". The goal "q" will then start processing the element. Thus "p" produces the list and "q" consumes the list.

4.1.4 UNIFICATION PARALLELISM

This involves processing multiple arguments in parallel. If a clause has more than one argument, the arguments can be processed in parallel. The processing of arguments involves decomposition of structured data and unification of two operands. The operands could be structured data, variables or ground terms.

4.1.5 SEARCH PARALLELISM

This is used for large databases. The database is partitioned into disjoint sets. Concurrent processes are used to search the sets separately. This raises the question about the best way to distribute Prolog information across a multiprocessor interconnection.

4.2 AND PARALLELISM - PROBLEMS ENCOUNTERED

And Parallelism is the most complicated form of Parallelism in Prolog. In this section we will describe the various problems encountered in AND parallellism and the next section describes the various approaches to solve the problems with AND parallelism.

Suppose we have the following rules and facts in out database -

```
f(X,Y) :- p(X),q(Y) .
p(jack).
p(james).
q(mary).
```

Let us pose the query : | ?- f(jack,mary). In this case it is possible to check if p(jack) and q(mary) are true in parallel. This is the most practical and simplest case of AND parallelism. With this case the performance achieved by using AND parallelism is high.

Suppose we have another simple rule in our database

f(X,Y,Z) :- p(Y,X),q(Z,X).

Now let us pose the query | ?- f(X,jack,mary). The goals p(jack,X) and q(mary,X) cannot be executed in parallel since they share a variable X. Parallel execution of the two goals may cause X to be instantiated to two different values. Thus we have a binding conflict here. To solve this we use the sequential model of Prolog. First p(jack,X) is executed, returning a value for X. Then q(mary,'value of X) is executed. If this fails we backtrack and work with any other value of X produced by p(jack,X). Thus from this we conclude that if the goals of a clause share one or more variables they cannot be executed in parallel.

The conclusion stated above is not enough to solve binding conflicts. Consider the first example, here it appears that X and Y are independent variables and that p and q do not depend on each other and can thus be executed in parallel. But this is not the case. Suppose we pose the following queries -

| ?- f(Z,Z).

| ?- f(Z,g(Z)).

| ?- f(g(Z),h(2,Z)).

Then at run time X and Y will be aliases of each other or wil have values that share at least one variable. That is X and Y have to be instantiated to the same object or at least have one variable in common which has to be instantiated to the same object. Thus for the first query f(Z,Z) if p(X) and q(Y) are executed in parallel X will be instantiated to jack and Y will be instantiated to mary. But since X and Y represent the same object we have a binding conflict and we conclude that though the 2 clauses do not share a variable they cannot be executed in parallel.

Now if we have the following rule in our data base -

f(X) :- p(X),q(X).

It is generally impossible at compile time to determine whether or not p(X) and q(X) can execute in parallel without creating a binding conflict. If f is called with a ground argument (any term not containing a variable), such as f(4), then the two subgoals can be run in parallel. Since the argument to f contains no variables , no binding conflicts can arise by the parallel execution of p and q. Thus even though the 2 goals share the variable X they can be executed in parallel since X is ground before a call is made to the two goals.

The situation can be even more complex if we consider the following rule -

f(X) :- p(X),q(X),s(X).

If at run time f is called with a ground argument, as in the call f(4) then all three subgoals can execute in parallel. But if f is called with a non-ground argument, p(X) must execute first and q(X) and s(X) wait. If the call to p(X) instantiates X to a ground term then q(X) and s(X) can execute in parallel. But if X is still uninstantiated after completion of p, then q(X) and s(X) must execute sequentially. Thus depending on the call and execution, three different situations can result.

Thus in general we can state that - if two or more goals share a variable they can be executed in parallel only if that variable is ground before the goals are called. We have just discussed the binding conflicts that arise when trying to execute two or more goals in parallel. The next section deals with the different approaches taken to solve these problems.

4.3 DIFFERENT APPROACHES

4.3.1 INTRODUCTION

A number of approaches have been taken in order to solve the binding conflicts problem. In this chapter, we will describe some of them. Without getting into the details of the implementation we will see some of the solutions.

The most dominant solution that has been proposed is a data flow approach, that requires information from the user. It requires the user to annotate some variables in the clauses. The goals involving these variables will then have to wait until they are fully instantiated. A clause that binds a value to a variable is called a "producer" and one that uses that bound variable is called a "consumer". This approach has been taken in Concurrent Prolog(Shapiro, Ehud Y.), Parlog(Clark,K.L. and Gregory,S.) and IC-Prolog(Clark,K.L. and McCabe,G.). The main goal of Logic Programming is hiding control and related issues from the user. The annotation solution although acceptable, does not meet the main goal.

The second approach Restricted AND parallelism tries to solve the binding conflicts problem with minimum information from the user. It requires a compile-time and run-time analysis.

4.3.2 CONCURRENT PROLOG

Concurrent Prolog was designed by Ehud Y. Shapiro to support concurrent programming and parallel execution. It incorporates guarded-command indeterminacy, data-flow synchronization and a commitment mechanism.

Concurrent Prolog adds 2 syntactic constructs to logic programs. Read-only annotations of variables X?, and the commit operator "|". Both are used to restrict the order in which the goals can be reduced, and restricting the choices of clauses that can be used to reduce them.

A concurrent Prolog program is a finite set of guarded clauses. A guarded clause has the form -

A :- G1,G2,....,Gm|B1,B2,...,Bn, m,n >= 0

where the G's and B's are atomic goals. The G's are called the guard of the clause and the B's are called its body. When the guard is empty the commit operator is not used. Any clause may contain variables marked "read-only".

The commit "|" operator achieves an effect similar to cut in sequential Prolog, but has a cleaner semantics due to its symmetry. It reads like a conjunction : A is implied by the G's and B's. Given a goal A1, A1 reduces to B if A1 unifies with A in the clause A :- G|B and following unification G terminates successfully.

A read-only term is written as follows - X?. The unification of such terms is an extension to normal unification. The unification of a read-only term X? with a term Y is defined as follows. If Y is non-variable then the unification succeeds only if X is non-variable and X and Y are recursively unifiable. If Y is a variable then the unification of X? and Y succeeds and the result is a read-only variable.

This definition implies that being read-only is not an inherited property, i.e. variables that occur in a read-only term are not necessarily read-only. Thus the scope of a read-only annotation is only the principal functor of a term, but not its arguments.

This also implies that the success of a unification may be time-dependent i.e. a unification that fails now, due to violation of a read-only constraint may succeed later, after the principal functor of a shared read-only variable is determined by another process, in which this variable does not occur as read-only.

4.3.3 RESTRICTED AND PARALLELISM

Restricted AND parallelism is used to overcome the binding conflicts that arise while using AND parallelism. The restricted AND parallelism model is a compromise between Conery's model which uses run-time support and the activation mode solution model which uses compile-time support. Conery's model uses a set of elaborate run-time algorithms. These algorithms dynamically compute parallel execution graphs based on data dependencies between subgoals. It can execute both deterministic and non-deterministic programs in parallel. This model achieves optimal AND parallelism but involves enormous run-time overhead. The activation mode solution relies on the information supplied by the programmer, that is the activation mode. Each unification involves matching a goal term (procedure call) against a clause head (procedure entry point). Each of the arguments in the clause head is assigned a mode, this helps the compiler to generate code for the above two cases. A "-" mode guarantees that the argument will match against an uninstantiated variable. A + mode guarantees that the match will already be bound. A ? mode guarantees nothing. The following code of the predicate insert which inserts a number into a sorted list illustrates the above scheme -

```
mode insert(+,+,-).
insert(X,[A|L],[A|NewL]) :- A=<X,!,insert(X,L,NewL).
insert(X,L,[X|L]).
```

This solution is not completely user transparent. It requires input from the user. Another problem is that there are no run-time checks involved.

The Restricted AND parallelism approach is a compromise solution between the run-time (Conery) and the compile-time (Activation Mode) solutions. The most important advantage of this method is that it requires much simpler run-time support. This method involves the compile time creation of a parallel execution graph expression for each program clause. Only one expression per clause is created. Since it computes only one execution graph it may fail to detect some potential for parallelism.

5.0 EXECUTION MODEL FOR PROLOG

5.1 INTRODUCTION

In this chapter I will describe the virtual machine instructions used to generate the execution model for Prolog. They are coded in Occam2 as individual procedures, which are independent from each other, and run on a Transputer Development System in a Unix environment residing on a SUN workstation.

5.2 MAJOR DECISIONS TAKEN TO GENERATE THE VIRTUAL MACHINE

5.2.1 GRAMMAR

Since my goal is to study the possibilities of parallelism in Prolog, the virtual machine generated will not be a commercial product but will be more of a research project. The virtual machine will run a subset of Prolog . The simplified grammar used is as follows -

<rule>	: =	<clause>. | <unit_clause>.
<clause>	: =	<head> :- <tail>.
<head>	: =	<goal>
<tail>	: =	<goal> { ,<goal> }
<unit_clause>	: =	<goal>
<goal>	: =	<functor> (<term> { ,<term>}) | <functor>
<functor>	: =	<identifier staring with a lower case letter>
<term>	: =	<constant> | <variable>
<constant>	: =	<integer>
<constant>	: =	< identifier starting with a lower case letter>
<variable>	: =	<identifier starting with an upper case letter>

An example program would be -

append([], L2, L2).

append([Head|L1], L2, [Head|L3]) :- append(L1,L2,L3).

5.2.2 PROLOG PROCEDURES

All Prolog procedures are hand compiled into the virtual machine which consists of a number of instructions. I have written these instructions in Occam2 and proposed a scheme in which these instructions can be operated in a data driven fashion i.e. when the input arguments of an instruction are available, the instruction can be executed. Each Prolog procedure is converted into a dataflow graph of the virtual machine instructions. The instructions on the same horizontal level in a graph can be executed in parallel. Since these instructions operate on different arguments it is sometimes called argument parallelism. Argument parallelism is realized by processing multiple arguments in parallel.

5.3. VIRTUAL MACHINE INSTRUCTIONS

The execution model consists mainly of five virtual machine instructions. They are decompose, unify, instantiate, check and construct. Few other instructions are needed to check the input, store the input and print out the results.

5.3.1. DECOMPOSE

This instruction tries to decompose the input term. In Prolog '|' is used to denote decomposition. If the term is a list the instruction will decompose the list into a head and a tail. The head is a single element, whereas the tail is a list. The head represents the first element of the list, while the tail represents the remaining elements. If the original list contains only one element then the tail will be an empty list. The tail is stored as a new list in the data area. If the input term is a shared or non-shared variable, the instruction generates two non shared variables H and T which denote the list Head and Tail. It then

creates a new binding message with the input variable and the binding [H|T] and stores this binding message in the stored binding environment. The instruction generates a fail if the input term is neither a list nor a variable.

5.3.2 UNIFY

This instruction unifies two input operands and outputs a binding environment. This is the most important operation that can be performed on terms. Two terms S and T can be unified if they are identical or the variables in both the terms can be instantiated to objects in such a way that they become identical after the substitution. This instruction identifies six different situations of its two operands and generates a binding environment.

1) If the two operands are empty lists, the instruction does not generate any binding messages.

2) If one operand is a variable, the instruction generates only one binding message in the binding environment. The binding message consists of the variable and the other operand as its binding instance.

Example

Operand	Value	Type
1	X	Shared Variable
2	[1,day(Y,3,4),jack]	List

Result

Binding environment

Variable	Value
X	[1,day(Y,3,4),jack]

In the above example the variable X is unified to the list [1,day(Y,3,4),jack].

3) If both operand are lists, the instruction checks if they have the same arities (number of elements). It then unifies the corresponding elements of both the lists and generates a binding environment. If all the elements of the two lists can be unified successfully the instruction succeeds, else it generates a fail.

Example

Operand	Value	Type
1	[jack,father(peter,X),20]	List
2	[Y,father(peter,mary),20]	List

Result

Binding environment

Variable	Value
X	mary
Y	jack

4) If both operands are structured data, the instruction checks if they have the same functors. It then proceeds to unify the components of the structures. It treats the components of the structures as lists and calls the procedure to unify lists.

Example

Operand	Value	Type
1	jack([1,2,Y],X,mary)	Structure
2	jack([1,2,[3,4]],[6,john],Z)	Structure

Result

Binding environment

Variable	Value
Y	[3,4]
X	[6,john]
Z	mary

5) If both operands are ground terms, the instruction checks if they are identical to each other. If they are it does not generate any binding messages. If they are not identical it generates a fail.

6) If none of the above situations are satisfied the instruction generates a fail.

5.3.3 INSTANTIATE

This instruction receives a term and the index of the stored binding environment. If the input term is a ground term the instruction does nothing but output the same term, otherwise the instruction instantiates the shared variables occurring in the term according to their binding instances in the stored binding environment, and then outputs the instantiated term.

Example

Variable	Value
A	7

X	3
Z	date(A,8)
Y	[X,jack(Z,4),A]

The instruction will not process the terms A and X since they are bound to ground terms. The term Z is bound to a structure. The instruction will instantiate all the components of the structure. If any of the components is a variable, the instruction will check the binding environment to see if the variable is bound to some term. It will then instantiate the variable in the structure to its binding. In the above example the term A is bound to 7, thus the instruction will instantiate the term A in the structure date(A,8) to 7 and the value of Z will be date(7,8). The same procedure is repeated with the term Y. The instruction will instantiate all the terms in the binding of Y, thus the term Y will have a value [3,jack(date(7,8),4,7].

5.3.4 CHECK

This instruction checks each binding message (X B) in the stored binding environment to see if in the binding instance B there are one or more shared variables which have their own binding instances in other binding messages in the temporary binding environment, if so, the instruction then replaces the shared variables by these binding instances. A stored binding environment is shared by all related instructions. A temporary binding environment is used by a unify instruction. A check instruction is executed after a unify instruction. If the unify instruction is successful, the temporary binding environment is transferred to the stored binding environment. The instruction also checks the binding consistencies for the shared variables. It checks if the shared variables with the same identifier have been bound to the same non-variable instances.

5.3.5 CONSTRUCT

This instruction takes two inputs, a list head and a tail list, to construct a new list.

5.4 DATAFLOW GRAPHS

To show how the execution model structures Prolog programs we will take a look at some Prolog procedures and their dataflow representations.

5.4.1 APPEND

This procedure appends the first list argument to the second list argument to give the third list argument.

append([], L2, L2).

append([Head|L1],L2,[Head|L3]) :- append(L1,L2,L3).

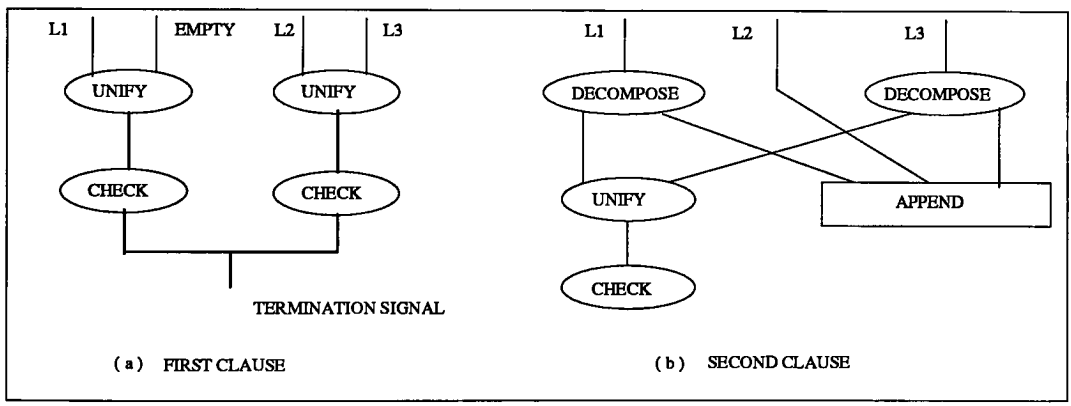


Figure 5.1.1 - Dataflow Representation of Append.

The execution flow charts for the two append clauses are as shown in figure 5.1.1. The instructions on the same horizontal level in a graph can be executed in parallel. In figure - 5.1.1(a), which corresponds to the first append clause, the operations of unifying list1 with empty and list2 with list3 can be done in parallel. In figure 5.1.1(b),

which corresponds to the second append clause, the operations of decomposing list1 and list3 can be done in parallel. The termination signal in figure - 5.1.2, fires three instantiate instructions to instantiate three arguments according to the stored binding environment and to generate the solution.

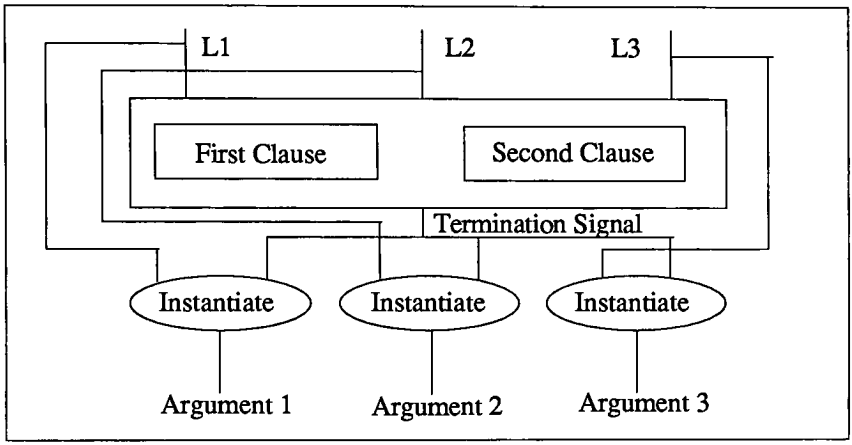


Figure 5.1.2 - Dataflow Representation of Append

5.4.2 DELETE

This procedure deletes a term L1 from a list L2 to produce a list L3. This procedure can be programmed as follows -

```
del(L1,[L1|L2],L2).
del(L1,[Head|L2],[Head|L3]) :- del(L1,L2,L3).
```

The first clause checks if term L1 is the head of the list L2. If it is then the result after deletion is the tail of the list L2. If this clause fails then the second clause decomposes the lists L2 and L3 and recursively calls delete again. If there are several occurrences of term L1 in list L2 then this procedures will be able to delete anyone of them by backtracking.

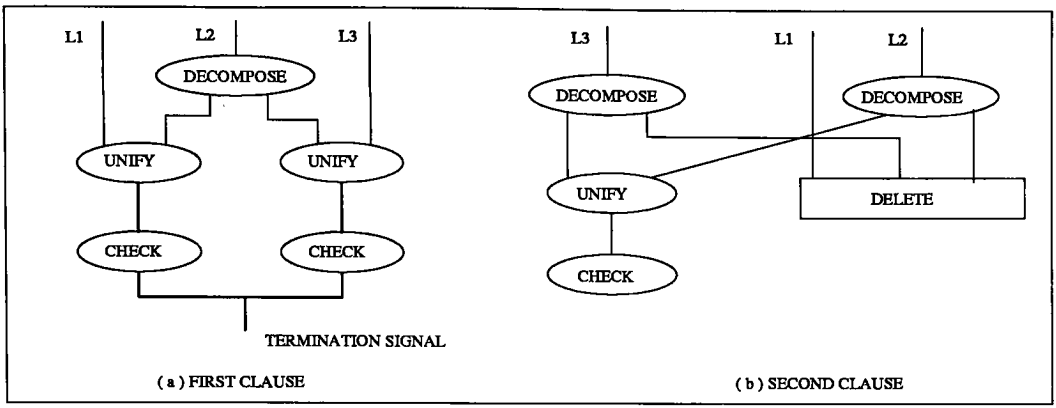


Figure 5.2 - Dataflow representation of delete

5.4.3 MEMBER

This procedure checks if a term L1 is a member of a list L2. The procedure can be programmed in Prolog as follows -

member(L1,[L1|L2]).

member(L1,[Head|L2]) :- member(L1,L2).

The term L1 is a member of the list L2 if either of the following conditions is satisfied -

1) L1 is the head of list L2.

2) L1 is a member of the tail of L2.

The first clause of member takes care of the first condition and the second clause of member takes care of the second condition.

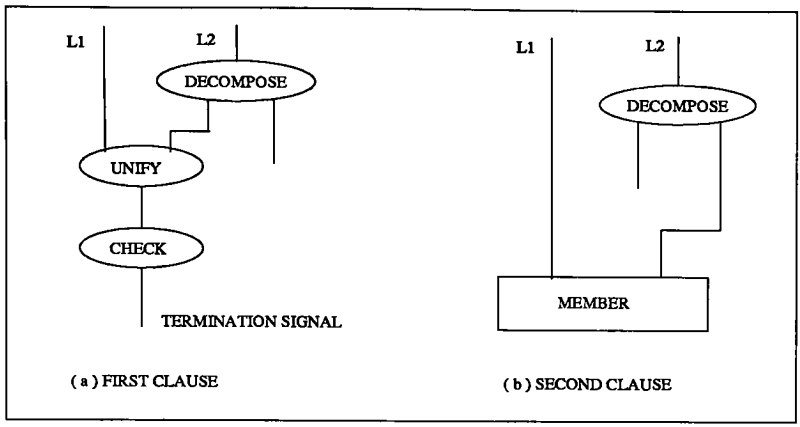


Figure 5.3 - Dataflow representation of member

5.4.4 REVERSE

This procedure reverses a list L1 to produce a list L2. This procedure is programmed in Prolog as follows -

```
reverse([],[]).
reverse([Head|L1],L2) :- reverse(L1,Term1),conc(Term1,[Head],L2).
conc([],L2,L2).
conc([Head|L1],L2,[Head|L3]) :- conc(L1,L2,L3).
```

The first clause of reverse checks for the end of recursion, i.e. when list L1 is empty. The second clause decomposes list L1 and recursively calls the procedure reverse. The function concatenate, concatenates list L2 to list L1 to give list L3.

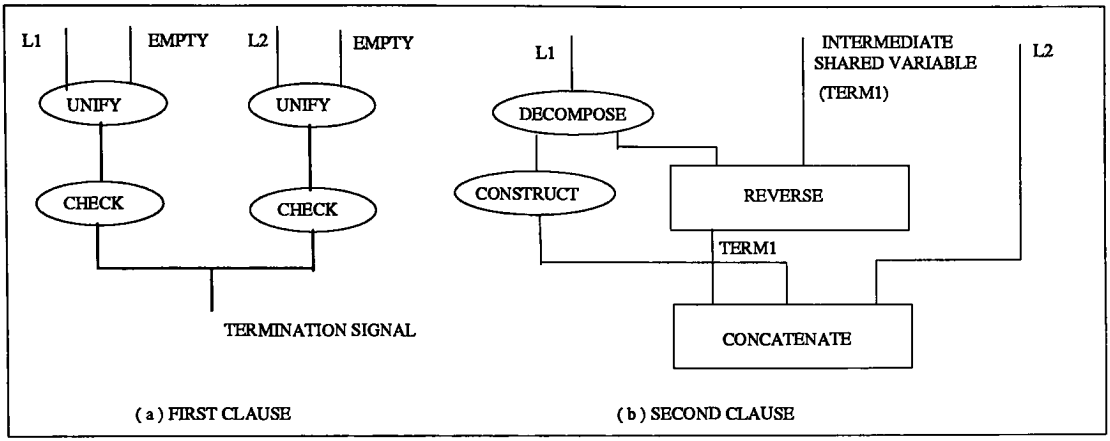


Figure 5.4 - Dataflow Representation of Reverse

In the above diagram I have introduced the concept of an intermediate variable. It is a variable that is only used in the body literals without occurrence in the head literals.

5.4.5 PERMUTATION

This procedure takes two arguments L1 and L2. If the argument L2 is a variable then the procedure instantiates L2 to a permutation of L1. It uses backtracking to produce all the permutations possible. If the argument L2 is a list then it checks if L2 is a permutation of L1. It is programmed as follows -

```
permutation([],[]).
permutation(L1,[L2|Tail]) :- del(L2,L1,Term1), permutation(Term1,Tail).
```

The first clause takes care of the following condition - If the first list is empty then the second list must also be empty. The second clause deletes an element L2 from the first list, permutes the rest of it obtaining a list Tail, and then adds L2 in front of Tail.

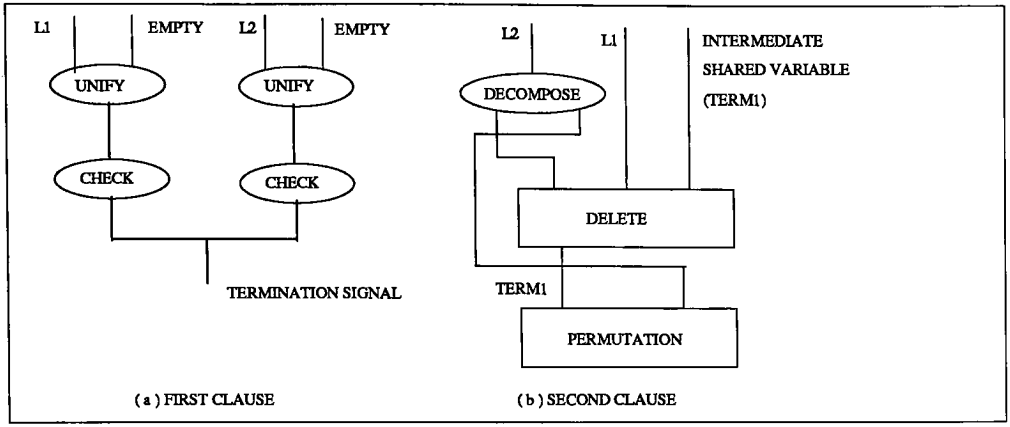


Figure 5.5 - Dataflow Representation of Permutation

5.4.6 PALINDROME

This function checks if a list is a palindrome. It is a palindrome if it reads the same in the forward and backward direction. It is programmed in Prolog as follows -

```
palindrome([]).
palindrome([_]).
palindrome([L1|Tail]) :- conc(Term1,[L1],Tail), palindrome(Term1).
```

The first clause checks if the list is empty. If the list is empty it is a palindrome. The second clause checks if the list has a single element. If it has a single element it is a palindrome. The working of the third clause can be explained with the help of an example. Let us assume the list has the following elements - m,a,d,a,m. The following sequence of events takes place with a call to the third clause -

1) Decompose the list m,a,d,a,m to produce a head - m and a tail - [a,d,a,m].

2) Call Concatenate to find a term such that when you concatenate m to it you should get a list [a,d,a,m].

3) Concatenate returns a result - [a,d,a].

4) Call Permutation with the result of Concatenate i.e. [a,d,a].

The effect of this is to check the first and last terms of the list. If concatenate fails to find a result it implies that the first and last terms do not match and the list is not a palindrome.

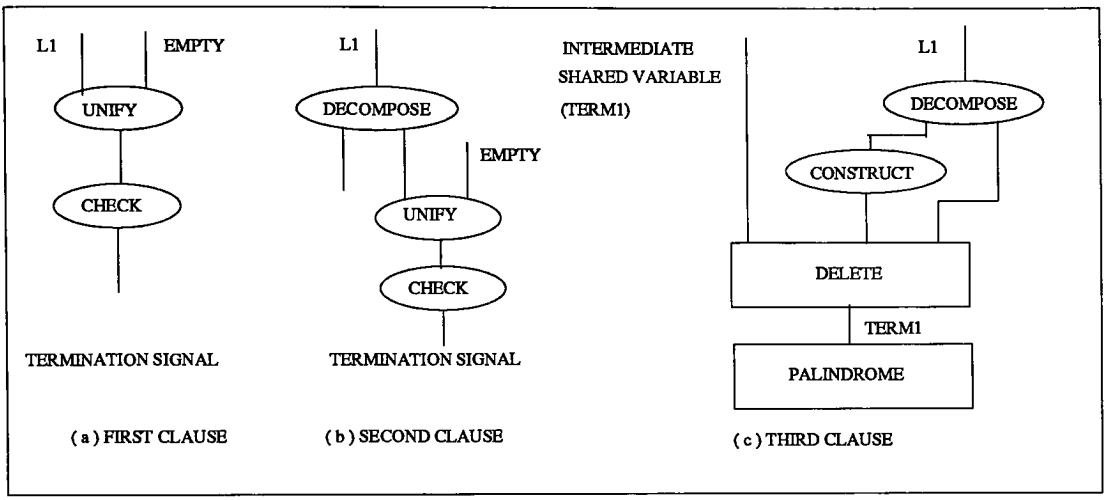


Figure 5.6 - Dataflow representation of Palindrome

5.4.7 SORTING

This procedure sorts a list if there is an ordering relation between the items in the list. It takes two arguments, the list to be sorted and the sorted list. It is programmed in Prolog as follows -

```
bubblesort(L1,L2) :- swap(L1,Term1),bubblesort(Term1,L2).
bubblesort(L1,L1).
swap([X,Y|Rest],[Y,X|Rest]) :- gt(X,Y).
swap([Z|Rest],[Z|Rest1]) :- swap(Rest,Rest1).
gt(X,Y) :- X > Y.
```

The sorting algorithm is based on the following idea -

- Find two adjacent elements, X and Y, in List such that gt(X,Y) is true and swap X and Y in List, obtaining List1, then sort List1.

- If there is no pair of adjacent elements, X and Y, in List such that gt(X,Y), then List is already sorted.

The purpose of swapping two elements, X and Y, that occur out of order, is that after swapping the new list is closer to a sorted list. After a sufficient amount of swapping we end up with all the elements in order.

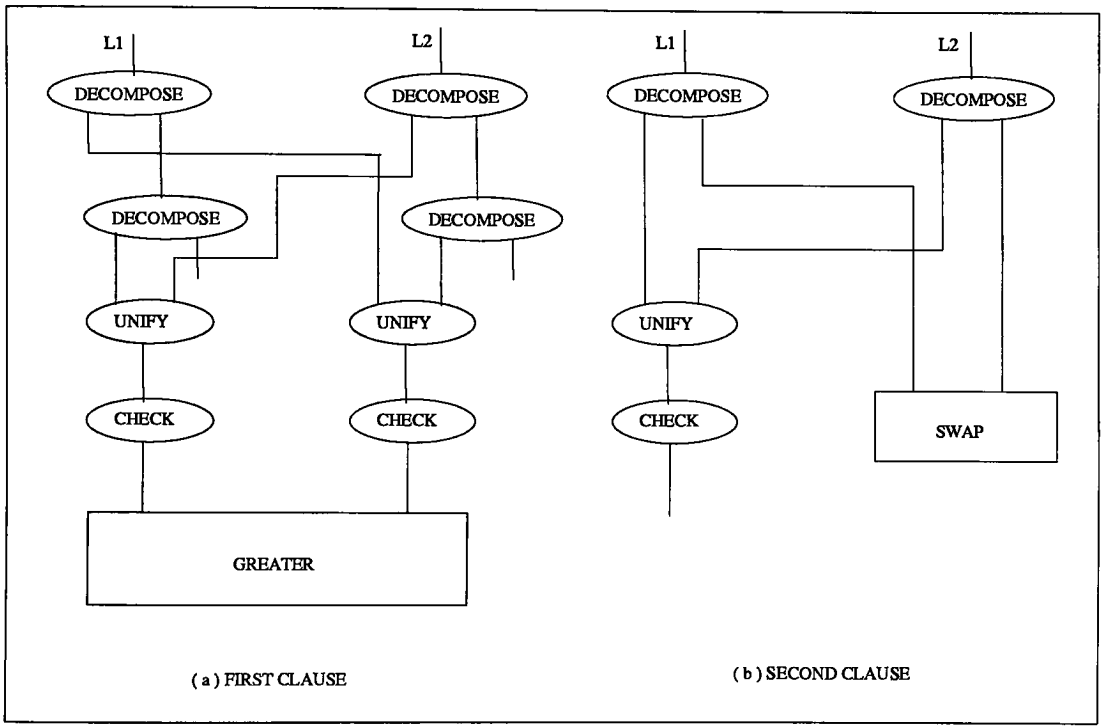


Figure 5.7.1 - Dataflow representation of Swap

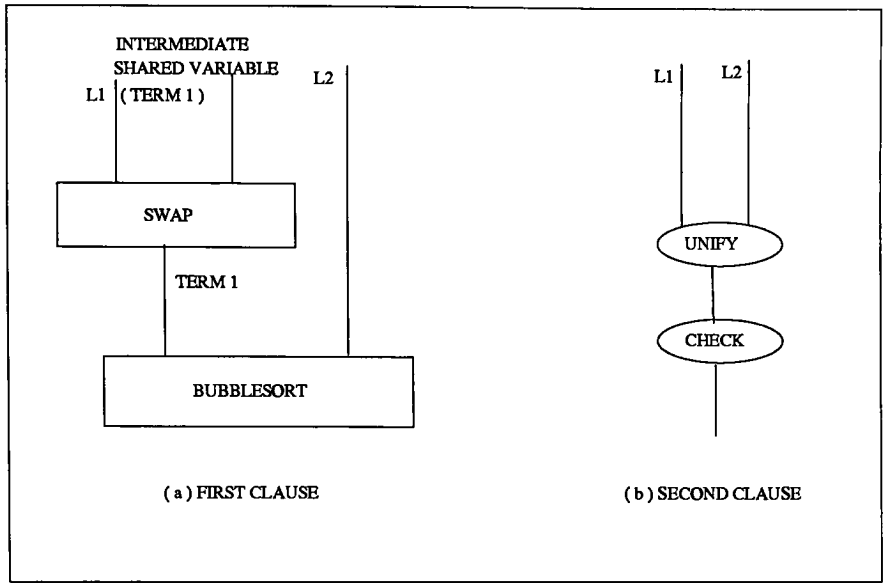


Figure 5.7.2 - Dataflow Representation of Bubblesort

5.4.8 SHIFT

This procedure shifts a list rotationally by one element to the left. It takes two arguments L1 and L2. If argument L2 is a variable, it will be instantiated to L1 shifted rotationally by one element to the left. If L2 is a list, the procedure checks if it is equal to L1 shifted to the left by one element. It is programmed in Prolog as follows -

```
shift([L1|Tail],L2) :- conc(Tail,[L1],L2).
```

This procedure decomposes the list L1 into a list head and tail. It then constructs a list from the head and concatenates this to the tail to give L2.

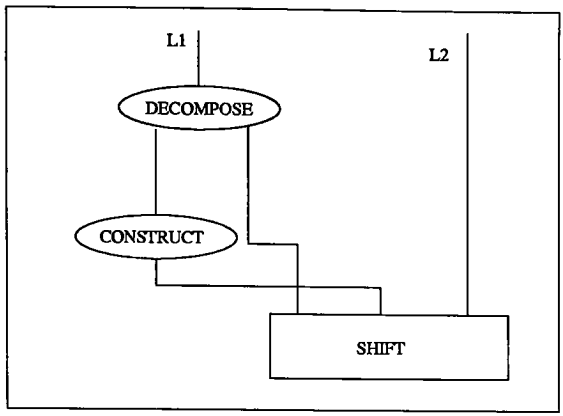


Figure 5.8 - Dataflow representation of shift

5.4.9 DIVIDELIST

This procedure partitions a list into two sublists. It takes three arguments L, L1 and L2. It partitions the elements of L between the lists L1 and L2 such that they are of approximately the same length. It is programmed in Prolog as follows -

```
dividelist([],[],[]).
dividelist([L],[L],[]).
dividelist([X,Y|L],[X|L1],[Y|L2])  :-  dividelist(L,L1,L2).
```

The first clause checks if list L is empty, if it is then list L1 and L2 are instantiated to empty. The second clause checks if list L has a single element, if it is true then the element is put in list L1 and list L2 is instantiated to empty. The third clause decomposes list L twice, puts one element in list L1 and the other element in list L2 and recursively calls dividelist. Thus if list L has an even number of elements, then lists L1 and L2 will be of the same length. If L has an odd number of elements list L1 will have one more element than list L2.

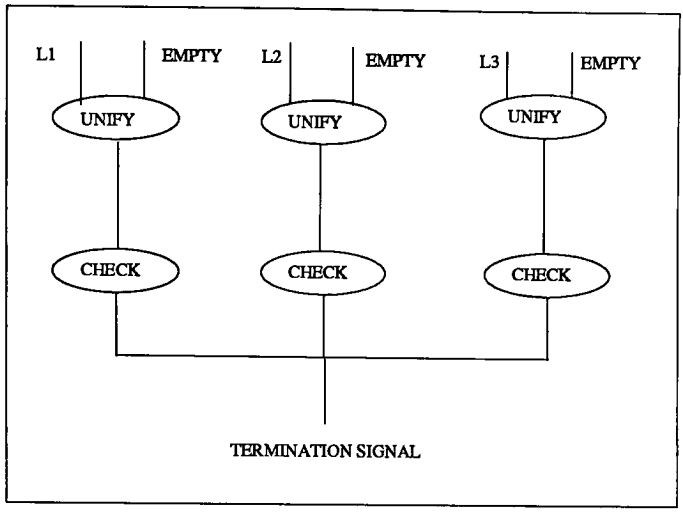


Figure 5.9.1 - Dataflow Representation for the first clause of Dividelist

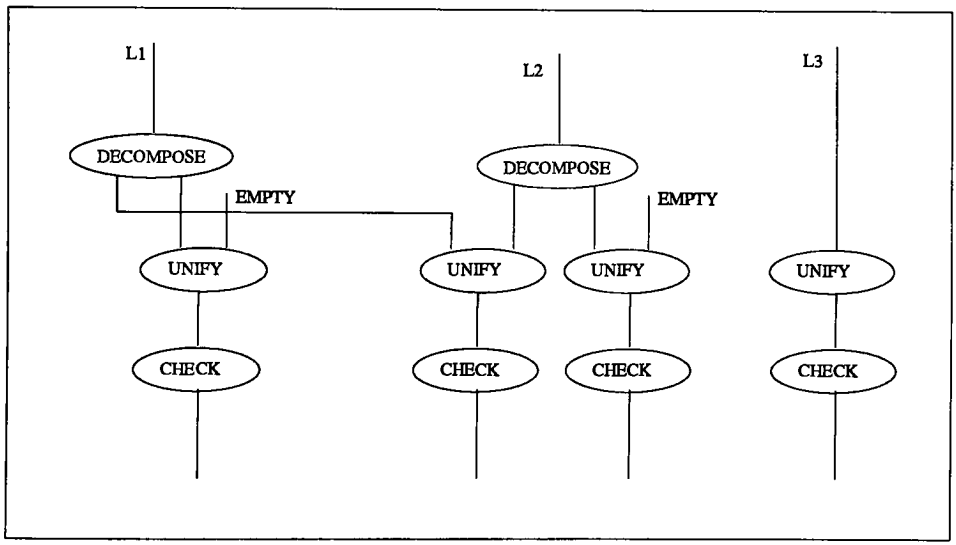


Figure 5.9.2 - Dataflow Representation for the second clause of Dividelist

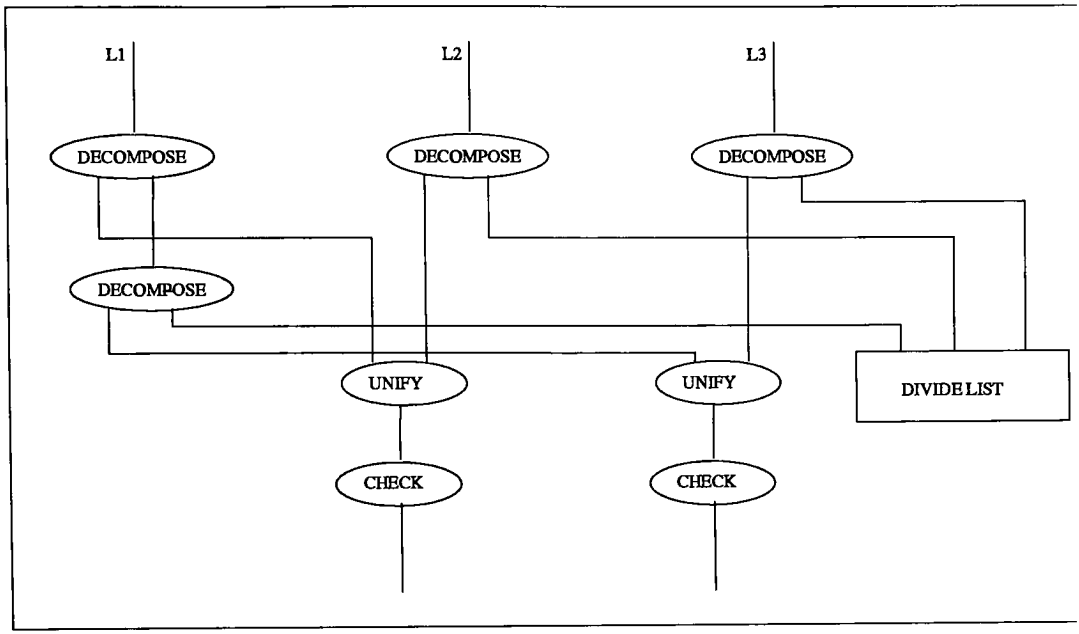


Figure 5.9.3 - Dataflow Representation for the third clause of Dividelist

5.4.10 SUBSET

This procedure takes two arguments L1 and L2. If the argument L2 is a variable then it instantiates it to all the subsets of list L1. If the argument L2 is a list, then the procedure checks if L2 is a subset of L1. It is programmed in Prolog as follows -

```
subset([],[]).
subset([First|L1],[First|L2]) :- subset(L1,L2).
subset([First|L1],L2) :- subset(L1,L2).
```

The first clause checks if list L1 is empty, if it is then list L2 is instantiated to empty. The second clause decomposes list L1 and retains the head of list L1 in the subset L2. The third clause also decomposes the list L1 but removes the head of list L1 from the subset L2.

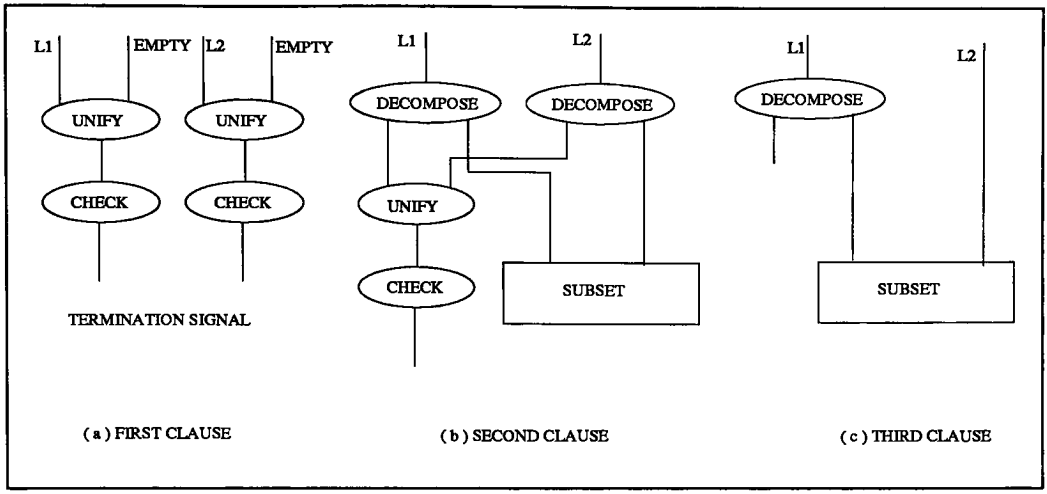


Figure 5.10 - Dataflow Representation of subset

6.0 THE IMPLEMENTATION OF THE MODEL

6.1 DATA REPRESENTATION

All Prolog terms are represented in a data area which is divided into eight fields. These eight fields represent the eight different types of data that can be used. The data area is implemented in Occam2 as a range of integers. Each data field occupies a subset of the range of integers. Each data term has an integer assigned to it. This integer implies two meanings - the term identifier for the field, and the index for the term content, for which the address may be obtained by subtracting an offset from the term.

The data area is implemented as shown in figure - 6.1. The different data fields used are as follows -

1) List field A list is a data structure which consists of a sequence of any number of terms. The terms could be integers, characters, strings, variables or structures. The list field in this model occupies a range from 1 to 400. A list counter keeps a track of the number of lists present. It is initialized to 1. Every time a list is created it is assigned the value of the list counter and the list counter is incremented by 1. This value identifies the list. Lists are represented as a two dimensional array of integers. The first dimension contains the list identifiers, the second dimension includes the list length (first array item) and the list components (all other items).

Example - [1,6,mary,X,date(month,day,year)].

2) Structure field - A structured object has a list of terms. The terms are grouped together with a functor. The terms can be integers, characters, strings, variables or structures. Thus a structure consists of a functor and a list. The structure field occupies a range from 401 to 800. A structure counter keeps track of the number of structures present. The structure counter is initialized to 401. Every time a structure is created it is assigned the value of the structure counter, and the value of the counter is then incremented by 1. Every structure

has an entry in the arrays functor length and functor list. They are one dimensional integer arrays. There is another array called functor which stores the functors. It is a one dimensional character array. The array functor length stores the length of each functor. The first element stores the length of the first functor, the second element stores the length of the second functor and so on. Each structure is represented by a number between 401 and 800. To get the corresponding entry in the array functor length, an offset of 400 is subtracted from the number representing the structure. The entry in the array functor list corresponds to the list where the structure terms are actually stored.

Example - father(jack,john), date(day(hour,minute,second),month,year)).

3) Shared Variable field Variables that occur in the head literal are called shared variables. They represent variable symbols. They are represented by a string of alphabetical characters starting with an uppercase letter. Shared variables occupy a range from 801 to 1200. Each shared variable is represented by a number between 801 and 1200. Each shared variable has an entry in the integer array called shared variable length. This entry is the length of the shared variable. The array shared variable stores the character string which represents the shared variable. The entry in the shared variable length array is obtained by subtracting an offset of 800 from the number representing the shared variable. A shared variable counter keeps track of the shared variables present. Every time a shared variable is created, it is assigned the value of the shared variable counter, and the value of the counter is then incremented by 1.

Example - John,X,Y.

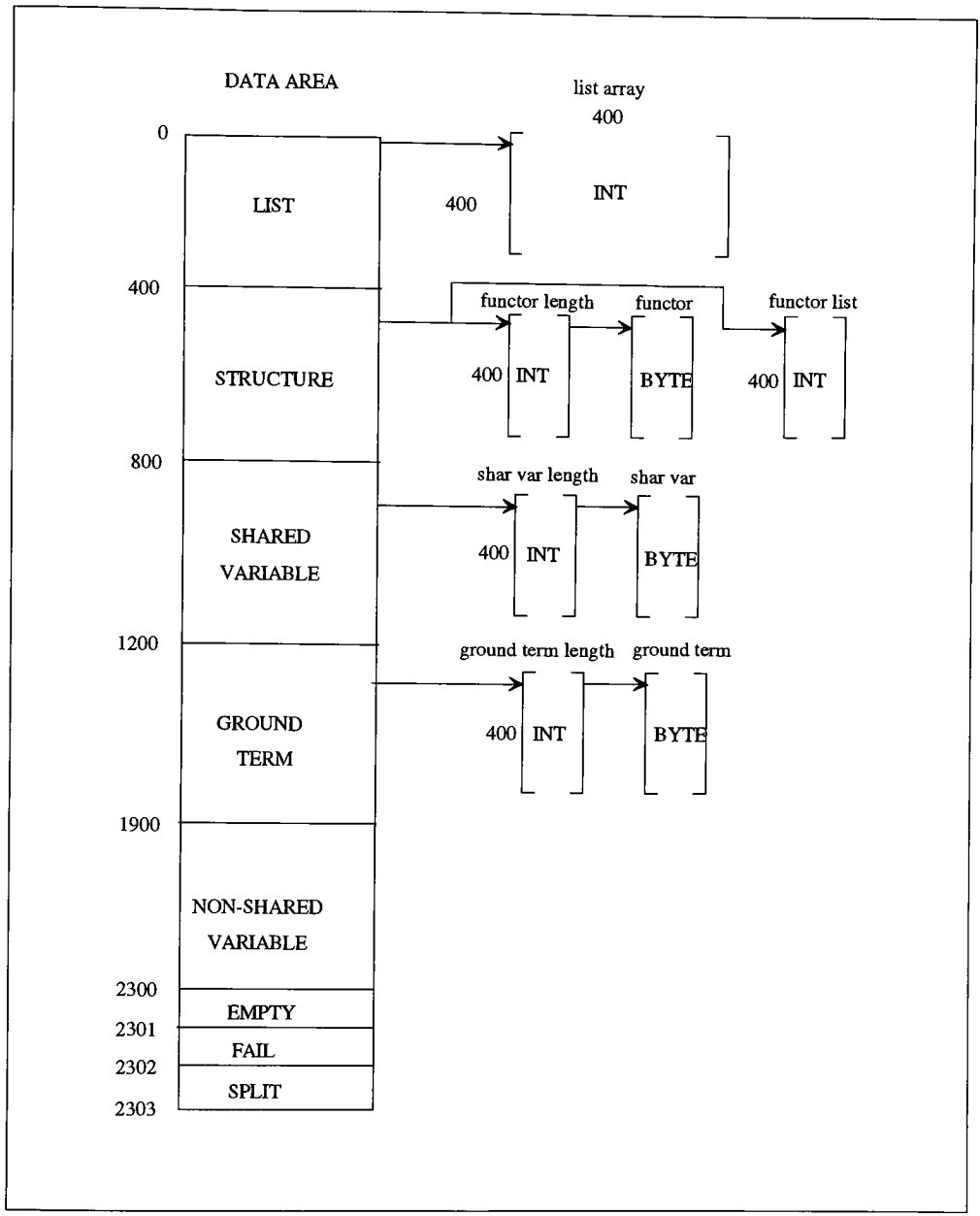


Figure 6.1 - Data Area

4) Ground Terms field A ground term can be an integer, character or string. Ground terms occupy a range from 1201 to 1900. Ground terms are stored exactly like shared

variables. Each ground term has an entry in the integer array called ground term length. This entry is the length of the ground term. The character array ground term stores the ground term. The entry in the ground term length array is obtained by subtracting an offset of 1200 from the integer representing the ground term. The ground term counter keeps track of the number of ground terms stored. It is initialized to 1201.

5) Non Shared Variable field Variables that occur in the body of the clause without occurring in the head of the clause are called non shared variables. Non shared variables are represented only by integers. They occupy a range from 1901 to 2300. The non shared variable counter keeps track of the number of non shared variables present.

6) Empty field This field represents empty data structures, such as empty lists. Any empty data structure is represented by an integer 2301 in this model.

7) Split field - A list can be viewed as consisting of two things - the first item, called the head of the list, and the remaining part of the list, called the tail. The split operation divides a list into the head and tail. Assume we have a list [5,6,7,8]. If we apply the split operation on this list, the head will be set to 5 and the tail will have a value of [6,7,8]. In this model a split will be stored as an integer 2302.

6.2 DATA ALLOCATION

6.2.1 CHECKING INPUT

In order to store the input, it is checked to see if it is in the right format. A procedure called check.term performs the operations of checking the input, which can be any of the data terms described above, except the non shared variables. Non shared variables are generated as the program executes. In the event of an error in the input an error routine is used to inform the user exactly where the error is.

6.2.1.1 PSEUDO CODE FOR CHECK TERM

```
Read the input into an array
IF
    [first element of the array is a square bracket] .... this implies the term is a list
        Call a procedure check.syntax to check the syntax of a list
        IF
            [Syntax is error free]
                Call a procedure assign.list to store the list
            TRUE
                Call a procedure write.error to print out exactly where the error is
    TRUE
        Find the attribute of the term ..... If the term is a variable, ground term or
                                                                    structure
        LOOP
            Check each element of the term to check the syntax
            IF
                [Syntax has an error]
                    Call a procedure write.error to print out exactly where the error is
                [Syntax is error free AND attribute is a ground term]
                    Call a procedure assign.ground.term to store the term
                [Syntax is error free AND attribute is a variable]
                    Call a procedure assign.shar.var to store the variable
                TRUE            .... The term is a structure
                    Call a procedure check.structure to check the syntax of the structure
                    IF
                        [Syntax is error free]
                            Call a procedure assign.structure to store the structure
```

TRUE

Call a procedure write.error to print out exactly where the error is

6.2.1.2 PSEUDO CODE FOR CHECKING SYNTAX OF A LIST

Call a procedure to count the parenthesis ... the number of opening parenthesis should be equal to the number of closing parenthesis

IF

[number of opening parenthesis is equal to number of closing parenthesis]

LOOP

Check each element of each term of the list

IF

[element is a '('] ..... this implies the term is a structure

Call a procedure to count the parenthesis of the structure

TRUE

Do nothing

TRUE

Do nothing

IF

[Syntax is okay till this point]

Check each term of the list. The checks that need to be done are -

- Check if the terms are separated by commas

An opening square parenthesis is followed by a character or another opening square parenthesis

An opening square parenthesis except the first one is preceded by a comma or another opening square parenthesis

- A closing square parenthesis except the last one is followed by a comma or another closing square parenthesis

A closing square parenthesis is preceded by a character, closing round parenthesis or a closing square parenthesis

- If an element has an opening round parenthesis, then its attribute has to be an alphabetic constant. If it is then call a procedure to check syntax of the structure.
- If the attribute of a term is a digit constant, then it can not have an upper case or lower case letter in it
- If a '|' is encountered in the input call a separate procedure to check the syntax of the term after the split. This procedure checks if there is only one element after the split. This element can be a list, character, integer, variable or structure.

TRUE

Call a procedure write.error to print out exactly where the error is

To check the syntax of a structure, the functor and the components of the structure have to be checked. To check the terms of a structure the same logic used to check the terms of a list is used. The functor has to be a string starting with a lower case alphabet.

6.2.2 STORING THE DATA ELEMENTS

6.2.2.1 STORING A LIST

Assume we have the following list -

[1,jack,Head,[[3,4],5],day(1,2,3),date(5,6,7),Tail]

This list will be stored as shown in figure - 6.2.

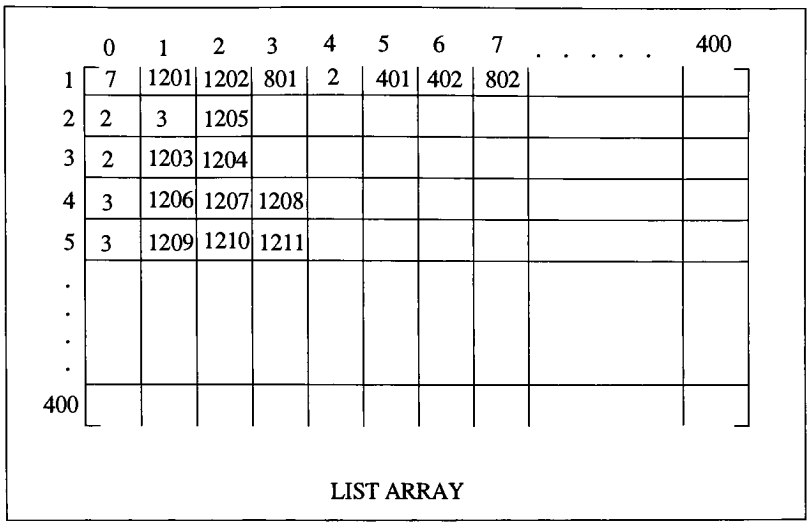


Figure 6.2 - Storing a list in the data area

If the above list is the first term to be stored, then the counters will have to be initialized before the list can be stored. The counters will be initialized to the following values list.ctr = 1, struct.ctr = 401, shar.var.ctr = 801, ground.term.ctr = 1201 and non.shar.var.ctr = 1901. The list is stored in a two dimensional array. The elements of the above list will be stored in the first row of the array, since the value of the list.ctr is equal to 1, which corresponds to the first list being stored. The value of the list.ctr is then incremented to 2. The first element of the row is the number of elements in the list and is 7 for the above example. The individual elements are stored as follows -

1) 1 This is a ground term. It will be stored as 1201 which is the value of the ground.term.ctr. The ground.term.ctr is then incremented to 1202.

2) jack This is a ground term. It will be stored as 1202 which is the value of the ground.term.ctr. The ground.term.ctr is then incremented to 1203.

3) Head - This is a shared variable since it begins with an uppercase letter. It will be stored as 801 which is the value of the shar.var.ctr. The shar.var.ctr is then incremented to 802.

4) [[3,4],5] This is a list. It will be stored as 2 which is the value of the list.ctr. The list.ctr is then incremented to 3. The elements of this list will then be stored in the second row of the list array. This list has 2 elements, so the first element of the second row of the list array will be 2. The elements are [3,4] and 5. The element [3,4] will be stored as 3, which is the value of the list.ctr and the value of the list.ctr will be incremented to 4. The elements of the list [3,4] will be stored in the third row of the list array. The element 3 will be stored as 1203 which is the value of the ground.term.ctr and the ground.term.ctr is incremented to 1204. The element 4 will be stored as 1204 and the ground.term.ctr is incremented to 1205. The element 5 of the list [[3,4],5] will then be stored as 1205 and the ground.term.ctr will be incremented to 1206.

5) day(1,2,3) This is a structure. It will be stored as 401 which is the value of the struct.ctr and the value of the struct.ctr will be incremented to 402. The components of the structure will be stored in row 4 of the list array, which is the value of the list.ctr. The list.ctr will then be incremented to 5. The components of the structure are ground terms and will be stored as 1206, 1207 and 1208 in the fourth row of the list array. The ground.term.ctr is then incremented to 1209.

6) date(5,6,7) - This structure will be stored as 402 which is the value of the struct.ctr and the struct.ctr will be incremented. The components of the structure will be stored in row 5 of the list array, which is the value of the list.ctr. The components of the structure are ground terms and will be stored as 1209, 1210 and 1211 in the fifth row of the list array. The ground.term.ctr is then incremented to 1212.

7) Tail This is a shared variable. It will be stored as 802 which is the value of the struct.ctr and the struct.ctr will be incremented to 803.

6.2.2.2 STORING A STRUCTURE

Assume we have the following structure -

john(1,[[5,7],mary],day(1,X,3))

Each structure has entries in three arrays. The arrays are functor length, functor list and functor. The array functor is a character array which stores each character of the functor and the list where the components of the structure are stored is stored in the array functor list. Let us assume that the above structure is the first term to be stored in the data area. The structure will be stored as shown in figure - 6.3.

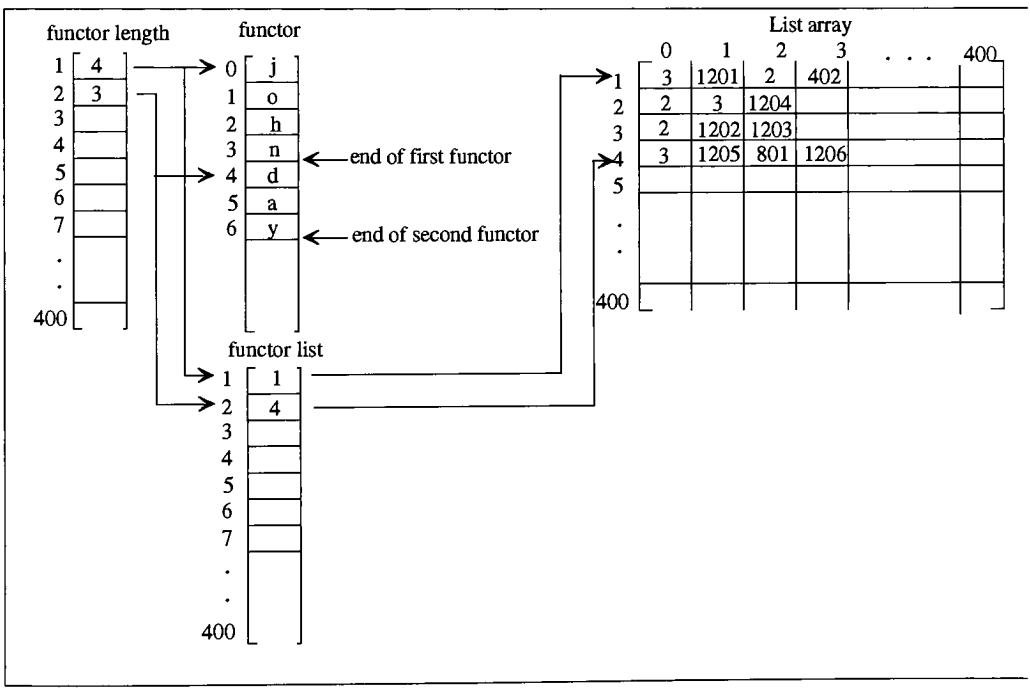


Figure 6.3 - Storing a structure in the data area

This structure is assigned a value of 401. The index into the array functor length for this structure is obtained by subtracting an offset of 400 from 401, which is 1. The first element of the array functor length is 4. Since it is the first functor to be stored it will be stored from the first element in the array functor i.e. functor[0]. It occupies 4 positions in the array. The corresponding element in the array functor list for this structure is 1, which means the components of this structure are stored in row 1 of the list array. The components of this structure are stored as follows -

1) 1 - This is a ground term and is stored as 1201.

2) [[5,7],mary] This is a list and is stored as 2. The components of this list are [5,7] and mary. The procedure for storing these elements has been described in section 6.2.2.1.

3) day(1,X,3) This a structure and is stored as 402. The index into the array functor length for this structure is obtained by subtracting an offset of 400 from 402, which is 2. The second element of the array functor length is 3. The index into the array functor for this structure is obtained by adding all the elements just above the second element in the array functor length. This gives us a result of 4, which means the functor for this structure is stored in the array functor, starting from position 4 i.e. functor[4]. The length of the functor is 3. The corresponding element in the array functor list for this structure is 4, which means the components of this structure are stored in row 4 of the list array. The procedure for storing the components of the structure has been described in section 6.2.2.1.

6.2.2.3 STORING A SHARED VARIABLE

Assume we have the following shared variables -

Mary, Jack.

They will be stored in the data area as shown in figure - 6.4.

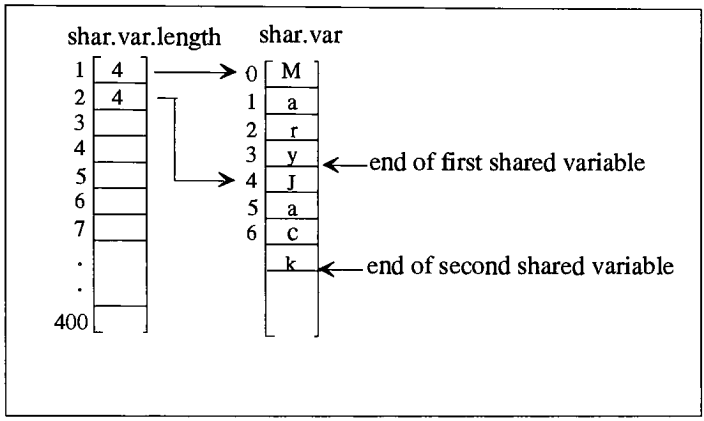


Figure 6.4 - Storing a shared variable in the data area

A shared variable is stored in two arrays. The array shar.var.length stores the index into the array shar.var. Shar.var is a character array which stores each character of the shared variable. Let us assume the variables Mary and Jack are the first two variables to be stored in the data area. They will be stored as shown below -

1) Mary - It will be assigned a value 801. The index into the array shar.var.length for this variable is obtained by subtracting an offset of 800 from 801, which is 1. The first element of the array shar.var.length is 4. Since this is the first variable to be stored in the data area, it will be stored from the first position in the array shar.vars i.e. shar.vars[0]. It will occupy 4 positions in the array.

2) Jack - It will be assigned a value of 802, which is the value of the struct.ctr. The index into the array shar.var.length for this variable is obtained by subtracting an offset of 800 from 802, which is 2. The second element of the array shar.var.length is 4. The length of the shared variable is thus 4. The index into the array shar.var for this variable is obtained by adding all the elements just above the second element in the array shar.var.length. This gives us a result of 4, which means the variable is stored in the array shar.var, starting from position 4 i.e. shar.var[4].

6.2.2.4 STORING A GROUND TERM

A ground term can be an integer, character or string. Let us assume we have the following ground terms -

345, jim, jill.

They are stored in the data area as shown in figure - 6.5.

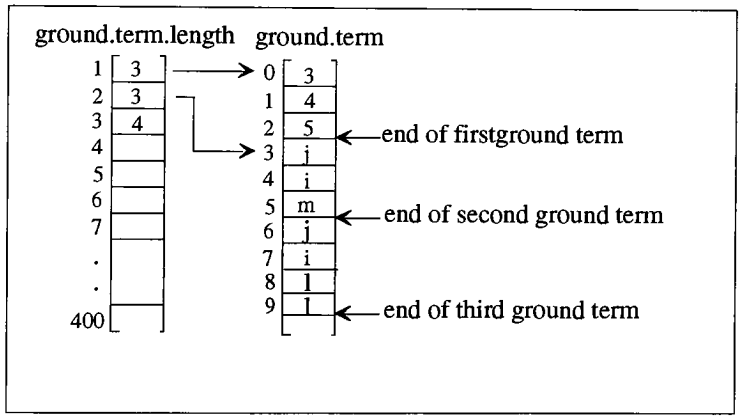


Figure 6.5 - Storing a ground term in the data area

A ground term is stored in two arrays. The array ground.term.length stores the length of each ground term. Ground.term is a character array which stores each character of the ground term. Let us assume the variables 345, jim and jill are the first three

variables to be stored in the data area. They will be stored as follows -

1) 345 - It will be assigned a value 1201. The index into the array ground.term.length for this variable is obtained by subtracting an offset of 1200 from 1201, which is 1. The first element of the array ground.term.length is 3. Since this is the first ground term to be stored in the data area, it will be stored from the first position in the array ground.term i.e. ground.term[0]. It will occupy 3 positions in the array.

2) jim - It will be assigned a value of 1202, which is the value of the ground.term.ctr. The index into the array ground.term.length for this ground term is obtained by subtracting an offset of 1200 from 1202, which is 2. The second element of the array ground.term.length is 3. The length of the ground term is 3. The index into the array ground.term for this variable is obtained by adding all the elements just above the second element in the array ground.term.length. This gives us a result of 3, which means the ground term is stored in the array ground.term, starting from position 3 i.e. ground.term[3].

3) jill To store this ground term, the same procedure used to store the ground term jim is used. It is assigned a value of 1203. The index into the array ground.term.length for this ground term is obtained by subtracting an offset of 1200 from 1203, which is 3. The third element of the array ground.term.length is 4. The length of the ground term is 4. The index into the array ground.term for this variable is obtained by adding all the elements just above the third element in the array ground.term.length. This gives us a result of 3 + 3 = 6, which means the ground term is stored in the array ground.term, starting from position 6 i.e. ground.term[6].

6.2.3 PSEUDO CODE FOR STORING A LIST

6.2.3.1 INTRODUCTION

This procedure uses an array list.track in the form of a stack to keep a track of the the list where elements were stored before the current list. This is useful in storing nested lists. Assume we have the following list - [3,[5,[7,8],9],10] . This will be stored in the data area as shown in figure - 6.6.1.

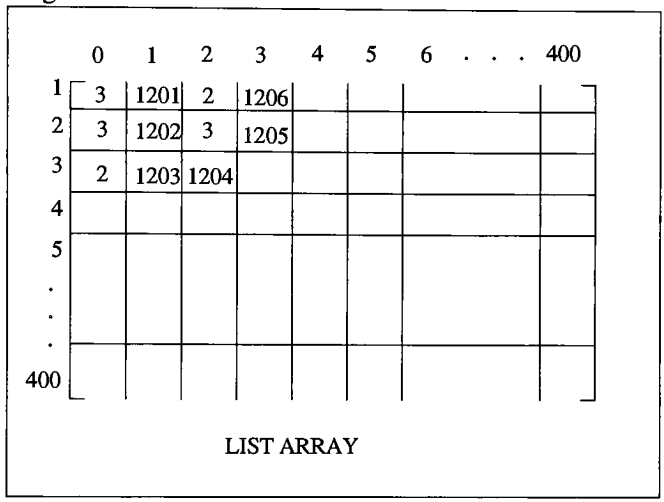


Figure 6.6.1 - Storing the list [3,[5,[7,8],9,10] in the data area

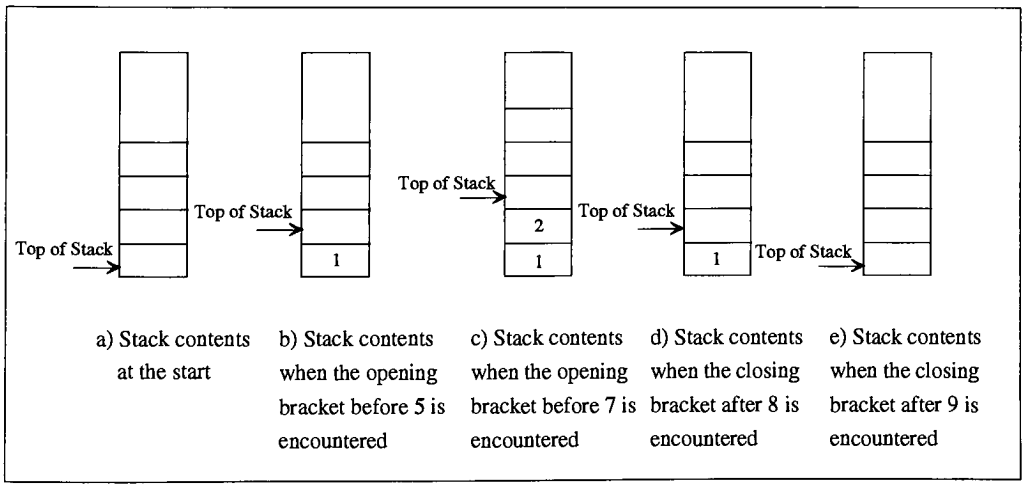


Figure 6.6.2 - Stack contents in different stages of storage of the list

The procedure will start storing the list in row 1 of the array. At this point the array list.track will have nothing stored in it. The stack pointer points to the element 0. When it encounters the opening bracket before 5, it will increment the list counter to 2, push the previous value of the list counter that is 1 on the stack, the list to which it has to return after storing elements in list 2. The pointer will be incremented by 1 to point to the first element. The list elements will now be stored in list 2. When the next opening bracket is encountered the same procedure will repeat. The list counter will be incremented to 3, the current value of the list counter will be pushed on the stack and the stack pointer will be incremented by 1 to point to the second element. The list elements will now be stored in list 3. When the first closing bracket after 8 is encountered, the procedure will pop the top element of the stack. The top element is 2 which corresponds to the list where elements were stored before they were stored in list 3. The current list will now be set to 2 and the elements of the list will now be stored in list 2. The stack pointer will be decremented by 1. When the next closing bracket after 9 is encountered the same procedure repeats. The top element of the stack which is 1 will be popped off, the current list will be set to 1 and the stack pointer will be decremented by 1. The list elements will now be stored in list 1.

6.2.3.2 PSEUDO CODE FOR PROCEDURE ASSIGN.LIST

```
Initialize balance to 1, current list to list counter.
Initialize the stack pointer to 0.
LOOP till balance not equal to 0
  IF
    (element equal to closing square bracket)
      Decrement balance by 1
```

Pop the top value of the stack. Initialize your current list to the

value popped off

Decrement stack counter by 1

(element equal to opening square bracket)

Increment balance by 1

Increment list.ctr by 1 and initialize current list to list.ctr

Push the values of last list on the stack

Increment stack counter by 1

(element equal to space or comma)

Do nothing

TRUE ..... This indicates it is a term of the list

Call a procedure separate character to separate the following term of the list

and to store it

6.2.3.3 PSEUDO CODE FOR PROCEDURE SEPARATE CHARACTER

LOOP till a comma, space, closing square parenthesis, opening round parenthesis or closing round parenthesis is encountered ..... this indicates end of the term

Check the first element of the term to find out the attribute

IF

(first element is a digit)

Set the attribute to digit constant

(first element is a lowercase letter)

Set the attribute to an alphabetic constant

(first element is an uppercase letter)

Set the attribute to variable

IF ..... Check how the loop ended. If it ended due to a comma, space, closing square

..... parenthesis or closing round parenthesis, the term is a variable or ground term.

..... If the loop ended due to an opening round parenthesis the term is a structure

(last element is an opening round parenthesis)

Increase the list.ctr by one ..... The components of the structure have to be stored in the next list

Initialize the current list to the list.ctr

Push the value of the last list on stack

Increment the value of the stack counter by 1

Call a procedure assign.struct to store the functor of the structure

TRUE

IF

(attribute of the term is a digit constant or alphabetic constant)

Call a procedure assign.ground.term to store the ground term

(attribute of the term is a variable)

Call a procedure assign.shar.var to store the variable

TRUE

Do Nothing

IF

(last element is a closing square parenthesis)

Decrement balance by one

Pop the top value from the list stack

Initialize the current list to the value popped

Decrement the stack counter by 1

(last element is a closing round parenthesis)

Pop the top value from the list stack

Initialize the current list to the value popped

Decrement the stack counter by 1

TRUE

Do Nothing

6.2.3.4 PSEUDO CODE FOR PROCEDURE ASSIGN.STRUCT

This procedure takes two arguments - the character array in which the functor is stored and the length of the functor. The code for this procedure is as follows -

Enter the length of the functor in array functor.length

Enter the list number where the components of the structure are stored in array func.list

Initialize index to 0

IF

[functor is not the first functor to be stored]

Calculate index into the array functor where the functor should be stored

TRUE

Do nothing

Store the functor in the array functor from the position given by index

Increment the structure counter by 1

6.2.3.5 PSEUDO CODE FOR PROCEDURE ASSIGN.GROUND.TERM

This procedure takes two arguments the character array in which the ground term is stored and the length of the ground term. The code for this procedure is as follows

Enter the length of the ground.term in array ground.term.length

Initialize index to 0

IF

[ground term is not the first ground term to be stored]

Calculate index into the array ground term where the ground term should be stored

TRUE

Do nothing

Store the ground term in the array ground term from the position given by index

Increment the ground term counter by 1

6.2.3.6 PSEUDO CODE FOR PROCEDURE ASSIGN.SHAR.VAR

This procedure takes two arguments the character array in which the shared variable is stored and the length of the shared variable. The code for this procedure is as follows -

Enter the length of the shared variable in array shar.var.length

Initialize index to 0

IF

[shared variable is not the first shared variable to be stored]

Calculate index into the array shar. var where the shared variable should be stored

TRUE

Do nothing

Store the shared variable in the array shar var from the position given by index

Increment the shared variable counter by 1

6.3 VIRTUAL MACHINE INSTRUCTIONS

The execution model mainly consists of five virtual machine instructions. The virtual machine instructions are coded as individual procedures, which are independent from each other. All Prolog procedures are compiled into the virtual machine instructions.

6.3.1 UNIFY

This instruction unifies two input operands and outputs a binding environment. It stores the bindings in two integer arrays called variable and environment. The array variable stores the variable identifier and the array environment stores the binding.

6.3.1.1 PSEUDO CODE FOR INSTRUCTION UNIFY

```
IF
    [term1 is a list AND term2 is a list]
        Call procedure unify.list to unify each term of the list
    [term1 is a structure AND term2 is a structure]
        Call procedure check.functor to check if both the structures have the same
        functor
        IF
            [both the structures have the same functor]
                Call procedure unify.list to unify the components of the structure
            TRUE
                Do nothing
    TRUE
        Call procedure unify.elemt to unify the terms
```

6.3.1.2 PSEUDO CODE FOR INSTRUCTION UNIFY.LIST

A list consists of a number of terms. The terms can be lists, structures, ground terms or variables. If the list contains nested lists or the structures contain lists as their components then the elements of the list have to be unified recursively. Since Occam does not support recursion this procedure uses a queue to unify the elements. The queue is a FIFO queue implemented as a one dimensional array of integers. If any of the elements of the list is a list, the corresponding list from both the terms are put on two queues, to be

unified later on. After the procedure has finished unifying the elements in both the lists, it unifies those elements in the queue. This procedure can be explained with the help of an example. Assume we have the following two terms or lists to be unified -

[[4,[5,6]],day(1,2,3),mary,3]

[[4,[5,6]],day(1,2,3),X,3]

They will be stored in the list array as shown in figure - 6.7.

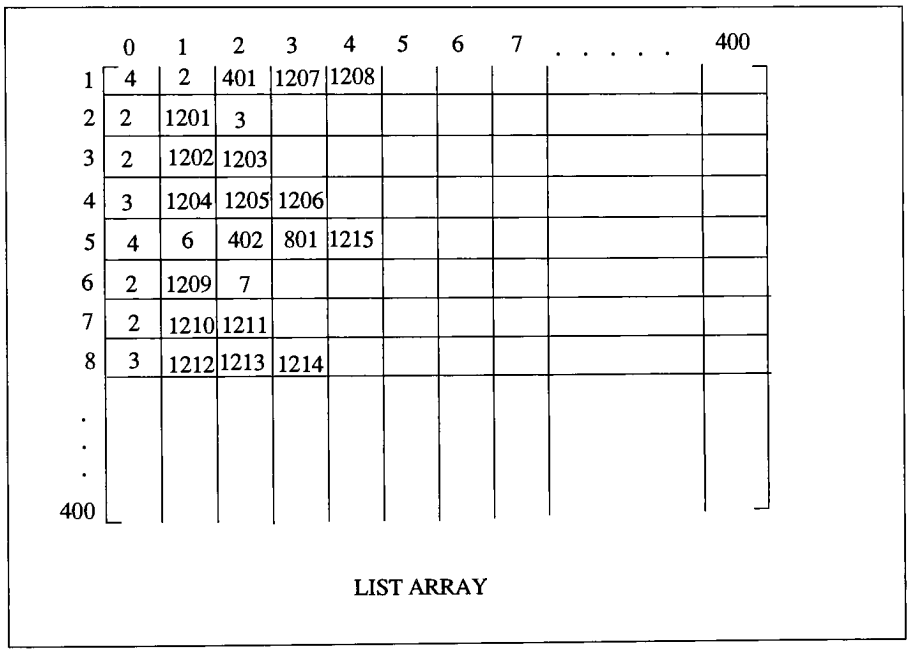


Figure 6.7. - Storing the lists to be unified in the data area

The first list will be assigned a value 1 and the second list will be assigned a value 5. The procedure unify.list will unify the four terms of the lists as follows -

1) The first component of both the lists is a list. It is not unified but will be put on queue1 and queue2 to be unified later on. Queue1 will contain the elements of term1 that are to be unified later on and Queue2 will contain the corresponding elements of term2. Thus

queue1 and queue2 will contain the elements shown in figure - 6.8(a).

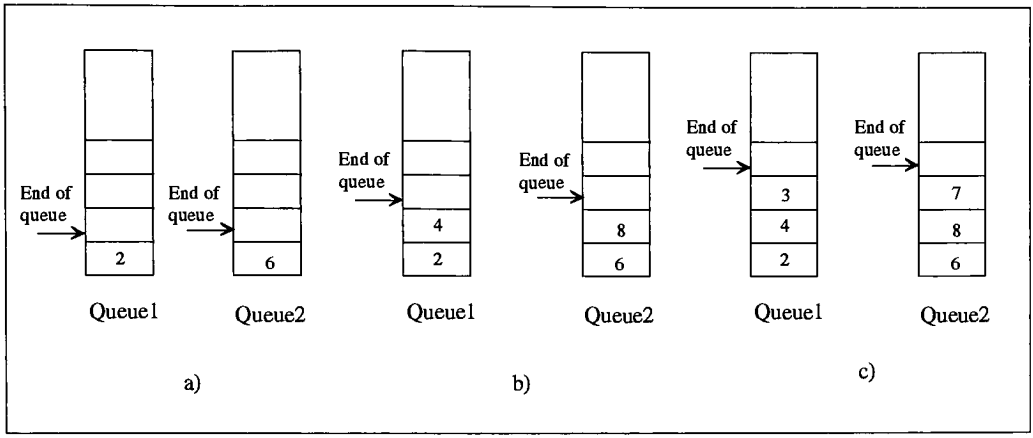


Figure 6.8 - Queue contents during unification

2) The second component of both the lists is a structure. To unify the structures the procedure calls another procedure to check if the two structures have the same functor. Since both the structures have the same functor it proceeds to unify the components of the structures. The components of the two structures are stored in lists 4 and 8. The procedure does not unify the components of the structure but puts the lists that contain the components on queue1 and queue2. Thus queue1 and queue2 will now contain the elements shown in figure - 6.8(b).

3) The third component of term1 is a ground term and the corresponding component of term2 is a shared variable. These two terms are unified directly and the binding is stored in the environment.

4) The fourth component of both the lists are ground terms and thus can be unified directly.

After this is done the procedure will check if there are elements in the queues to be unified. Since the queues contain elements to be unified another procedure queue.check is called to unify the elements in the queue. The procedure will start by unifying elements 2 and 6 which are in the first position in queue1 and queue2 respectively. List 2 and list 6 each has two terms. The first term of each list is a ground term and can be unified directly. The second term of each list is a list. The lists are not unified at this point but are put in queue1 and queue2 to be unified later. The two queues now contain the elements shown in figure - 6.8(c).

The elements 4 and 8 which are the elements in the second position in queue1 and queue2 will be unified now. List 4 and list 8 each has 3 terms. Each of the terms can be unified directly since they are ground terms. The last elements on queue1 and queue2 to be unified are elements 3 and 7. List 3 and list 7 each has 2 ground terms which can be unified directly. Since there are no more elements on the stack to be unified, the unification of term1 with term2 is complete. The pseudo code for the procedure is as follows -

Call procedure check.list.length to check if the lists to be unified have the same length, if they do not it checks if the lists can be unified and it returns the length of the lists to be unified

LOOP till unification is successful AND till all the terms of the list have not been unified

IF

[element of list1 is a list AND element of list2 is a list]

Put the element of list1 in queue1

Put the element of list2 in queue2

Increment the queue pointer by one

[element of list1 is a structure AND element of list2 is a structure]

Call a procedure check.functor to check if both the lists have the same functor

IF

[both the lists have the same functor]

Put the list that contains the components of structure1 in queue1

Put the list that contains the components of structure2 in queue2

Increment the queue pointer by one

TRUE

Do nothing

TRUE

Call a procedure unify.elemt to unify the elements

IF

[queue.ptr is not equal to 0 AND unification is successful till this point]

Call a procedure queue.check to unify the elements in queue1 and queue2

TRUE

Do nothing

6.3.1.3 PSEUDO CODE FOR PROCEDURE QUEUE.CHECK

LOOP till there are no more elements in the queue to be unified AND till unification is successful

IF

[current element of queue1 is a list AND current element of queue2 is a list]

Call procedure check.list.length to check if the lists to be unified have the same length, if they do not it checks if the lists can be unified and it returns the lengths of the lists to be unified

LOOP till unification is successful AND till all the terms of the list have

not been unified

IF

[element of list1 is a list AND element of list2 is a list]

Put the element of list1 in queue1

Put the element of list2 in queue2

Increment the queue pointer by one

[element of list1 is a structure AND element of list2 is a structure]

Call a procedure check.functor to check if both the lists have the same functor

IF

[both the lists have the same functor]

Put the list that contains the components of structure1 in queue1

Put the list that contains the components of structure2 in queue2

Increment the queue pointer by one

TRUE

Do nothing

TRUE

Call a procedure unify.elemt to unify the elements

TRUE

Call a procedure unify.elemt to unify the elements directly

6.3.1.4 PSEUDO CODE FOR CHECK.FUNCTOR

This function takes the structure identifiers as its arguments. It checks if the two structures have the same functors. The pseudo code is as follows -

Initialize count1 i.e. the index into the array functor for structure1 and count2 i.e. the

index into the array functor for structure 2 to 0

IF

[structure1 is not the first structure to be stored]

Calculate the index into the array functor by summing all the elements in the array functor.length that are above the structure.

Initialise count1 to this index

TRUE

Leave count1 as 0 ..... the index into the array functor is 0

IF

[structure2 is not the first structure to be stored]

Calculate the index into the array functor by summing all the elements in the arrray functor.length that are above the structure.

Initialise count2 to this index

TRUE

Leave count2 as 0 ..... the index into the array functor is 0

IF

[Both the functors have the same length]

Check the elements of the array functor from the index count1 with the elements from the index count2

TRUE

Do nothing

6.3.1.5 PSEUDO CODE FOR UNIFY.ELEMT

This procedure unifies those terms that can be unified directly and stores the appropriate binding messages. The arrays variable and environ are used to store the binding messages. The array variable stores the variable identifier and the array environ stores the corresponding binding. The pseudo code for the procedure is as follows -

[term1 is a non shared variable AND term2 is a non shared variable]

Put the binding of term1 to term2 and term2 to term1 in the arrays variable and environ

[term1 is a non shared variable AND term2 is not a non shared variable]

Put the binding of term1 to term2 in the arrays variable and environ i.e. put term1 in array variable and term2 in the corresponding position in array environ

[term2 is a non shared variable AND term1 is not a non shared variable]

Put the binding of term2 to term1 in the arrays variable and environ i.e. put term2 in array variable and term1 in the corresponding position in array environ

[term1 is a shared variable]

IF

[term2 is not a shared variable]

Check the binding environment to see if term1 is bound

IF

[term1 is bound]

Unify the binding of term1 with term2

TRUE

Put the binding of term1 to term2 in the arrays variable and environ i.e. put term1 in the array variable and term2 in the corresponding position in the array environ

TRUE

Check the binding environment to see if term1 is bound

IF

[term1 is bound]

Check the binding environment to see if term2 is bound

IF

[term2 is bound] ..... both term1 and term2 are bound

Unify the binding of term1 with the binding of term2

TRUE

Put the binding term2 to the binding of term1 in the arrays variable and environ i.e. put term2 in the array variable and the binding of term1 in the array environ

TRUE

Check the binding environment to see if term2 is bound

IF

[term2 is bound]

Put the binding term1 to the binding of term2 in the arrays variable and environ i.e. put term1 in the array variable and the binding of term2 in the array environ

TRUE

Put the binding term1 to term2 in the arrays variable and environ

[term2 is a shared variable]

Repeat the same procedure followed for term1

[term1 is a ground term AND term2 is a ground term]

Initialise count1 i.e. the index into the array ground.term for term1 and count2 i.e. the index into the array ground.term for term2 to 0

IF

[term1 is not the first ground.term to be stored]

Calculate the index into the array ground.term by summing all the elements in the array ground.term.length that are above term1.

Initialise count1 to this index

TRUE

Leave count1 as 0 ..... the index into the array ground.term is 0

IF

[term2 is not the first ground.term to be stored]

Calculate the index into the array ground.term by summing all the elements in the array ground.term.length that are above term2.

Initialise count2 to this index

TRUE

Leave count2 as 0 ..... the index into the array ground.term is 0

IF

[Both the ground.terms have the same length]

Check the elements of the array ground.term from the index count1 with the elements from the index count2

TRUE

Do nothing

TRUE

Unification fails

6.3.2 DECOMPOSE

This instruction decomposes the input term into a list head and tail list. It takes the term identifier as its argument and produces a head and modifies the term identifier to produce another list. Assume we have the following list -

[4,3,mary,[5,7]]

After the list is decomposed the list array will contain the elements shown in figure - 6.9.

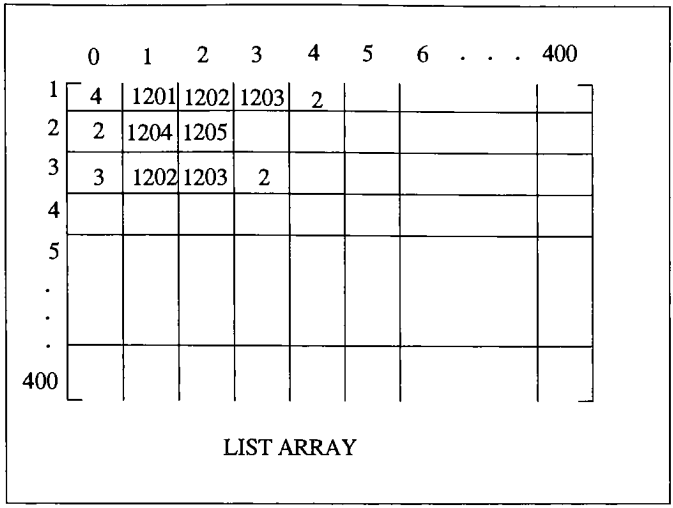


Figure 6.9 - Storing the lists [4,3,mary,[5,7]] and [3,mary,[5,7]] in the data area

The list has an identifier 1 i.e. it is stored in the first row of the list array. The procedure decompose will perform the following operations -

1) The head is initialized to the first element of the list, that is 1201.

2) The remaining three elements of the list will be stored in the next available row in the list array, that is row 3. The term identifier is then changed to three. If the input is a shared or non shared variable, the procedure creates two non shared variables as the list head and tail and stores the list [head | tail] in the first row of the list array. It also creates a binding message with the input variable and the binding [head | tail] and stores the binding in the stored environment.

6.3.2.1 PSEUDO CODE FOR DECOMPOSE

```
IF
    [term is a list]
        assign the first element of the list to the head
        IF
            [list contains only one element]
                assign empty to the tail
            TRUE
                assign the remaining elements of the list to the tail
    [term is a shared variable OR term is a non shared variable]
        assign the value of the non shared variable counter to the head
        Increment the non shared variable counter
        assign the value of the non shared variable to the tail
        Store the list [head | tail] in the first available list
        Store the binding of the input to the list [head | tail] in the binding environment
    TRUE ..... if the input is anything else besides a list or variable
        decompose fails
```

6.3.3 CHECK

This instruction checks each binding message (X B) of the stored binding environment to see if in the binding instance B there are one or more shared variables which have their own binding instances in other binding messages in the temporary binding environment. A check instruction is always executed after a unify instruction. A unify instruction stores the bindings produced by unification in a temporary binding environment and a check instruction transfers the contents of a temporary binding environment to a stored binding environment if the unification is successful. If so the instruction then replaces the shared variables by these binding instances. Let us assume we

have the following binding environments -

stored binding environment X = [3, relation(Z,[5,W]),[4,W]]

temporary binding environment Z = 7

W = mary

They are stored as shown in figure - 6.10.

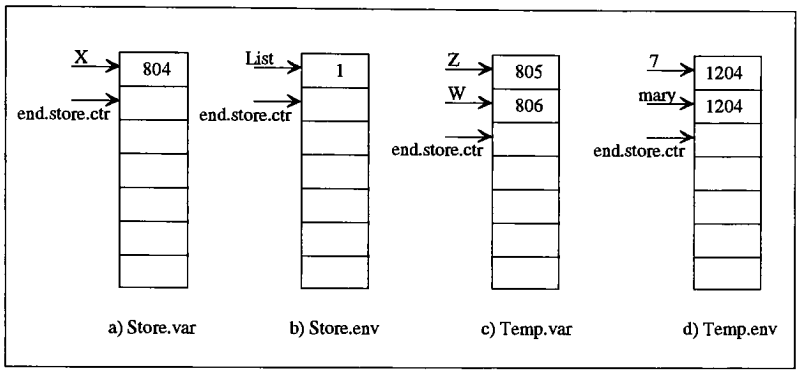


Figure 6.10 - Stored binding environment and temporary binding environment

The lists are stored in the list array as shown in figure - 6.11.

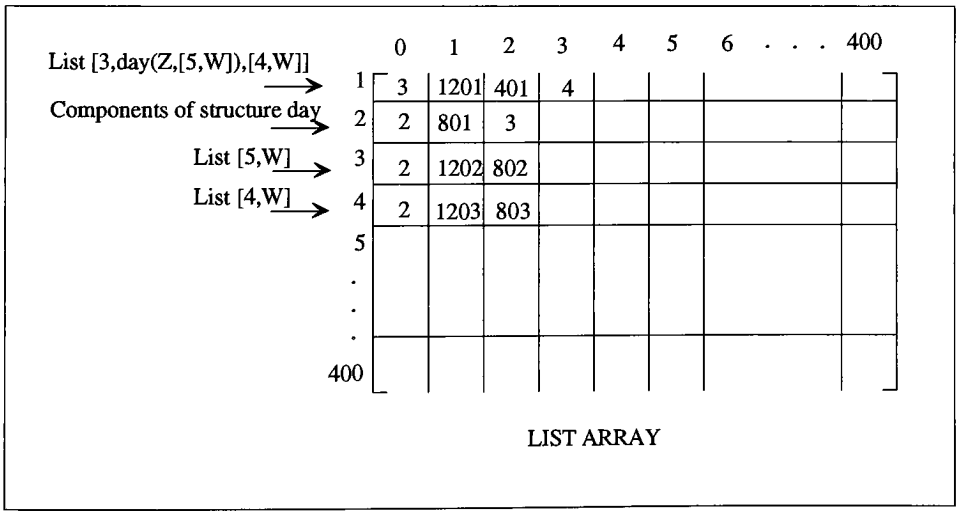


Figure 6.11 - Storing the lists of the binding environments in the data area

The stored binding environment contains only one variable X. The value of X is a list. When the instruction check is executed on this stored binding environment with the temporary binding environment shown in figure - 6.10, the following events take place -

1) The first component of the list 1 is checked. Since it is a ground term it is ignored.

2) The second component of the list is the structure day. The components of the structure are stored in list 2. Thus the components of list 2 have to be checked to see if it contains any variables. The list 2 is not checked at this point, but is put in a queue to be checked later on. The queue now contains the elements shown in figure - 6.12(a).

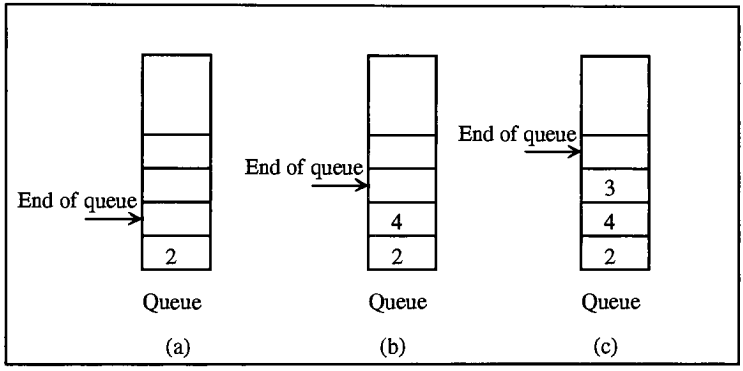


Figure 6.12 - Contents of queue during execution of instruction check

3) The third component of the list is the list [4,W] stored as list 4. This list has to also be checked to see if it contains any variables. The components of this list are not checked at this point, but it is put in the queue to be checked later on. The queue now contains the elements shown in figure - 6.12(b).

After all the components of the list are checked, the components in the queue have to be checked. The first element in the queue is list 2. The components of list 2 are

checked. The first component of list 2 is the variable Z, stored as 801. The temporary binding environment will be checked to see if it contains the variable Z. The temporary binding environment contains the variable Z which is bound to the ground term 7, which is stored as 1204. Thus the first component of the list 2 which is 801 will be replaced by 1204. The second component of the list 2 is the list [5,W] which is stored as list 3. The components of this list will not be checked at this point, but it will be put in the queue to be checked later on. The queue will now contain the elements show in figure - 6.12(c)

This completes the checking of the components of list 2. The second element of the queue is list 4. The components of this list will be checked in a similar manner. The variable W stored as 803, will be replaced by 1205 which is the binding of W to mary, stored in the temporary binding environment. The last component in the queue to be checked is list 3. After the execution of the instruction check the value of X will be [3, day(7, [5, mary]), [4, mary]]. The list array will now contain the elements shown in figure-6.13.

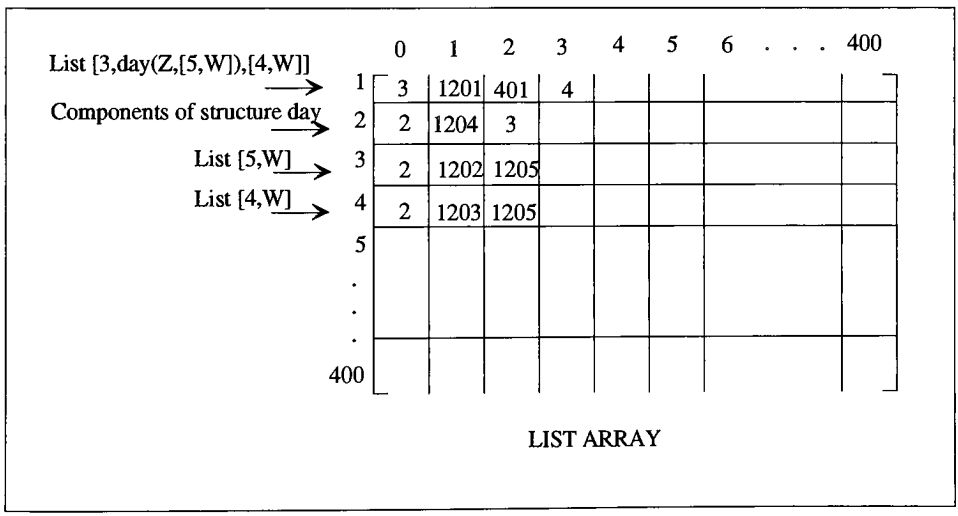


Figure 6.13 - Contents of list array after execution of instruction check

6.3.3.1 PSEUDO CODE FOR INSTRUCTION CHECK

This instruction calls another instruction queue.check to check the elements in the queue. The code for this instruction is as follows -

```
LOOP till all the elements of the stored binding environment are checked
    Initialize the queue pointer to 0
    LOOP till all the elements of the temporary binding environments are checked with
    the current element of the stored binding environment
        IF
            [the binding of the current element of the stored binding environment
             is a list or structure]
                LOOP till all the elements of the list are checked
                    IF
                        [element is a list]
                            Put the list number in the queue
                            Increment the queue pointer
                        [element is a structure]
                            Put the list number that contains the components
                            of the structure in the queue
                            Increment the queue pointer
                        [element is a shared variable]
                            IF
                                [variable is the same as the current variable of
                                the temporary binding environment]
                                    Replace the variable by its binding from
                                    the temporary binding environment
                                TRUE
                                    Do Nothing
```

TRUE

Do Nothing

[the binding of the current element of the stored binding environment is a variable]

IF

[variable is the same as the current variable of the temporary binding environment]

Replace the variable by its binding instance in the temporary binding environment

TRUE

Do Nothing

TRUE

Do Nothing

IF

[queue.ptr is not equal to 0]

Call a procedure queue.check to check the elements in the queue

TRUE

Do Nothing

6.3.4 INSTANTIATE

This instruction receives a term and the stored binding environment. If the input term is a ground term the instruction does nothing but output the same term, otherwise, the instruction instantiates the variables occurring in the term according to their binding instances in the stored binding environment and then outputs the instantiated term. We will consider the procedure to instantiate lists and variables under two different sections.

6.3.4.1 INSTANTIATING A LIST

Let us assume we have the following list to instantiate -

[[4,X],day(1,X,Z),Z]

The list will be stored in the data area as shown in figure - 6.14.

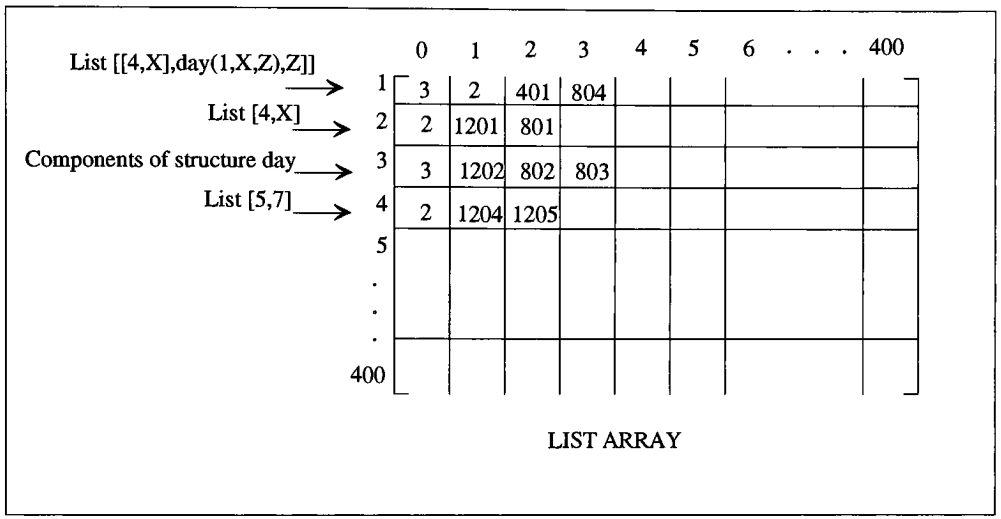


Figure 6.14 - Storing the list [[4,X],day(1,X,Z),Z] in the data area

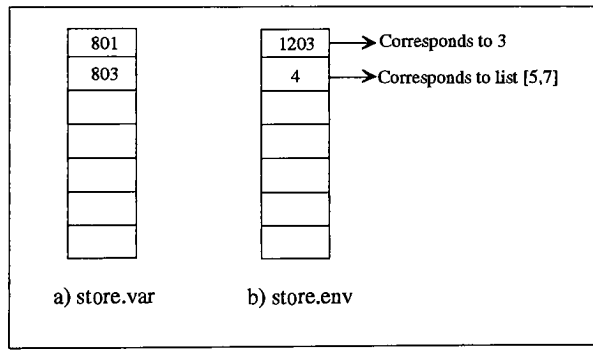


Figure 6.15 - Stored binding environment

This procedure uses two queues - list.queue and print.queue. The list.queue is used to take care of recursion i.e. it is used to instantiate nested lists. The queue print.queue

keeps track of the variables whose values have been printed out, to avoid the same results from being printed out twice. The components of the list will be instantiated as follows -

1) The first term of the list is a list. It will be put in list.queue to be instantiated later on. The queues will now contain the elements shown in figure - 6.16(a).

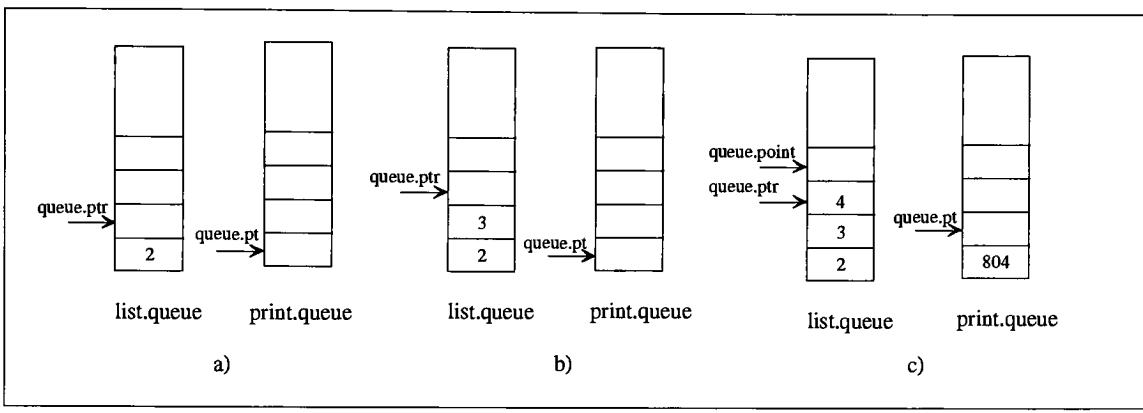


Figure 6.16 - Contents of queues during execution of instruction instantiate

2) The second term of the list is a structure. The components of the structure are stored in list 3. The components of the structure will not be instantiated at this point, but will be put in list queue to be instantiated later on. The queues will now contain the elements shown in figure - 6.16(b).

3) The third component of the list is a variable. At this point another variable queue.point will be initialized to queue.ptr. The stored environment will be checked to see if the variable has been instantiated to a value. Since the variable does have a value in the stored binding environment, the print queue will be checked to see if the variable is present in the queue. Since the variable is not present in print.queue, it has not been printed out. The variable is put in print.queue. Since the value of the variable is a list, it is not instantiated

at this point, but it is put in list.queue and the value of queue.point is incremented by one. The queues will now contain the elements shown in figure - 6.16(c) After this the value of queue.ptr is compared with that of queue.point. Since they are not equal, there is an element in list.queue that has to be instantiated. After this element has been instantiated the value of the variable is printed out.

After all the elements of the list have been instantiated, the list queue is checked to see if there are any elements to be instantiated. Since there are 2 elements in list.queue to be instantiated, they are instantiated next. The first element to be instantiated will be list 2. This contains 2 elements. The first element is a ground term so it will be left alone, the second term has to be instantiated since it is a variable. The stored binding environment will be checked to see if the variable has a value. Since the variable does have a value, the print queue will be checked to see if the variable has been printed out. Since it has not been printed out, it will be put in print.queue and its value will be printed out from the stored binding environment. The next element in the list queue to be instantiated is list 3. The first component of this list is a ground term so it will be left alone. The next two elements are variables, but since they are present on the print queue, it implies that they have been printed out, and hence will not be instantiated again.

6.3.4.2 PSEUDO CODE FOR INSTANTIATING A LIST

```
LOOP for the number of elements present in the list
  IF
      [element is a list]
          Put the element in list.queue
          Increment queue.ptr
      [element is a structure]
```

Put the list that contains the components of the structure in list.queue

Increment queue.ptr

[element is a variable]

Initialise queue.point to queue.ptr

LOOP for the number of elements present in store.var

Call a procedure check.var to see if the current element of store.var and the variable are the same

IF

[current element of store.var and variable are same]

Call a procedure check.print.queue to see if the variable has been printed before

IF

[variable has not been printed before]

Put the variable in print.queue

IF

[current element of store.envir is a variable]

Find the value of this variable from the stored binding environment

[current element of store.envir is a list]

Put the list in list.queue

Increment queue.point by 1

[current element of store.envir is a structure]

Put the list that contains the components of the structure in list.queue

Increment queue.point by 1

TRUE

Call a procedure to print out the value of the variable

TRUE

SKIP

IF

[queue.ptr is not equal to queue.point]

Call a procedure list.queue.check to instantiate the elements in the queue

Call a procedure to print out the value of the variable

TRUE

SKIP

TRUE

SKIP

IF

[queue.ptr is not equal to zero]

Call a procedure list.queue.check to instantiate the elements in the queue

TRUE

SKIP

The procedure list.queue.check instantiates all the elements in the queues. This procedure will use the procedure to instantiate lists, to instantiate all the elements in the queue, which are lists.

6.3.4.3 INSTANTIATING A VARIABLE

To instantiate a variable the stored binding environment is checked to see if the variable has a value. If the variable does not have an entry in the stored binding

environment, it implies that the variable has not been instantiated. If the variable has an entry in the stored binding environment, then its value from the binding environment is instantiated. If its value is a list or a structure, the procedure to instantiate a list explained in section 6.3.4.1 is used. If its value is another variable, then the value of that variable is found from the stored binding environment. After the value of the variable has been instantiated, it is printed out.

6.3.4.4 PSEUDO CODE FOR INSTANTIATING A VARIABLE

```
Find where the variable is located in the stored binding
IF
    [variable does not have an entry in the stored binding environment]
      Print the value of the variable as uninstantiated
    TRUE
      IF
          [variable has a value equal to a list or a structure]
             Use the procedure to instantiate a list
          [variable has a value equal to another variable]
             Find the value of that variable from the stored binding environment
          TRUE
             SKIP
      Print out the instantiated value of the variable
```

6.3.5 CONSTRUCT

This instruction takes two inputs : a list head and a tail list, to construct a list. It stores the newly formed list in the first available row of the list array. Let us assume we want to construct a list from the following elements -

Head [6,8]

Tail [4,3,mary,[5,7]]

The above elements will be stored in the list array as shown in figure - 6.17.

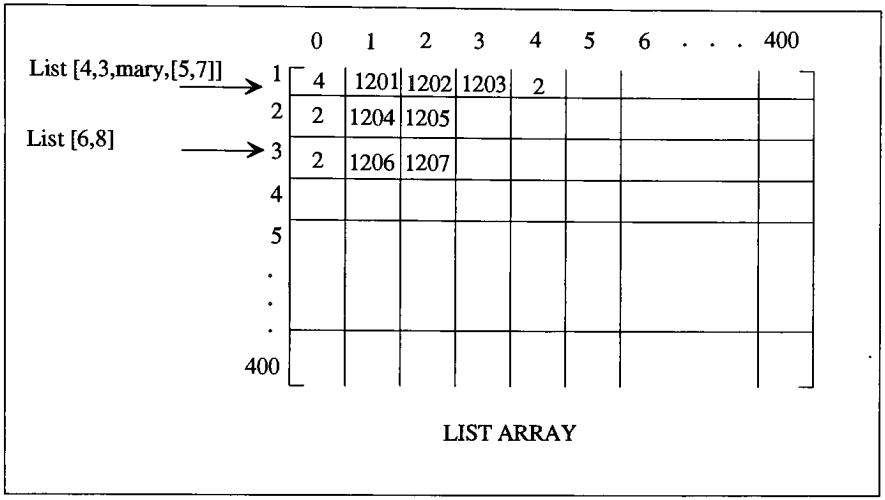


Figure 6.17 - Storing the list [4,3,mary,[5,7]] and [6,8] in the data area

The Tail is stored as the first list and the head is stored as the third list in the data area. The newly constructed list will be stored as the fourth list in the data area. This list will contain 5 elements, which are the head and the elements of the tail list. The newly constructed list will be stored as shown in figure - 6.18.

6.3.5.1 PSEUDO CODE FOR INSTRUCTION CONSTRUCT

IF

[tail is empty]

Number of elements in the newly constructed list is 1

Assign the head to the first element of the new list

TRUE

Number of elements of the newly constructed list is 1 + number of elements of

the Tail list

Assign the head to the first element of the new list

Assign the elements of the Tail list to the remaining elements of the new list

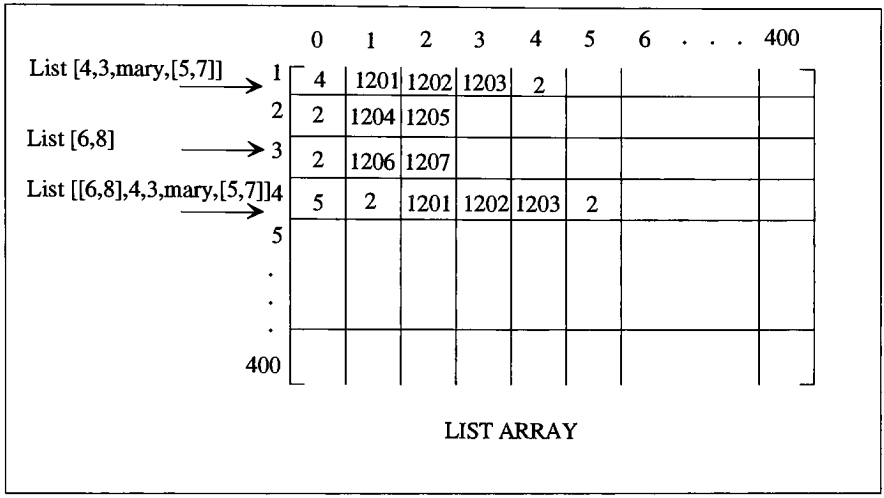


Figure 6.18 - Storing the newly constructed list [[6,8],4,3,mary,[5,7]] in the data area

6.4 USING THE VIRTUAL MACHINE INSTRUCTIONS TO CONVERT A PROLOG PROCEDURE TO OCCAM

Let us consider the Prolog definition of the procedure delete, which deletes an element from a list and outputs the new list.

```
del(L1,[L1|L2],L2).
del(L1,[Head|L2],[Head|L3]) :- del(L1,L2,L3).
```

This procedure can be hand compiled into an Occam2 representation of the virtual machine instructions and run on a single transputer. The following is the compiled code of the procedure -

```
PROC delete(INT list1, list2, list3)
  ... declarations
```

```
SEQ
    . . . initializations
    WHILE run
        SEQ
            decompose(store.env, store.var, L2, head2)
            IF
                (run)
                    SEQ
                        unify(L1, head2, succ, var1, envir1)
                        IF
                            (succ) AND (run)
                                SEQ
                                    check(store.env, store.var, var1, envir1)
                                    unify(L2,L3,succ,var2,envir2)
                                    check(store.env, store.var, var2, envir2)
                                    instantiate(store.env, store.var, list1)
                                    instantiate(store.env, store.var, list2)
                                    instantiate(store.env, store.var, list3)
                                    run := FALSE
                            TRUE
                                SEQ
                                    decompose(store.env, store.var, L3, head3)
                                    unify(head2, head3, var1, envir1)
                                    check(store.env, store.var, var1, envir1)
                TRUE
                    find := FALSE
```

The local variables are declared in the "declarations" fold. The "initializations" fold contains the assignments of initial values to some variables, such as TRUE to "run" and "find", "listi" to "Li" (i=1,2,3). The procedure first decomposes list2. If list2 can not be deomposed any further it implies that list2 does not contain the element list. The procedure then sets the variable "find" to FALSE. If list2 can be decomposed the procedure unifies the head of list2 with list1. The success of unification implies that list2 contains the element list1. The element list1 is deleted from list2 to from list3. If the unification fails, list3 is decomposed and the procedure is repeated with list1, tail of list2 and tail of list3. The procedure ends when the variable "run" becomes FALSE.

6.5 EXECUTION OF RECURSIVE FUNCTIONS THAT CALL OTHER RECURSIVE FUNCTIONS

Let us consider the function permutation. It is programmed in Prolog as follows -

```
permutation([],[]).
permutation(L1,[L2|Tail]) :-  del(L2,L1,Term1), permutation(Term1,Tail).
del(L1,[L1|L2],L2).
del(L1,[Head|L2],[Head|L3]) :-  del(L1,L2,L3).
```

In the above definition of the function permutation, a call to the function delete can result in a number of solutions. The problem associated with such a recursive function is the selection of one particular solution returned by delete to carry on the recursion and the storing of the other solutions to work with later on. This can be understood better with the help of an example. Let us assume we have to find the permutations of the list [1,2,3]. The permutations of the list are [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2] and [3,2,1]. The execution of the function permutation on the list [1,2,3] is shown in figure - 6.19.

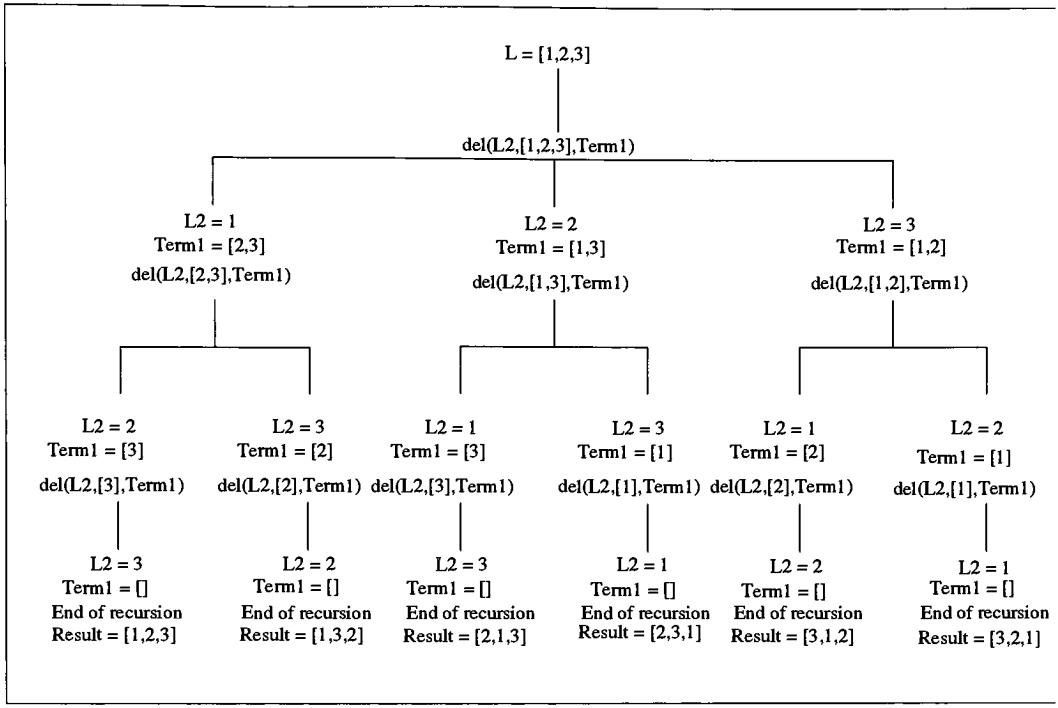


Figure 6.19 - Steps in finding the permutations of the list [1,2,3]

It can be seen from figure 6.19, that the structure of the solution is a tree. A particular permutation or solution to the problem is obtained by traversing the tree from top to bottom till a node that does not have any branches is encountered. The first permutation obtained is [1,2,3]. To get the next solution the tree is traversed from that node in an upward direction till a node is obtained that has branches which have not been traversed. The tree is then traversed from that point. In figure 6.19, it can be seen that after the first permutation [1,2,3] is obtained the tree will be traversed upward till the node with L2 = 1 and Term1 = [2,3] is encountered. This node has a branch that has not been traversed. The next solution will be obtained by going down that branch. The paths traversed to get the second and third solutions are shown in figures 6.20 and 6.21 with dotted lines. The other solutions can be obtained in a similar manner.

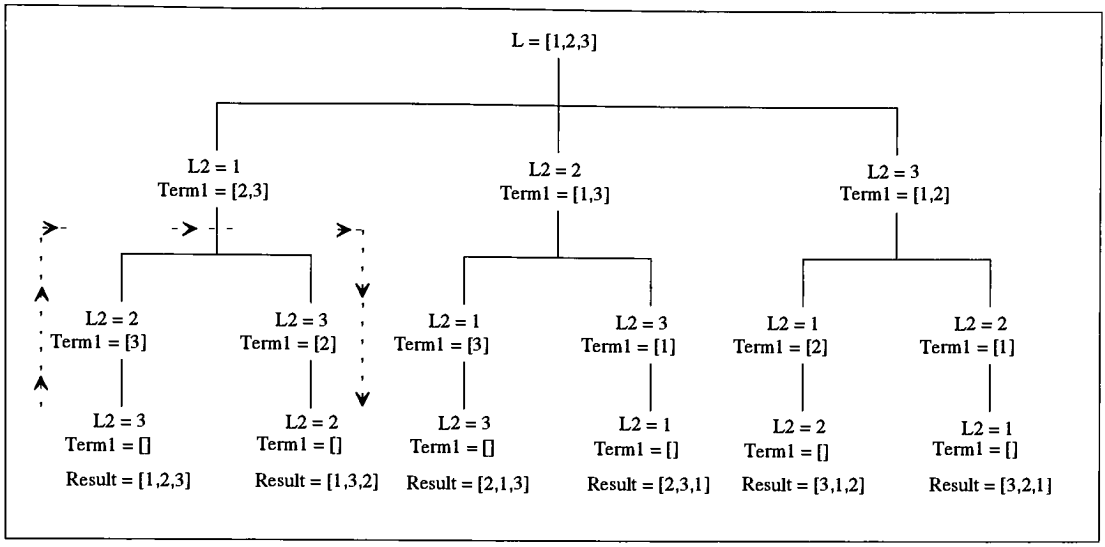


Figure 6.20 - Path traversed from the first solution to the second solution

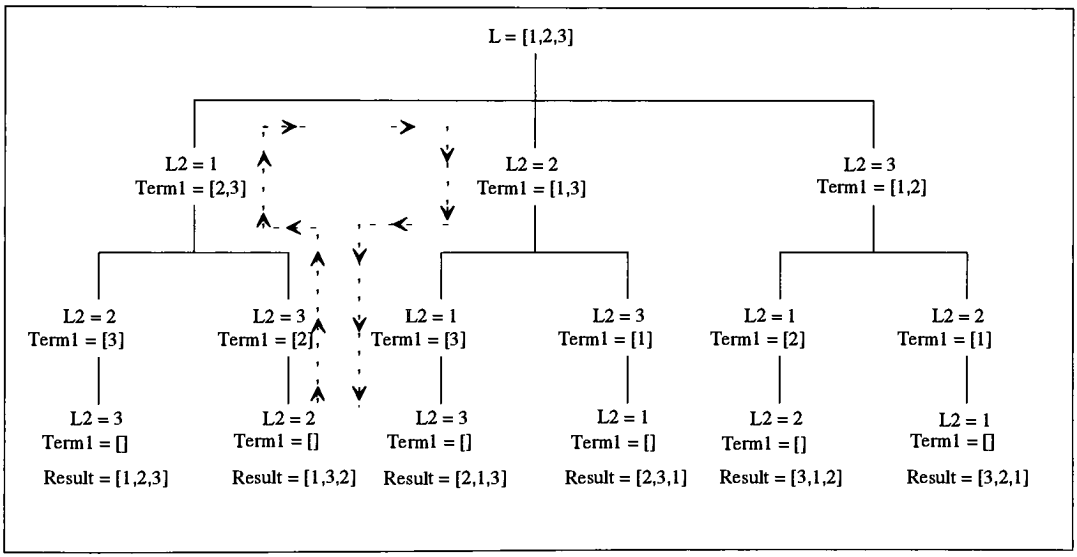


Figure 6.21 - Path traversed from the second solution to the third solution

6.5.1 IMPLEMENTATION

The structure of the tree shown in figure 6.19, will be stored in a two dimensional array. The first dimension is the level of the tree and the second dimension has a number

of components. The first component is the node number, the second component is the number of children or branches the node has, the third component is the number of branches that have been traversed completely to get a solution, the fourth component onwards contain the node numbers of the children. The structure of the array is shown in figure - 6.22.

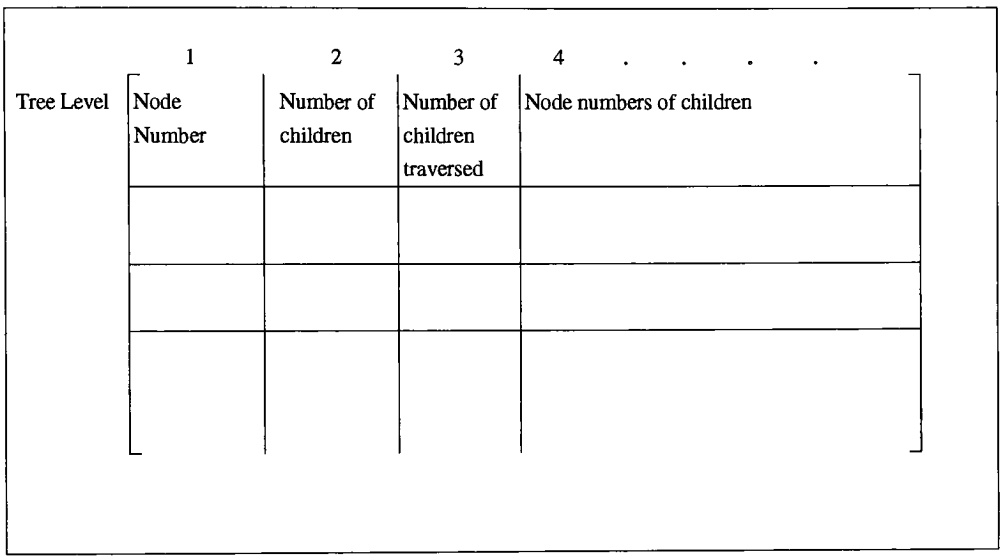


Figure 6.22 - Format of the array used to store the tree structure

The execution of the program to find the permutations of the list [1,2,3] begins with a call to delete. The call to delete returns three solutions which are nodes 1, 2 and 3 in the tree. This information will be entered in the array as shown in figure - 6.23.

Figure 6.23, shows that the level 1 of the tree has a node number 0. This node has three children and the number of children traversed is 0. The children have node numbers 1, 2 and 3. The first child i.e. node 1 is chosen to carry on the execution. The same procedure is repeated for node 1. This process goes on till a solution is obtained. The nodes are numbered as shown in figure - 6.24.

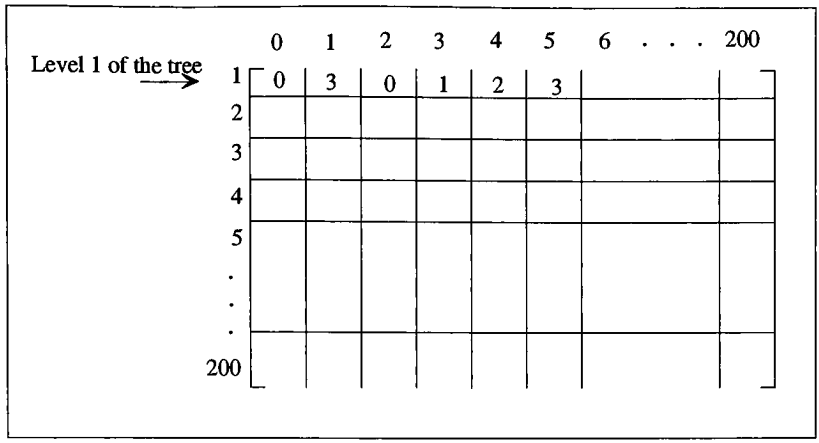


Figure 6.23 - Contents of the array tree.structure after the first call to delete

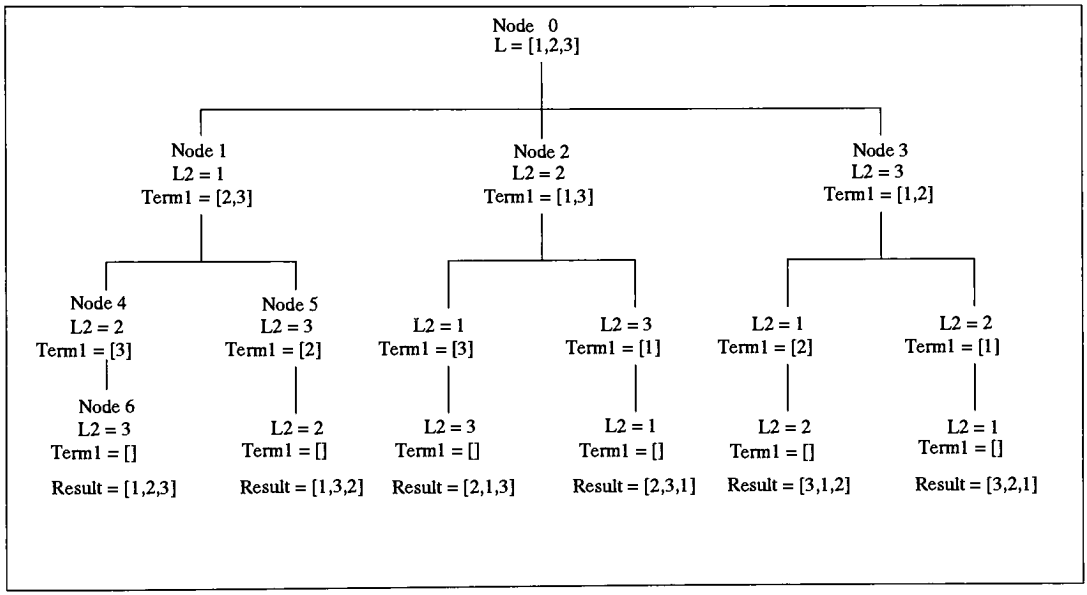


Figure 6.24 - Numbering of the nodes after the first solution is obtained

The contents of the array tree.structure after the first solution is obtained are as shown in figure - 6.25.

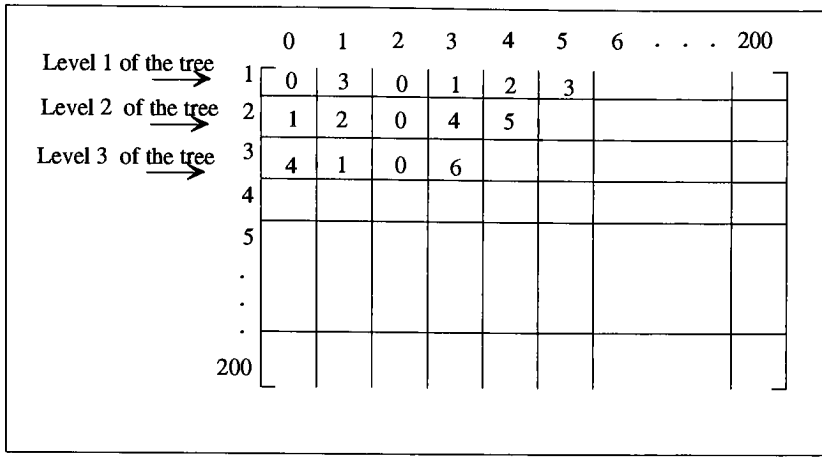


Figure 6.25 - Contents of the array tree.structure after the first solution is obtained

To get the second solution the tree has to be traversed in an upward direction till we encounter a node that has paths which have not been traversed. To traverse the tree upward, the number of children traversed of the last level of the tree i.e. level 3, is incremented by one. This quantity, which is 1 in our case, is subtracted from the number of children. This gives us a result of 0, which indicates that this node does not have any untraversed branches. The level of the tree will be decremented by 1. The same process is repeated for level 2, the number of children traversed is incremented by 1, this quantity is then subtracted from the number of children, which gives us a result of 1. This indicates that there is a branch from this node that has not been traversed. The node number of this branch or child is 5. This node is selected and execution continues from that point to obtain the second solution.

7.0 CONCLUSION

A Prolog execution model based on dataflow computation and consisting of a number of virtual machine instructions was implemented in Occam2 on a Transputer Development System with a single transputer. A few Prolog procedures were hand compiled to the virtual machine instructions, and run on the system. This project gave the details of the implementation of the virtual machine instructions. A method for exploiting small scale parallelism was presented by converting the Prolog procedures into dataflow graphs.

It was seen that Occam2 lacks some important features required for being an implementation vehicle for Prolog. The major disadvantage of Occam2 is that all data allocation is static, recursive calling of procedures and functions is not supported by the language. So, in the implementation of recursively defined data structures in Prolog, only a limited depth of recursions is allowed. The recursively defined clauses are implemented by iterations with run-time queues. Thus the depth of recursions is determined by the size of the run-time queue. For all the virtual machine instructions that use recursion, run-time queues were used. The virtual machine instruction "unify" is used to unify two lists and store the corresponding bindings in the binding environments. It uses run-time queues to unify nested lists. A list consists of a number of terms. The terms can be lists, structures, ground terms or variables. If the list contains nested lists or the structures contain lists as their components then the elements of the list have to be unified recursively. A queue was used to unify the elements. The queue was a FIFO queue implemented as a one dimensional array of integers. If any of the elements of the list is a list, the corresponding list from both the terms were pushed on two queues, to be unified later on. After the procedure had finished unifying the primary elements in both the lists, it unified those elements in the queue.

Another disadvantage of Occam2 is the lack of suitable data structures. The whole data area in Prolog, which consisted of lists, structures, variables and ground terms had to be implemented with the help of arrays. A structure which consists of a functor and a list of elements can be implemented as a single data structure in any other high level language, but in Occam2 the structure was implemented with three arrays. The first array was an integer array which stored the length of each functor in the data area, the second array was a character array which stored each character of the array and the third array was an integer array which stored the index into the list array for each structure, where the components of the structure were actually stored.

An execution model which exploited parallelism at the argument level was proposed. The Prolog procedures were converted to data-flow graphs of the virtual machine instructions. The instructions on the same horizontal level in a graph could be executed in parallel. For example the first clause of the Prolog procedure append which is append([], L2, L2) has data-flow graph shown in figure - 7.1.

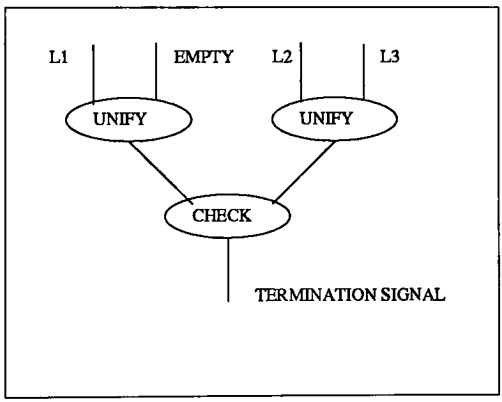


Figure 7.1 - Dataflow representation for the first clause of append

It can be seen from figure - 7.1 that the unification of list L1 with empty and that of list L2 with list L3 can be done in parallel. The unification operation, which occupies 60-70% of the execution time of Prolog programs was the main source of parallelism. My research on different forms of parallelism revealed that a number of models have been developed to exploit OR and AND parallelism inherent in Prolog. Since no models have been developed for unification parallelism, I decided to use this form of parallelism. A reason why I did not choose AND parallelism is because it requires a lot of communication due to the binding conflicts that may occur. This form of parallelism is more suited to a shared memory model. Since the transputers basically rely on message passing to share any data, they are not suitable to implement AND parallelism. I could not estimate the speed up that can be obtained by this form of parallelism because I implemented it on a single transputer. I estimate that this form of parallelism will only result in small scale parallelism for the following reasons.

(a) A lot of time will be spent on the checks for binding consistencies that have to be performed if the unification of terms is done on different processors.

(b) The maximum degree of this type of parallelism is limited by the number of arguments in an individual prolog predicate

In this project I designed and implemented the virtual machine instructions required to convert a Prolog procedure into Occam. This could be used as the back end or framework required to build a compiler or parser that converts Prolog to Occam. This compiler will be suitable only for a single transputer system. This can be recommended for future work. After my research on the unification operation and the binding consistency checks to be performed, I do not think the speed up gained would be sufficient to warrant building a compiler for a multi transputer system.

APPENDIX
IMPLEMENTATION OF THE VIRTUAL MACHINE INSTRUCTIONS

```
#INCLUDE "hostio.inc"
PROC permutation(CHAN OF SP keyboard,screen, []INT memory)
 #USE "hostio.lib"
 #USE "string.lib"
 #USE "convert.lib"
 INT len,open,close,balance,i,size,j:
 [100]BYTE line,a,p:
 [100] BYTE s:
 BOOL find,error:
 INT head2,index,l,m,i,open.cnt,current.list,last.list,temp,attribute.term:
 [50] INT list.track:
 BYTE result:
 VAL INT uninstant IS 2310:
 VAL INT greater IS 1:
 VAL INT equal IS 0:
 VAL INT less IS 2:
 VAL INT dig.const IS 0:
 VAL INT alph.const IS 1:
 VAL INT var IS 2:
 VAL INT unknown IS 3:
 VAL INT lis IS 400:
 VAL INT struct IS 800:
 VAL INT shar.var IS 1200:
 VAL INT ground.term IS 1900:
 VAL INT non.shar.var IS 2300:
 VAL INT empty IS 2301:
 VAL INT fail IS 2302:
 VAL INT split IS 2303:
 VAL INT data.area.size IS 702:
 VAL INT max.num.funcs IS 2000:
 VAL INT max.num.shar IS 2000:
 VAL INT max.ground.terms IS 700:
 VAL INT end.var IS 2304:
 VAL INT end.queue IS 2305:
 VAL INT op.sq.brack IS 2306:
 VAL INT cl.sq.brack IS 2307:
 VAL INT op.brack IS 2308:
 VAL INT cl.brack IS 2309:
 VAL INT node.number IS 0:
 VAL INT num.child IS 1:
 VAL INT num.child.trav IS 2:
```

```
[max.num.funcs] BYTE functor,a:
[struct - lis] INT func.length:
[struct - lis] INT func.list:
[max.num.shar] BYTE shar:
[max.ground.terms] INT g.terms:
[max.ground.terms] INT s.var:
[700] BYTE ground.terms:
[500] BYTE shar.vars:
[max.list.size] INT var1,envir1,var2,envir2:
[800]INT store.env,store.var:
INT j,head1,term,term3,head3,start.store.ctr,len:
BYTE result:
VAL  list2 IS [1,5]:
SEQ
 -- ********************************************************************
 -- This procedure checks if two shared variables are identical. Two variables are
 -- identical if the have the same identifiers. The procedure finds the length of the
 -- shared variables from the array s.var. It then compares each character of the
 -- variables, stored in array shar.var. If the variables are identical, it sets the variable
 -- found to 1, else it is set to 0.
 -- ********************************************************************

 PROC check.var(VAL INT tr1,tr2,INT   found)
  INT count1,count2,i,k,m :
  SEQ
   IF
    ((tr1 < shar.var) AND ( tr1 > struct)) AND ((tr2 < shar.var) AND (tr2 > struct))
     SEQ
      i := tr1 - struct
      k := tr2 - struct
      count1 := 0
      count2 := 0
      IF
       ((i   1) <> 0)
        SEQ
         SEQ m = 1 FOR (i - 1)
          count1 := s.var[m] + count1
       TRUE
        SEQ
         SKIP
      IF
       ((k - 1) <> 0)
        SEQ
         SEQ m = 1 FOR (k - 1)
          count2 := s.var[m] + count2
```

```
          TRUE
            SKIP
          found := 1
          IF
           (s.var[i] = s.var[k])
            SEQ
             m := 0
             WHILE ((m < s.var[i]) AND (found <> 0))
              SEQ
               IF
                (shar.vars[count1] = shar.vars[count2])
                 SEQ
                  count1 := count1 + 1
                  count2 := count2 + 1
                  m := m + 1
                TRUE
                 SEQ
                  found := 0
          TRUE
           found := 0
      (tr1 = tr2)
       found := 1
      TRUE
       SKIP
  :

  -- ***************************************************************************
  -- This procedure takes two arguments tr1 and tr2, to be unified. It checks if the two
  -- arguments tr1 and tr2 have been bound and have their bindings in the temporary
  -- binding environment. If both the arguments have been bound, then the procedure
  -- unifies their bindings. If tr1 is bound and tr2 is not bound, it places the binding of
  -- tr2 to the binding of tr1 in the temporary binding environment and vice versa.
  -- ***************************************************************************

  PROC check.environ(INT tr1,tr2,[] INT queue1,queue2,variable,environ,INT
queue.ptr)
    INT found:
    INT m:
    SEQ
     found := 0
     m := 0
     WHILE (variable[m] <> end.var) AND (found <> 1)
      SEQ
       check.var(variable[m],tr1,found)
       IF
```

```
    (found <> 1)
     m := m + 1
    TRUE
     SKIP
IF
 (found = 1 ) AND (environ[m] <> uninstant)
  SEQ
   temp := environ[m]
   found := 0
   m := 0
   WHILE (variable[m] <> end.var) AND   (found <> 1)
    SEQ
     check.var(variable[m],tr2,found)
     IF
      (found <> 1)
       m := m + 1
      TRUE
       SKIP
   IF
    (found = 1) AND (environ[m] <> uninstant)
     SEQ
      queue1[queue.ptr] := temp
      queue2[queue.ptr] := environ[m]
      queue.ptr := queue.ptr + 1
      queue1[queue.ptr] := end.queue
    TRUE
     SEQ
      variable[envir.num] := tr2
      environ[envir.num] := temp
      envir.num := envir.num + 1
      variable[envir.num] := end.var
 TRUE
  SEQ
   m := 0
   WHILE (variable[m] <> end.var) AND (found <> 1)
    SEQ
     check.var(variable[m],tr2,found)
     IF
      (found <> 1)
       m := m + 1
      TRUE
       SKIP
   IF
    (found = 1) AND (environ[m] <> uninstant)
     SEQ
```

```
            variable[envir.num] := tr1
            environ[envir.num] := variable[m]
            envir.num := envir.num + 1
            variable[envir.num] := end.var
          TRUE
           SEQ
            variable[envir.num] := tr1
            environ[envir.num] := tr2
            envir.num := envir.num + 1
            variable[envir.num] := end.var
  :
  PROC check.environ.one(INT trm1,trm2,[] INT variable,environ,queue1,queue2,INT
queue.ptr)
    INT found:
    INT p:
    SEQ
     found := 0
     p := 0
     WHILE (variable[p] <> end.var) AND (found <> 1)
      SEQ
       check.var(variable[p],trm1,found)
       IF
        (found <> 1)
         p := p + 1
        TRUE
         SKIP
     IF
      (found = 1)
       SEQ
        queue1[queue.ptr] := environ[p]
        queue2[queue.ptr] := trm2
        queue.ptr := queue.ptr + 1
        queue1[queue.ptr] := end.queue
      TRUE
       SEQ
        variable[envir.num] := trm1
        environ[envir.num] := trm2
        envir.num := envir.num + 1
        variable[envir.num] := end.var
  :
```

```
  -- ***************************************************************
  -- This procedure unifies those terms that can be unified directly and stores the
  -- appropriate binding messages. The arrays variable and environ are used to store the
  -- binding messages. The array variable stores the variable identifier and the array
  -- environ stores the corresponding binding.
  -- ***************************************************************

  PROC unify.elemt(VAL INT trm1,trm2,i,[] INT variable,environ,[] INT
queue1,queue2,INT queue.ptr,succ)
    INT found,l,m,count1,count2,k,p,tr1,tr2:
    SEQ
      IF
       ((trm1 > ground.term) AND (trm1 < non.shar.var)) AND ((trm2 > ground.term)
AND (trm2 < non.shar.var))
         SEQ
          variable[envir.num] := trm1
          environ[envir.num] := trm2
          envir.num := envir.num + 1
          variable[envir.num] := end.var
          variable[envir.num] := trm2
          environ[envir.num] := trm1
          envir.num := envir.num + 1
          variable[envir.num] := end.var

       (trm1 > struct) AND (trm1 < shar.var) AND (trm2 = empty)
         succ := 0
       (trm1 > ground.term) AND (trm1 < non.shar.var) AND (trm2 = empty)
         succ := 0
       (trm2 > struct) AND (trm2 < shar.var) AND (trm1 = empty)
         succ := 0
       (trm2 > ground.term) AND (trm2 < non.shar.var) AND (trm1 = empty)
         succ := 0
       (trm1 < non.shar.var) AND (trm1 > ground.term) AND ((trm2 < ground.term) OR
(trm2 > non.shar.var))
         SEQ
          variable[envir.num] := trm1
          environ[envir.num] := trm2
          envir.num := envir.num + 1
          variable[envir.num] := end.var
       (trm2 < non.shar.var) AND (trm2 > ground.term) AND ((trm1 < ground.term) OR
(trm1 > non.shar.var))
         SEQ
          variable[envir.num] := trm2
          environ[envir.num] := trm1
```

```
   envir.num := envir.num + 1
   variable[envir.num] := end.var
(trm1 = uninstant) OR (trm2 = uninstant)
  SKIP
(trm1 = empty) AND (trm2 = empty)
  SEQ
   succ := 1
   variable[envir.num] := end.var
(trm1 < lis) AND (trm2 < lis)
  SEQ
   IF
    (list[trm1][i] = list[trm2][i])
      SEQ
       variable[envir.num] := end.var
       succ := 1
    TRUE
      succ := 0
(trm1 < shar.var) AND (trm1 > struct)
  IF
   (trm2 > shar.var) OR (trm2 < struct)
     SEQ
      tr1 := trm1
      tr2 := trm2
     check.environ.one(tr1,tr2,variable,environ,queue1,queue2,queue.ptr)
   TRUE
     SEQ
      tr1 := trm1
      tr2 := trm2
     check.environ(tr1,tr2,queue1,queue2,variable,environ,queue.ptr)
(trm2 < shar.var) AND (trm2 > struct)
  IF
   (trm1 > shar.var) OR (trm1 < struct)
     SEQ
      found := 0
      p := 0
      WHILE (variable[p] <> end.var) AND (found <> 1)
        SEQ
         check.var(variable[p],trm2,found)
         IF
           (found <> 1)
             p := p + 1
           TRUE
             SKIP
      IF
       (found = 1)
```

```
                SEQ
                  queue1[queue.ptr] := environ[p]
                  queue2[queue.ptr] := trm1
                  queue.ptr := queue.ptr + 1
                  queue1[queue.ptr] := end.queue
                TRUE
                  SEQ
                    variable[envir.num] := trm2
                    environ[envir.num] := trm1
                    envir.num := envir.num + 1
                    variable[envir.num] := end.var
            TRUE
              SEQ
                tr1 := trm1
                tr2 := trm2
                check.environ(tr1,tr2,queue1,queue2,variable,environ,queue.ptr)
        (trm1 < ground.term) AND (trm1 > shar.var) AND (trm2 < ground.term) AND
(trm2 > shar.var)
          SEQ
            IF
              g.terms[trm1 - shar.var] = g.terms[trm2 - shar.var]
                SEQ
                  l := trm1 - shar.var
                  m := trm2 - shar.var
                  count1 := 0
                  count2 := 0
                  IF
                    (l <> 1)
                      SEQ
                        SEQ k = 1 FOR (l - 1)
                          count1 := g.terms[k] + count1
                    TRUE
                      SKIP
                  IF
                    (m <> 1)
                      SEQ
                        SEQ k = 1 FOR (m - 1)
                          count2 := g.terms[k] + count2
                    TRUE
                      SKIP
                  k := 0
                  WHILE (k < g.terms[l]) AND (succ = 1)
                    SEQ
                      IF
                        (ground.terms[count1] = ground.terms[count2])
```

```
                SEQ
                  count1 := count1 + 1
                  count2 := count2 + 1
                  k := k + 1
                TRUE
                  succ := 0
            variable[envir.num] := end.var
          TRUE
            succ := 0
      ((trm1 = split) AND (trm2 = split))
        variable[envir.num] := end.var
      TRUE
        succ := 0
  :

  -- ************************************************************************
  -- This procedure checks if the lists to be unified have the same length, if they do not it
  -- checks if the lists can be unified and it returns the length of the lists to be unified.
  -- ************************************************************************

  PROC check.list.length(VAL INT trm1,trm2,INT succ,length.min,[] INT
variable,environ,queue1,queue2, INT queue.ptr)
    INT m,tr1,tr2,ext,extra,num.extra.terms,trm.index,i,length.list1,length.list2,temp:
    [max.list.size] INT term2.extra,term1.extra:
    INT found :
    SEQ
      length.list1 := list[trm1][0]
      length.list2 := list[trm2][0]
      IF
        ((list[trm1][length.list1] = split) AND (list[trm2][length.list2] = split))
          SEQ
            extra := (list[trm2][0] - 1 ) - (list[trm1][0] - 1)
            IF
              (list[trm2][0] > list[trm1][0])
                ext :=  greater
              (list[trm2][0] < list[trm1][0])
                ext := less
              TRUE
                ext := equal
        (list[trm1][length.list1] = split)
          SEQ
            IF
              (list[trm2][0] > (list[trm1][0] - 1))
                ext :=  greater
              (list[trm2][0] < (list[trm1][0]   1))
```

```
            ext := less
          TRUE
            ext := equal
        extra := list[trm2][0] - (list[trm1][0] - 1)

      (list[trm2][length.list2] = split)
        SEQ
          IF
           ((list[trm2][0] - 1)  > list[trm1][0] )
             ext :=  greater
           ((list[trm2][0] - 1)< list[trm1][0] )
             ext := less
           TRUE
             ext := equal
         extra := (list[trm2][0] - 1) - list[trm1][0]
       TRUE
        SEQ
         IF
          (list[trm2][0]   > list[trm1][0] )
            ext :=  greater
          (list[trm2][0] < list[trm1][0] )
            ext := less
          TRUE
            ext := equal
        extra := list[trm2][0] - list[trm1][0]

    IF
     ((ext=equal)  AND (succ = 1))
      SEQ
       IF
        (list[trm1][length.list1] = split) AND (list[trm2][length.list2] = split)
          SEQ
            IF
             (list[trm1][length.list1 + 1] > struct) AND (list[trm1][length.list1 + 1] <
shar.var)  AND (list[trm2][length.list2 + 1] > struct) AND (list[trm2][length.list2 + 1] <
shar.var)
               SEQ
                tr1 := list[trm1][length.list1 + 1]
                tr2 := list[trm2][length.list2 + 1]
                check.environ(tr1,tr2,queue1,queue2,variable,environ,queue.ptr)
                length.min := length.list1   1

             (list[trm1][length.list1 + 1] > struct) AND (list[trm1][length.list1 + 1] <
shar.var)
               SEQ
```

```
                    length.min := length.list1   1
                    tr1 := list[trm1][length.list1 + 1]
                    tr2 := list[trm2][length.list2 + 1]
                    check.environ.one(tr1,tr2,variable,environ,queue1,queue2,queue.ptr)
                  (list[trm2][length.list2 + 1] > struct) AND (list[trm2][length.list2 + 1] <
shar.var)
                   SEQ
                    length.min := length.list2 - 1
                    tr1 := list[trm1][length.list1 + 1]
                    tr2 := list[trm2][length.list2 + 1]
                    check.environ.one(tr2,tr1,variable,environ,queue1,queue2,queue.ptr)
                  TRUE
                   length.min := length.list1 + 1
             (list[trm1][length.list1] = split)
              IF
               (list[trm1][length.list1 + 1] > struct) AND (list[trm1][length.list1 + 1] <
shar.var)
                SEQ
                 tr1 := list[trm1][length.list1 + 1]
                 tr2 := empty
                 check.environ.one(tr1,tr2,variable,environ,queue1,queue2,queue.ptr)
                 length.min := length.list1   1
                TRUE
                 succ := 0
             (list[trm2][length.list2] = split)
              IF
               (list[trm2][length.list2+ 1] > struct) AND (list[trm2][length.list2 + 1] <
shar.var)
                SEQ
                 tr2 := list[trm2][length.list2 + 1]
                 tr1 := empty
                 check.environ.one(tr2,tr1,variable,environ,queue1,queue2,queue.ptr)
                 length.min := length.list2 - 1
                TRUE
                 succ := 0
             TRUE
              SEQ
               length.min := length.list2
        (ext = greater)
         SEQ
          IF
           (list[trm1][length.list1] = split) AND (list[trm2][length.list2] = split)
            length.min := length.list1   1
           (list[trm1][length.list1] = split)
            length.min := length.list1   1
```

```
          TRUE
            succ := 0
          IF
          (list[trm1][length.list1] = split) AND (list[trm2][length.list2] = split) AND (succ
= 1)
            SEQ
             extra := length.list2 - length.list1
             SEQ i = 1 FOR extra
               SEQ
                trm.index := length.min + i
                term2.extra[i] := list[trm2][trm.index]
              IF
               (list[trm1][length.list1 + 1] > struct) AND (list[trm1][length.list1 + 1] <
shar.var)
                 SEQ
                  IF
                   (list[trm2][length.list2 + 1] < lis)
                     SEQ
                      tr1 := list[trm1][length.list1 + 1]
                      temp := list[trm2][length.list2 + 1]
                      SEQ i = 1 FOR list[temp][0]
                        term2.extra[extra + i] := list[temp][i]
                      num.extra.terms := extra + list[temp][0]
                      list[list.ctr][0] := num.extra.terms
                      [list[list.ctr] FROM 1 FOR num.extra.terms] := [term2.extra FROM
1 FOR num.extra.terms]

check.environ.one(tr1,list.ctr,variable,environ,queue1,queue2,queue.ptr)
                      list.ctr := list.ctr + 1
                    TRUE
                     SEQ
                      list[list.ctr][0] := extra + 1
                      [list[list.ctr] FROM 1 FOR extra] := [term2.extra FROM 1 FOR
extra]
                      list[list.ctr][extra + 1] := split
                       IF
                        (list[trm2][length.list2 + 1] > struct) AND (list[trm2][length.list2 +
1] < shar.var)
                          SEQ
                           m := 0
                           found := 0
                           WHILE (variable[m] <> end.var) AND (found <> 1)
                             SEQ
                              check.var(variable[m],list[trm2][length.list2 + 1],found)
                               IF
```

```
                                (found <> 1)
                                  m := m + 1
                                TRUE
                                  SKIP
                              IF
                                (found = 1)
                                  SEQ
                                    list[list.ctr][extra + 2] := environ[m]
                                TRUE
                                  list[list.ctr][extra + 2] := list[trm2][length.list2 + 1]
                            TRUE
                              list[list.ctr][extra + 2] := list[trm2][length.list2 + 1]

                          tr1 := list[trm1][length.list1 + 1]

check.environ.one(tr1,list.ctr,variable,environ,queue1,queue2,queue.ptr)
                          list.ctr := list.ctr + 1
                      (list[trm1][length.list1 + 1] < lis)
                        SEQ
                          IF
                            (list[trm2][length.list2 + 1] < lis)
                              SEQ
                                temp := list[trm2][length.list2 + 1]
                                SEQ i = 1 FOR list[temp][0]
                                  term2.extra[extra + i] := list[temp][i]
                                num.extra.terms := extra + list[temp][0]
                                list[list.ctr][0] := num.extra.terms
                                [list[list.ctr] FROM 1 FOR num.extra.terms] := [term2.extra FROM
1 FOR num.extra.terms]
                                queue1[queue.ptr] := list.ctr
                                queue2[queue.ptr] := list[trm1][length.list1 + 1]
                                queue.ptr := queue.ptr + 1
                                queue1[queue.ptr] := end.queue
                            TRUE
                              succ := 0
                        TRUE
                          succ := 0
                    (list[trm1][length.list1 ] = split) AND (succ = 1)
                      IF
                        (list[trm1][length.list1+1] > struct) AND (list[trm1][length.list1+1] <
shar.var)
                          SEQ
                            num.extra.terms:= list[trm2][0] - (list[trm1][0] - 1)
                            SEQ i=1 FOR num.extra.terms
                              SEQ
```

```
                trm.index := length.min + i
                term2.extra[i] := list[trm2][trm.index]
              list[list.ctr][0]:= num.extra.terms
              [list[list.ctr] FROM 1 FOR num.extra.terms] := [term2.extra FROM 1
FOR num.extra.terms]
              tr1 := list[trm1][length.list1 + 1]
              check.environ.one(tr1,list.ctr,variable,environ,queue1,queue2,queue.ptr)
              list.ctr := list.ctr + 1
            (list[trm1][length.list1 +1] < lis)
              SEQ
              num.extra.terms := list[trm2][0] - (list[trm1][0] - 1)
              SEQ i = 1 FOR num.extra.terms
                SEQ
                trm.index := length.min + i
                term2.extra[i] := list[trm2][trm.index]
              list[list.ctr][0]:=num.extra.terms
              [list[list.ctr] FROM 1 FOR num.extra.terms] := [term2.extra FROM 1
FOR num.extra.terms]
              queue1[queue.ptr] := list[trm1][length.list1+1]
              queue2[queue.ptr] := list.ctr
              list.ctr := list.ctr + 1
              queue.ptr := queue.ptr + 1
              queue1[queue.ptr] := end.queue
            TRUE
              succ := 0
          TRUE
            succ := 0
      ( ext = less )
        SEQ
          IF
          (list[trm1][length.list1] = split) AND (list[trm2][length.list2] = split)
            length.min := length.list2 - 1
          (list[trm2][length.list2] = split)
            length.min := length.list2 - 1
          TRUE
            succ := 0
          IF
          (list[trm1][length.list1] = split) AND (list[trm2][length.list2] = split) AND (succ
= 1)
            SEQ
            extra := length.list1 - length.list2
            SEQ i = 1 FOR extra
              SEQ
              trm.index := length.min + i
              term1.extra[i] := list[trm1][trm.index]
```

```
            IF
             (list[trm2][length.list2 + 1] > struct) AND (list[trm2][length.list2 + 1] <
shar.var)
               SEQ
                IF
                 (list[trm1][length.list1 + 1] < lis)
                   SEQ
                    tr2 := list[trm2][length.list2 + 1]
                    temp := list[trm1][length.list1 + 1]
                    SEQ i = 1 FOR list[temp][0]
                      term1.extra[extra + i] := list[temp][i]
                    num.extra.terms := extra + list[temp][0]
                    list[list.ctr][0] := num.extra.terms
                    [list[list.ctr] FROM 1 FOR num.extra.terms] := [term1.extra FROM
1 FOR num.extra.terms]

check.environ.one(tr2,list.ctr,variable,environ,queue1,queue2,queue.ptr)
                    list.ctr := list.ctr + 1
                  TRUE
                   SEQ
                    list[list.ctr][0] := extra + 1
                    [list[list.ctr] FROM 1 FOR extra] := [term1.extra FROM 1 FOR
extra]
                    list[list.ctr][extra + 1] := split
                     IF
                      (list[trm1][length.list1 + 1] > struct) AND (list[trm1][length.list1 +
1] < shar.var)
                        SEQ
                         m := 0
                         found := 0
                         WHILE (variable[m] <> end.var) AND (found <> 1)
                           SEQ
                            check.var(variable[m],list[trm1][length.list1 + 1],found)
                             IF
                               (found <> 1)
                                 m := m + 1
                               TRUE
                                SKIP
                        IF
                         (found = 1)
                           SEQ
                            list[list.ctr][extra + 2] := environ[m]
                         TRUE
                          list[list.ctr][extra + 2] := list[trm1][length.list1 + 1]
                       TRUE
```

```
                    list[list.ctr][extra + 2] := list[trm1][length.list1 + 1]
                  tr2 := list[trm2][length.list2 + 1]

check.environ.one(tr2,list.ctr,variable,environ,queue1,queue2,queue.ptr)
                    list.ctr := list.ctr + 1
            (list[trm2][length.list2 + 1] < lis)
              SEQ
               IF
                (list[trm1][length.list1 + 1] < lis)
                  SEQ
                   temp := list[trm1][length.list1 + 1]
                   SEQ i = 1 FOR list[temp][0]
                     term1.extra[extra + i] := list[temp][i]
                   num.extra.terms := extra + list[temp][0]
                   list[list.ctr][0] := num.extra.terms
                   [list[list.ctr] FROM 1 FOR num.extra.terms] := [term1.extra FROM
1 FOR num.extra.terms]
                   queue1[queue.ptr] := list.ctr
                   queue2[queue.ptr] := list[trm2][length.list2 + 1]
                   queue.ptr := queue.ptr + 1
                   queue1[queue.ptr] := end.queue
                 TRUE
                  succ := 0
            TRUE
             succ := 0
          (list[trm2][length.list2] = split) AND (succ = 1)
            IF
             (list[trm2][length.list2+1] > struct) AND (list[trm2][length.list2+1] <
shar.var)
              SEQ
               num.extra.terms := list[trm1][0] - (list[trm2][0]   1)
               SEQ i = 1 FOR num.extra.terms
                 SEQ
                  trm.index := length.min + i
                  term1.extra[i] := list[trm1][trm.index]
              variable[envir.num]:=list[trm2][length.min+2]
              list[list.ctr][0]:=num.extra.terms
              [list[list.ctr] FROM 1 FOR num.extra.terms] := [term1.extra FROM 1
FOR num.extra.terms]
              environ[envir.num] := list.ctr
              list.ctr := list.ctr + 1
              envir.num := envir.num + 1
              variable[envir.num] := end.var
            (list[trm2][length.list2 +1] < lis)
              SEQ
```

```
                num.extra.terms := list[trm1][0] - (list[trm2][0] - 1)
                SEQ i = 1 FOR num.extra.terms
                  SEQ
                    trm.index := length.min + i
                    term1.extra[i] := list[trm1][trm.index]
                list[list.ctr][0]:=num.extra.terms
                [list[list.ctr] FROM 1 FOR num.extra.terms] := [term1.extra FROM 1
FOR num.extra.terms]
                queue1[queue.ptr] := list[trm2][length.list2+1]
                queue2[queue.ptr] := list.ctr
                list.ctr := list.ctr + 1
                queue.ptr := queue.ptr + 1
                queue1[queue.ptr] := end.queue
              TRUE
                succ := 0
            TRUE
              succ := 0
  :

  -- ************************************************************************
  -- This procedure checks if two structures have the same functor. Two functors are
  -- identical if the have the same identifiers. The procedure finds the length of the
  -- functors from the array func.length. It then compares each character of the
  -- functors, stored in array functor. If the functors are identical, it sets the variable
  -- succ to 1, else it is set to 0.
  -- ************************************************************************

  PROC check.functor(VAL INT tr1,tr2,INT succ)
    INT count1,count2,i,j:
    SEQ
      i := tr1   lis
      j := tr2 - lis
      count1 := 0
      count2 := 0
      IF
        ((i - 1) <> 0)
          SEQ
            SEQ k= 1 FOR (i - 1)
              count1 := func.length[k] + count1
        TRUE
          SKIP
      IF
        ((j - 1) <> 0)
          SEQ
            SEQ k = 1 FOR (j - 1)
```

```
          count2 := func.length[k] + count2
      TRUE
        SKIP
    IF
     (func.length[i] = func.length[j])
      SEQ
       SEQ l= 1 FOR func.length[i]
        IF
         (functor[count1] = functor[count2])
          SEQ
           count1 := count1 + 1
           count2 := count2 + 1
         TRUE
          SEQ
           succ := 0
           count1 := count1 + 1
           count2 := count2 + 1
     TRUE
      succ := 0
:

-- ********************************************************************
-- A list consists of a number of terms. The terms can be lists, structures, ground
-- terms or variables. If the list contains nested lists or the structures contain lists as
-- their components then the elements of the list have to be unified recursively. Since
-- Occam does not support recursion a queue is used to unify the nested lists. The
-- queue is a FIFO queue implemented as a one dimensional array of integers. If any of
-- the elements of the list is a list, the corresponding list from both the terms are
-- put  in two queues. This procedure will unify those elements in the queue, after
-- the procedure unify.list has finished unifying the basic components of the list.
-- ********************************************************************


PROC queue.check([] INT queue1,queue2,INT succ,queue.ptr,[]INT variable,environ)
  INT tr1,tr2,q,m,queue.var1,queue.var2,length.min:
  SEQ
   m := 0
   WHILE(queue1[m] <> end.queue) AND (succ = 1)
    SEQ
     IF
      ((queue1[m] < lis) AND (queue2[m] < lis)) OR ((queue1[m] < struct) AND
(queue1[m] > lis) AND (queue2[m] < struct) AND (queue2[m] > lis))
        SEQ
         queue.var1 := queue1[m]
         queue.var2 := queue2[m]
```

```
check.list.length(queue.var1,queue.var2,succ,length.min,variable,environ,queue1,queue2,q
ueue.ptr)
            q := 1
            WHILE (q <= length.min)  AND (succ = 1)
              IF
               (list[queue.var1][q] < lis) AND (list[queue.var2][q] < lis)
                 SEQ
                  queue1[queue.ptr] := list[queue.var1][q]
                  queue2[queue.ptr] := list[queue.var2][q]
                  queue1[queue.ptr +1] := end.queue
                  queue.ptr := queue.ptr + 1
                  q := q + 1
               (list[queue.var1][q] > lis) AND (list[queue.var1][q] < struct) AND
(list[queue.var2][q] >lis) AND (list[queue.var2][q] < struct)
                 SEQ
                  check.functor(list[queue.var1][q],list[queue.var2][q],succ)
                  IF
                   (succ = 0)
                     SKIP
                   TRUE
                     SEQ
                      queue1[queue.ptr] := func.list[list[queue.var1][q] - lis]
                      queue2[queue.ptr] := func.list[list[queue.var2][q] -lis]
                      queue1[queue.ptr + 1] := end.queue
                      queue.ptr := queue.ptr + 1
                      q := q + 1
               TRUE
                 SEQ
                  tr1 := list[queue.var1][q]
                  tr2 := list[queue.var2][q]
                  unify.elemt(tr1,tr2,q,variable,environ,queue1,queue2,queue.ptr,succ)
                  q := q + 1
          TRUE
            SEQ
              tr1 := queue1[m]
              tr2 := queue2[m]
              q := 1
             unify.elemt(tr1,tr2,q,variable,environ,queue1,queue2,queue.ptr,succ)
        m := m + 1
  :
```

```
  -- ***********************************************************************
  -- This procedure is used to unify two lists. It calls a procedure to check if the lists
  -- have the same length and can be unified, the procedure also returns the length of the
  -- lists to be unified. This procedure takes two arguments, that are lists to be unified.
  -- It then unifies the corresponding elements of both the lists. If the elements are lists,
  -- then they are placed on the queue, to be unified later on.
  -- ***********************************************************************

  PROC unify.list(INT trm1,trm2,succ,[] INT variable,environ,[]INT queue1,queue2,INT
queue.ptr )
    INT length.min,i,j,p:
    SEQ

check.list.length(trm1,trm2,succ,length.min,variable,environ,queue1,queue2,queue.ptr)
      i := 1
      WHILE (succ = 1) AND (i <= length.min)
        SEQ
          IF
           (list[trm1][i] < lis) AND (list[trm2][i] < lis)
             SEQ
              queue1[queue.ptr] := list[trm1][i]
              queue2[queue.ptr] := list[trm2][i]
              queue1[queue.ptr + 1] := end.queue
              queue.ptr := queue.ptr + 1
           (list[trm1][i] < struct) AND (list[trm1][i] > lis) AND (list[trm2][i] < struct) AND
(list[trm2][i] > lis)
             SEQ
              check.functor(list[trm1][i],list[trm2][i],succ)
              IF
                (succ = 1)
                  SEQ
                   queue1[queue.ptr] := func.list[list[trm1][i] - lis]
                   queue2[queue.ptr] := func.list[list[trm2][i] - lis]
                   queue1[queue.ptr + 1] := end.queue
                   queue.ptr := queue.ptr + 1
                TRUE
                  SKIP
           TRUE
             SEQ

unify.elemt(list[trm1][i],list[trm2][i],i,variable,environ,queue1,queue2,queue.ptr,succ)


        i := i+1
      IF
```

```
    (queue.ptr <> 0) AND (succ = 1)
      SEQ
       queue.check(queue1,queue2,succ,queue.ptr,variable,environ)
    TRUE
      SKIP
  IF
    (succ = 0)
     variable[0] := end.var
    TRUE
     SKIP
:

-- ********************************************************************
-- This instruction unifies two input operands and ouputs a binding environment. It
-- stores the bindings in two integer arrays called variable and environmet. These two
-- arrays form the temporary binding environment. The array variable stores the
-- variable identifier and the array environment stores the binding. If the two terms are
-- lists, this procedure calls a function unify.list to unify the two lists. If the two
-- terms are structures, it checks if the two terms have the same functors. If they have
-- the same functors, it then calls the function unify.list to unify the components of
-- the lists. If the terms are neither lists or structures, the procedure calls another
-- procedure unify.elemt to unify the terms.
-- ********************************************************************

PROC unify(VAL  INT trm1,trm2,INT succ,[]INT variable,environ)
 INT queue.ptr,extra,length.min,num.extra.terms,trm.index,i,j,count1,count2,p,tr1,tr2:
 [max.list.size]INT term2.extra,term1.extra:
 [500] INT queue1,queue2:
 SEQ
  envir.num := 0
  succ := 1
  queue.ptr := 0
  queue1[queue.ptr] := end.queue
  variable[envir.num] := end.var
  IF
   (trm1<lis) AND (trm2<lis)
     SEQ
      tr1 := trm1
      tr2 := trm2
      unify.list(tr1,tr2,succ,variable,environ,queue1,queue2,queue.ptr)
   (trm1 < struct) AND (trm1 > lis) AND (trm2 < struct) AND (trm2 > lis)
     SEQ
      i := trm1 - lis
      j := trm2 - lis
      check.functor(trm1,trm2,succ)
```

```
          IF
           (succ = 1)
            SEQ
             tr1 := func.list[i]
             tr2 := func.list[j]
             unify.list(tr1,tr2 ,succ,variable,environ,queue1,queue2,queue.ptr)
           TRUE
            succ := 0
       (trm1 < lis)
        SEQ
         IF
          (list[trm1][1] = split)
           SEQ
            IF
             ((list[trm1][2] < struct) AND (list[trm1][2] > lis)) AND ((trm2 < struct)
AND (trm2 > lis))
               SEQ
                i := list[trm1][2] - lis
                j := trm2 - lis
                check.functor(list[trm1][2],trm2,succ)
                IF
                 (succ = 1)
                  SEQ
                   tr1 := func.list[i]
                   tr2 := func.list[j]
                   unify.list(tr1,tr2 ,succ,variable,environ,queue1,queue2,queue.ptr)
                 TRUE
                  succ := 0
              TRUE
               SEQ
                i := 1

unify.elemt(list[trm1][2],trm2,i,variable,environ,queue1,queue2,queue.ptr,succ)
          ((trm2 > struct) AND (trm2 < shar.var) ) OR ((trm2 < non.shar.var) AND
(trm2 > ground.term))
            SEQ
             i := 1
             unify.elemt(trm1,trm2,i,variable,environ,queue1,queue2,queue.ptr,succ)

           TRUE
            succ := 0
       (trm2 < lis)
        SEQ
         IF
          (list[trm2][1] = split)
```

```
            SEQ
              IF
               ((list[trm2][2] < struct) AND (list[trm2][2] > lis)) AND ((trm1 > lis) AND
(trm1 < struct))
                 SEQ
                  i := list[trm2][2] - lis
                  j := trm1 - lis
                  check.functor(list[trm2][2],trm1,succ)
                  IF
                   (succ = 1)
                     SEQ
                      tr1 := func.list[i]
                      tr2 := func.list[j]
                      unify.list(tr1,tr2 ,succ,variable,environ,queue1,queue2,queue.ptr)
                   TRUE
                     succ := 0
               TRUE
                SEQ
                 i := 1

unify.elemt(trm1,list[trm2][2],i,variable,environ,queue1,queue2,queue.ptr,succ)
          ((trm1 > struct) AND (trm1 < shar.var) ) OR ((trm1 < non.shar.var) AND
(trm1 > ground.term))
             SEQ
              i := 1
              unify.elemt(trm1,trm2,i,variable,environ,queue1,queue2,queue.ptr,succ)
            TRUE
             succ := 0

       TRUE
        INT i:
        SEQ
         i := 1
         unify.elemt(trm1,trm2,i,variable,environ,queue1,queue2,queue.ptr,succ)
  :
  PROC check.dec(INT temp,tem,end.store.ctr,process.num,[]INT
store.process,store.envir,store.va)
    INT k:
    SEQ
     IF
      (process.num = 1)
       k := 0
      TRUE
       k := store.process[process.num   1] +  1
     WHILE(k <> end.store.ctr)
```

```
      SEQ
       IF
        (store.va[k] = temp)
         store.envir[k] := tem
        TRUE
         SKIP
       k := k + 1
  :

  -- ********************************************************************
  -- This instruction decomposes the input term into a list head and tail list. It takes the
  -- term identifier as its argument and produces a head and modifies the term identifier
  -- to produce another list. If the term is a shared or non-shared variable, the
  -- instruction generates another two shared variables H and T as the list head and tail
  -- respectively, it then creates a new binding message with the input shared variable
  -- and the binding [H|T] and stores the binding message into the stored binding
  -- environment. If the instruction is neither a list nor a shared variable, the instruction
  -- generates a fail.
  -- ********************************************************************

  PROC decompose([]INT store.envir,store.va,store.process,INT
end.store.ctr,process.num,trm1,head,BOOL run)
    INT tem,trm2:
    INT term.ctr,num.terms:
    SEQ
     IF
      (trm1 = empty)
       SEQ
        run := FALSE
        find := FALSE
      (trm1<lis)
       SEQ
        IF
         (list[trm1][1] = split)
          SEQ
           IF
            (list[trm1][2] < lis)
             SEQ
              head := list[list[trm1][2]][1]
              IF
               ((list[list[trm1][2]][0]   1) = 0)
                trm1 := empty
               TRUE
                SEQ
                 trm2 := list[trm1][2]
```

```
                    trm1 := list.ctr
                    list.ctr := list.ctr+ 1
                    IF
                     (list[trm2][list[trm2][0]] = split)
                       term.ctr := list[trm2][0]
                     TRUE
                       term.ctr := list[trm2][0] - 1
                    list[trm1][0] := term.ctr
                    [list[trm1] FROM 1 FOR term.ctr] := [list[trm2] FROM 2 FOR
 term.ctr]
              (list[trm1][2] > shar.var) AND (list[trm1][2] < ground.term)
                SEQ
                 head := list[trm1][2]
                 trm1 := empty
               TRUE
                SEQ
                 head := list[trm1][2]
                 trm1 := empty
           TRUE
            SEQ
             head := list[trm1][1]
             IF
              ((list[trm1][0] - 1) = 0)
               trm1 := empty
              TRUE
               SEQ
                trm2 := trm1
                trm1 := list.ctr
                list.ctr := list.ctr+ 1
                IF
                 (list[trm2][list[trm2][0]] = split)
                   SEQ
                    term.ctr := list[trm2][0]
                    num.terms := term.ctr - 1
                 TRUE
                   SEQ
                    term.ctr := list[trm2][0]   1
                    num.terms := term.ctr
               list[trm1][0] := num.terms
               [list[trm1] FROM 1 FOR term.ctr] := [list[trm2] FROM 2 FOR term.ctr]
     ((trm1 < shar.var) AND (trm1 > struct)) OR ((trm1 > ground.term) AND (trm1 <
non.shar.var))
       SEQ
        trm2 := trm1
        head := non.shar.var.ctr
```

```
        non.shar.var.ctr := non.shar.var.ctr + 1
        trm1 := non.shar.var.ctr
        non.shar.var.ctr := non.shar.var.ctr + 1
        tem  := list.ctr
        list.ctr := list.ctr + 1
        list[tem][0] := 2
        list[tem][1] := head
        list[tem][2] := split
        list[tem][3] := trm1
        store.va[end.store.ctr] := trm2
        store.envir[end.store.ctr] := tem
        end.store.ctr := end.store.ctr + 1
       check.dec(trm2,tem,end.store.ctr,process.num,store.process,store.envir,store.va)
     (trm1 > lis) AND (trm1 < struct)
       SEQ
        run := FALSE
        find := FALSE
     (trm1 < ground.term) AND (trm1 > shar.var)
       SEQ
        run := FALSE
        find := FALSE
     (trm1 = empty)
       SEQ
        run := FALSE
        find := FALSE
     TRUE
       SKIP
:

-- ************************************************************************
--The bindings of variables to terms can be lists, structures, ground terms or variables.
-- If the binding is a list or a structure, the components of the list or structure are not
-- checked from the main procedure for check, but they are pushed on a queue. This
-- procedure is used to check elements on the queue.
-- ************************************************************************
PROC store.queue.check([]INT store.queue,INT var,env,queue.ptr)
 INT ptr,num.terms,temp,num.list,found :
 SEQ
   ptr := 0
   WHILE (store.queue[ptr] <> end.queue)
    SEQ
     temp := list[store.queue[ptr]][0]
     IF
      (list[store.queue[ptr]][temp] = split)
        num.terms := temp + 1
```

```
          TRUE
            num.terms := temp
          SEQ j = 1 FOR num.terms
            SEQ
              IF
                (list[store.queue[ptr]][j] < lis)
                  SEQ
                    store.queue[queue.ptr] := list[store.queue[ptr]][j]
                    store.queue[queue.ptr + 1] := end.queue
                    queue.ptr := queue.ptr + 1
                (list[store.queue[ptr]][j] > lis) AND (list[store.queue[ptr]][j] < struct)
                  SEQ
                    num.list := list[store.queue[ptr]][j] - lis
                    store.queue[queue.ptr] := func.list[num.list]
                    store.queue[queue.ptr + 1] := end.queue
                    queue.ptr := queue.ptr + 1
                (list[store.queue[ptr]][j] > struct) AND (list[store.queue[ptr]][j] < shar.var)
                  SEQ
                    found := 0
                    check.var(list[store.queue[ptr]][j],var,found)
                    IF
                      (found = 1)
                        list[store.queue[ptr]][j] := env
                      TRUE
                        SKIP
                (list[store.queue[ptr]][j] > ground.term) AND (list[store.queue[ptr]][j] <
non.shar.var)
                  SEQ
                    IF
                      (list[store.queue[ptr]][j] = var)
                        list[store.queue[ptr]][j] := env
                      TRUE
                        SKIP
                TRUE
                  SKIP
            ptr := ptr + 1
    :

    -- ************************************************************************
    -- This instruction checks each binding message (X B) of the stored binding
    -- environment to see if in the binding instance B there are one or more shared
    -- variables which have their own binding instances in other binding messages in the
    -- temporary  binding environment. A check instruction is always executed after a
    -- unify instruction. A unify instruction stores the bindings produced by unification in
    -- a temporary binding environment and a check instruction transfers the contents of a
```

```
  -- temporary binding environment to a stored binding environment if the unification is
  -- successful. If so the instruction then replaces the shared variables by these binding
  -- instances.
  -- ********************************************************************

  PROC check(INT end.store.ctr,process.num,[] INT
store.process,store.envir,store.va,variable,environ)
    INT j,k,l,y,end,num.list,num.terms,temp,queue.ptr,found,num.lis:
    [200]INT queue.store:
    SEQ
     IF
      (process.num = 1)
       SEQ
        y := 0
        end := end.store.ctr
      TRUE
       SEQ
        end := (end.store.ctr - 1) - store.process[process.num - 1]
        y := store.process[process.num - 1] + 1
     SEQ k = y FOR end
      SEQ
       j := 0
       WHILE (variable[j] <> end.var)
        SEQ
         found := 0
         check.var(store.va[k],variable[j] , found)
         IF
          (found = 1)
           SEQ
            store.envir[k] := environ[j]
            j := j + 1
          TRUE
           j := j + 1
     IF
      (envir.num = 0)
       variable[0] := end.var
      TRUE
       SKIP
     IF
      (end.store.ctr = start.store.ctr)
       SEQ
        k := 0
        WHILE (variable[k] <> end.var)
         SEQ
          store.va[end.store.ctr] := variable[k]
```

```
        store.envir[end.store.ctr] := environ[k]
        end.store.ctr := end.store.ctr + 1
        k := k + 1
  TRUE
    SKIP
IF
  (process.num = 1)
    SEQ
      y := 0
      end := end.store.ctr
  TRUE
    SEQ
      end := (end.store.ctr - 1) - store.process[process.num - 1]
      y := store.process[process.num   1] + 1
SEQ j = y FOR end
  SEQ
    k := 0
    queue.ptr := 0
    queue.store[queue.ptr] := end.queue
    WHILE variable[k] <> end.var
      SEQ
        IF
          (store.envir[j] < struct)
            SEQ
              IF
                (store.envir[j] < lis)
                  num.lis := store.envir[j]
                TRUE
                  num.lis := func.list[store.envir[j] - lis]
              temp := list[num.lis][0]
              IF
                (list[num.lis][temp] = split)
                  num.terms := temp + 1
                TRUE
                  num.terms := temp
              SEQ l = 1 FOR num.terms
                IF
                  (list[num.lis][l] < lis)
                    SEQ
                      queue.store[queue.ptr] := list[num.lis][l]
                      queue.ptr := queue.ptr + 1
                      queue.store[queue.ptr] := end.queue
                  (list[num.lis][l] > lis ) AND (list[num.lis][l] < struct)
                    SEQ
                      num.list := list[num.lis][l] - lis
```

```
            queue.store[queue.ptr] := func.list[num.list]
            queue.ptr := queue.ptr + 1
            queue.store[queue.ptr] := end.queue
          (list[num.lis][l] > struct ) AND (list[num.lis][l] < shar.var)
            SEQ
             found := 0
             check.var(variable[k],list[num.lis][l],found)
             IF
               (found = 1)
                 SEQ
                  list[num.lis][l] := environ[k]
               TRUE
                 SKIP
          (list[num.lis][l] > ground.term ) AND (list[num.lis][l] < non.shar.var)
            SEQ
             IF
              (variable[k] = list[num.lis][l])
                SEQ
                 list[num.lis][l] := environ[k]
               TRUE
                 SKIP
           TRUE
            SKIP
      (store.envir[j] < shar.var) AND (store.envir[j] > struct)
        SEQ
         found := 0
         check.var(store.envir[j],variable[k],found)
         IF
          (found = 1)
            SEQ
             store.envir[j] := environ[k]
          TRUE
            SKIP
       TRUE
        SKIP
      IF
       (queue.ptr <> 0)
        store.queue.check(queue.store,variable[k],environ[k],queue.ptr)
       TRUE
        SKIP
     k := k + 1
start.store.ctr := end.store.ctr


*****************************************************************
```

```
-- This procedure is used initialise the various counters used  at the start of execution
-- of the procedure.
-- ***************************************************************

PROC initialise.variables()
  INT q,r:
  SEQ
    list.ctr := 1
    struct.ctr := 401
    shar.var.ctr := 801
    ground.term.ctr :=1201
    non.shar.var.ctr := 1901
    start.store.ctr := 0
    succ := 1
    envir.num := 0
    find := TRUE
    SEQ q = 0 FOR 400
      SEQ r = 0 FOR 400
        list[q][r] := 0
:

-- ***************************************************************
-- This procedure is used to find the next comma encountered in the input stream. If it
-- finds a character before it finds a comma, it sets error to TRUE, else error is set to
-- FALSE.
-- ***************************************************************

PROC find.comma()
  SEQ
    WHILE (a[j] <> ',')  AND (error = FALSE) AND (j < (len - 1))
      IF
        (a[j] <> ' ')
          error := TRUE
        TRUE
          j := j + 1
:

-- ***************************************************************
-- This procedure skips all spaces till it encounters a character or a parenthesis. It also
-- finds the attribute of the term, if it is a digit or a variable or a term starting with a
-- lower case letter. It sets error to TRUE if it encounters any other character.
-- ***************************************************************

PROC find.char()
  INT f,r:
```

```
  [10] BYTE m:
  SEQ
   f := j + 1
   j := j + 1
   WHILE (f < len) AND (error = FALSE)
     SEQ
      IF
       (a[f] = ']')
         SEQ
           j := f
           f := len
       (a[f] = ' ')
         f := f+1
       (a[f] = '[')
         SEQ
           j := f
           f := len
       (a[f] >= '0') AND (a[f] <= '9')
         SEQ
           attribute.term := dig.const
           j := f
           f := len
       (a[f] >= 'a') AND (a[f] <= 'z')
         SEQ
           attribute.term := alph.const
           j := f
           f := len
       (a[f] >= 'A') AND (a[f] <= 'Z')
         SEQ
           attribute.term := var
           j := f
           f := len
       TRUE
         SEQ
           error := TRUE
:

-- ***************************************************************************
-- This procedure is used to count the parenthesis in any term. In any term the number
-- of opening parenthesis is equal to the number of closing parenthesis. this procedure
-- can be used to count the parenthesis in a structure or a list. If it reaches the end of
-- the term and the number of opening parenthesis is not equal to the number of
-- closing parenthesis it sets error to TRUE.
-- ***************************************************************************
```

```
PROC count.brackets(VAL BYTE op.bracket,cl.bracket)
  INT y:
  SEQ
    y := j
    close := 0
    open := 1
    balance := open - close
    WHILE (balance <> 0) AND (j < len)
      IF
        (a[j] = op.bracket)
          SEQ
            j := j + 1
            open := open + 1
            balance := open - close
        (a[j] = cl.bracket)
          SEQ
            j := j+1
            close := close + 1
            balance := open - close
        TRUE
          SEQ
            j := j + 1
            balance := open - close
    IF
      (j = len ) AND (balance <> 0)
        error := TRUE
      (j <> len) AND (balance = 0) AND (op.bracket = '[') AND (y = 1)
        error := TRUE
      TRUE
        SKIP
:
-- ***************************************************************
-- This procedure looks for the first character it encounters after a closing parenthesis.
-- The next character after a closing parenthesis can be either another closing
-- parenthesis or a comma, but it can not be a character. If a comma is found the
-- procedure will look for the next character which has to be an alphanumeric
-- character.
-- ***************************************************************

PROC close.bracket(INT open,close)
  SEQ
    j := j + 1
    IF
      (j < ( len   1 ))
        SEQ
```

```
        find.comma()
        IF
          (error = FALSE)
            SEQ
              find.char()
              IF
                (a[j] = ']')
                  SEQ
                    error := TRUE
                TRUE
                  SKIP
          (error = TRUE) AND (a[j] = '[')
            SEQ
              error := FALSE
          (error = TRUE) AND (a[j] = ']')
            SEQ
              j := j + 1
              error := FALSE
          (error = TRUE) AND (a[j] = ')')
            SEQ
              close := close + 1
              j := j + 1
              error := FALSE
          TRUE
            SKIP
      (j = (len - 1)) AND (a[j] = ')')
        SEQ
          close := close + 1
      TRUE
        j := j + 1
:

PROC end.split()
  SEQ
    WHILE (a[j] <> ']') AND (error = FALSE)
      SEQ
        IF
          (a[j] = ' ')
            j := j + 1
          TRUE
            error := TRUE
:

-- ***************************************************************************
-- This procedure checks if the number of  opening square parenthesis is equal to the
```

```
-- number of closing square parenthesis in any given term.
-- ***************************************************************

PROC check.balance()
  INT op.bal,cl.bal,f ,bal :
  SEQ
    f := j + 1
    op.bal := 1
    cl.bal := 0
    bal :=  op.bal - cl.bal
    WHILE (bal <> 0)
      SEQ
        IF
          (a[f] = '[')
            SEQ
              op.bal := op.bal + 1
              bal := op.bal - cl.bal
              f := f + 1
          (a[f] = ']')
            SEQ
              cl.bal := cl.bal + 1
              bal := op.bal - cl.bal
              f := f + 1
          TRUE
            f := f + 1
    WHILE (a[f] = ' ')
      f := f + 1
    IF
      (a[f] = ']')
        SKIP
      TRUE
        SEQ
          error := TRUE
          j := f
:

PROC check.after.split()
  SEQ
    j := j + 1
    IF
      (a[j] = ' ')
        SEQ
          WHILE (a[j] = ' ')
            j := j + 1
      TRUE
```

```
        SKIP
      IF
       ((a[j] >= '0') AND (a[j] <= '9'))
        SEQ
         WHILE ((a[j] >= '0') AND (a[j] <= '9'))
          SEQ
           IF
            ((a[j] >= 'a') AND (a[j] <= 'z')) OR ((a[j] >= 'A') AND (a[j] <= 'Z'))
              error := TRUE
            TRUE
              j := j + 1
         end.split()
       ((a[j] >= 'a') AND (a[j] <= 'z'))
        SEQ
         WHILE ((((a[j] >= 'a') AND (a[j] <= 'z')) OR ((a[j] >= 'A') AND (a[j] <= 'Z'))
OR ((a[j] >= '0') AND (a[j] <= '9'))) AND (a[j] <> '('))
          j := j +1
          IF
           (a[j] <> '(')
            end.split()
           TRUE
            SKIP
       ((a[j] >= 'A') AND (a[j] <= 'Z'))
        SEQ
         WHILE ((a[j] >= 'A') AND (a[j] <= 'Z')) OR ((a[j] >= 'a') AND (a[j] <= 'z')) OR
((a[j] >= '0') AND (a[j] <= '9'))
          j := j + 1
         end.split()
       (a[j] = '[')
        SEQ
         check.balance()
       TRUE
        error := TRUE
  :
  -- ***************************************************************************
  -- This procedure checks each term of a structure of. The checks that need to be done
  -- are -
  --      - Check if the terms are separated by commas
  --      - An opening square parenthesis is followed by a character or another opening
  --        square  parenthesis
  --      - An opening square parenthesis except the first one is preceded by a comma or
  --        another opening square parenthesis
  --      - A closing square parenthesis except the last one is followed by a comma or
  --        another closing square parenthesis
  --      - A closing square parenthesis is preceded by a character, closing round
```

```
--          parenthesis or a closing square parenthesis
--        - If an element has an opening round parenthesis, then its attribute has to be an
--          alphabetic constant. If it is then call a procedure to check syntax of the
--          structure.
--          If the attribute of a term is a digit constant, then it can not have an upper case
--          or lower case letter in it
--          If a 'l' is encountered in the input call a separate procedure to check the syntax
--          of the term after the split. This procedure checks if there is only one element
--          after the split. This element can be a list, character, integer, variable or
--          structure.
-- ********************************************************************

PROC check.structure()
  INT open,close,balance,init.count:
  BOOL done:
  SEQ
    open := 1
    close := 0
    balance := open - close
    j := j + 1
    WHILE (balance <> 0) AND ( error = FALSE)
      SEQ
        attribute.term := unknown
        done := FALSE
        IF
          (a[j] = '(')
            SEQ
              open := open + 1
              balance := open - close
              find.char()
              IF
                (a[j] = ']')
                  error := TRUE
                TRUE
                      SKIP
          (a[j] = ')')
            SEQ
              close := close + 1
              balance := open - close
              IF
                (j < (len - 1))
                  SEQ
                    close.bracket(open,close)
                    balance := open - close
                TRUE
```

```
            SKIP
          (a[j] = '[')
            find.char()
          (a[j] = ']')
            SEQ
              close.bracket(open,close)
              balance := open -close
          (a[j] = ',') OR (a[j] = ' ')
            j := j + 1
          TRUE
            SEQ
              WHILE (a[j] <> ',') AND (a[j] <> ']') AND (a[j] <> ')') AND (error = FALSE)
AND (done = FALSE)
                SEQ
                  IF
                    (attribute.term = unknown)
                      SEQ
                        IF
                          (a[j] >= 'A') AND (a[j] <= 'Z')
                            attribute.term := var
                          (a[j] >= '0') AND (a[j] <= '9')
                            attribute.term := dig.const
                          (a[j] >= 'a') AND (a[j] <= 'z')
                            attribute.term := alph.const
                    TRUE
                      SKIP
                  IF
                    (a[j] = '|')
                      SEQ
                        check.after.split()
                        IF
                          (a[j] = '[')
                            done := TRUE
                          TRUE
                            SKIP
                    (a[j] = '(') AND (attribute.term = dig.const)
                      error := TRUE
                    (attribute.term = dig.const) AND (a[j] >= 'a') AND (a[j] <= 'z')
                      error := TRUE
                    (attribute.term = dig.const) AND (a[j] >= 'A') AND (a[j] <= 'Z')
                      error := TRUE
                    (attribute.term = var) AND (a[j] = '(')
                      error := TRUE
                    (a[j] = '(')
                      SEQ
```

```
          find.char()
          open := open + 1
          balance := open - close
          done := TRUE
        (a[j] = ')')
          SEQ
          close := close +1
          balance := open - close
        (a[j] = ' ')
          SEQ
          find.comma()
          IF
            (error = FALSE)
              SEQ
              find.char()
              IF
                (error = FALSE) AND (a[j] = ']')
                  error := TRUE
                TRUE
                  SKIP
            (error = TRUE) AND (a[j] = ']')
              error := FALSE
            (error = TRUE) AND (a[j] = ')')
              SEQ
                        close := close + 1
                        balance := open - close
                      error := FALSE
            TRUE
              SKIP
        (a[j] = '[') OR (a[j] = ']')
          error := TRUE
        TRUE
          j := j + 1
    IF
      (error = FALSE) AND (a[j] = ',')
        SEQ
        find.char()
        IF
          (a[j] = ']')
            SEQ
              error := TRUE
          TRUE
            SKIP
      (error = FALSE) AND (a[j] = ']')
        SEQ
```

```
            close.bracket(open,close)
            balance := open - close
          (error = FALSE) AND (a[j] = ')')
           SEQ
            close := close + 1
            balance := open - close
            IF
             (j < (len - 1))
               SEQ
                close.bracket(open,close)
                balance := open - close
              TRUE
               SKIP
          TRUE
           SKIP
:

-- ************************************************************************
-- This procedure prints out exactly where the error in the input string is.
-- ************************************************************************

PROC write.error()
  SEQ
   so.write.nl(keyboard,screen)
   so.write.string(keyboard,screen,"Syntax error")
   so.write.nl(keyboard,screen)
   SEQ m = 0 FOR j
     p[m] := a[m]
   [p FROM (j+2) FOR (len - j)] := [a FROM (j+1) FOR (len - j)]
   m := 0
   SEQ m = 0 FOR j
     SEQ
       so.write.char(keyboard,screen,p[m])
   so.write.string(keyboard,screen,"  here  ")
   SEQ n= j FOR (len - j)
    so.write.char(keyboard,screen,a[n])
:

-- ************************************************************************
-- This procedure is used to check the terms of the list. The checks that need to be
-- performed on the different terms are -
--       - Check if the terms are separated by commas
--         An opening square parenthesis is followed by a character or another opening
--          square  parenthesis
--       - An opening square parenthesis except the first one is preceded by a comma or
```

```
--          another opening square parenthesis
--          A closing square parenthesis except the last one is followed by a comma or
--          another closing square parenthesis
--        - A closing square parenthesis is preceded by a character, closing round
--          parenthesis or a closing square parenthesis
--          If an element has an opening round parenthesis, then its attribute has to be an
--          alphabetic constant. If it is then call a procedure to check syntax of the
--          structure.
--        - If the attribute of a term is a digit constant, then it can not have an upper case
--          or lower case letter in it
--        - If a '|' is encountered in the input call a separate procedure to check the syntax
--          of the term after the split. This procedure checks if there is only one element
--          after the split. This element can be a list, character, integer, variable or
--          structure.
-- ***************************************************************************

PROC check.syntax()
  BYTE op.bracket,cl.bracket:
  INT f,temp:
  BOOL done:
  SEQ
    error := FALSE
    IF
      (a[j] = '[')
        SEQ
          op.bracket := '['
          cl.bracket := ']'
          j := 1
          count.brackets(op.bracket,cl.bracket)
      TRUE
        error := TRUE
    IF
      (error)
        SKIP
      TRUE
        SEQ
          j :=0
          WHILE (j < len) AND (error = FALSE)
            IF
              (a[j] = '(')
                SEQ
                  j := j + 1
                  f := j
                  op.bracket := '('
                  cl.bracket := ')'
```

```
            count.brackets(op.bracket,cl.bracket)
            IF
                    (error)
                      SKIP
                    TRUE
                      SEQ
                temp := j
                        j := f
                        close := 0
                        open := 0
                        WHILE (j < temp) AND (error = FALSE)
                          IF
                              (a[j] = '[')
                                SEQ
                                  open := open + 1
                              j := j + 1
                                  balance := open - close
                              (a[j] = ']')
                                SEQ
                                  close := close + 1
                                  j := j + 1
                                  balance := open - close
                              TRUE
                                j := j + 1
                        IF
                     (balance <> 0)
                              error := TRUE
                            TRUE
                              SKIP
          TRUE
            j := j + 1
IF
  (error)
    SKIP
  TRUE
    SEQ
      j := 0
WHILE (j < len) AND (error = FALSE) AND (a[j] <> ']')
  SEQ
    done := FALSE
    i := 0
    attribute.term := unknown
    IF
      (a[j] = '[')
        SEQ
```

```
            find.char()
          (a[j] = ']')
            close.bracket(open,close)
          (a[j] = ' ') OR (a[j] = ',')
            j := j+1
          TRUE
            SEQ
            WHILE (a[j] <> ',') AND (a[j] <> ']') AND (error = FALSE) AND (j < len)
AND (done = FALSE)
              SEQ
               IF
                (attribute.term = unknown)
                  SEQ
                   IF
                    (a[j] >= 'A') AND (a[j] <= 'Z')
                      attribute.term := var
                    (a[j] >= 'a') AND (a[j] <= 'z')
                      attribute.term := alph.const
                    (a[j] >= '0') AND (a[j] <= '9')
                      attribute.term := dig.const
                    TRUE
                      SKIP
                TRUE
                  SKIP
               IF
                (a[j] = '|')
                  SEQ
                   check.after.split()
                   IF
                    ((a[j] = '[') OR (a[j] = '('))
                      done := TRUE
                    TRUE
                      SKIP
               (a[j] = '(') AND (attribute.term = var)
                 error := TRUE
               (a[j] = '(') AND (attribute.term = dig.const)
                 error := TRUE
               (attribute.term = dig.const) AND (a[j] >= 'a') AND (a[j] <= 'z')
                 error := TRUE
               (attribute.term = dig.const) AND (a[j] >= 'A') AND (a[j] <= 'Z')
                 error := TRUE
               (a[j] = '(')
                 SEQ
                  check.structure()
                  done := TRUE
```

```
            (a[j] = ' ')
              SEQ
               find.comma()
               IF
                (error = FALSE)
                  SEQ
                   find.char()
                   IF
                    (error = FALSE) AND (a[j] = ']')
                      error := TRUE
                    TRUE
                      SKIP
                  (error = TRUE) AND (a[j] = ']')
                    error := FALSE
                  TRUE
                    SKIP
              (a[j] = '[') OR (a[j] = ']')
                error := TRUE
              TRUE
                j := j + 1
          IF
           (error = FALSE) AND (a[j] = ',')
             SEQ
              find.char()
              IF
               (a[j] = ']')
                 SEQ
                   error := TRUE
               TRUE
                 SKIP
           (error = FALSE) AND (a[j] = ']')
            close.bracket(open,close)
           TRUE
            SKIP
  IF
   (error = TRUE)
     SEQ
       write.error()
   TRUE
     SKIP
:

-- ************************************************************************
-- Each structure has entries in three arrays. The arrays are functor length, functor list
-- and functor. The array functor is a character array which stores each character of
```

```
-- the functor and the list where the components of the structure are stored is stored in
-- the array functor list. This procedure takes two arguments - the character array in
-- which the functor is stored and the length of the functor. It then puts the
-- corresponding entries of the structure in the array functor and functor list.
-- ************************************************************************

PROC assign.struct([] BYTE s,INT count)
  INT count1:
  SEQ
    func.length[struct.ctr - lis] := count
    func.list[struct.ctr - lis] := current.list
    count1 := 0
    IF
      ((struct.ctr - lis) <> 1)
        SEQ
          SEQ k = 1 FOR ((struct.ctr - lis) - 1)
            count1  := func.length[k] + count1
      TRUE
        SKIP
    [functor FROM count1 FOR count] := [s FROM 0 FOR count]
    struct.ctr := struct.ctr + 1
:

-- ************************************************************************
-- A ground term is stored in two arrays. The array ground.term.length stores the
-- length of each ground term. The characters of the ground term are stored in the
-- character array ground.term.length. This procedure stores the entries of any ground
-- term in the two array.
-- ************************************************************************

PROC assign.ground.term([]BYTE array,INT count)
  INT count1:
  SEQ
    g.terms[ground.term.ctr - shar.var] := count
    count1 := 0
    IF
      ((ground.term.ctr - shar.var) <> 1)
        SEQ
          SEQ k = 1 FOR ((ground.term.ctr - shar.var) - 1)
            count1 := g.terms[k] + count1
      TRUE
        SKIP
    [ground.terms FROM count1 FOR count] := [array FROM 0 FOR count]
    ground.term.ctr := ground.term.ctr + 1
:
```

```
-- ************************************************************************
-- A shared variable is stored in two arrays. The array shar.var.length stores the length
-- of each shared variable and the array shar.var is a character array which stores each
-- character of the shared variable. This procedure takes two arguments the character
-- array in which the shared variable is stored and the length of the shared variable,
-- and stores them in the corresponding arrays.
-- ************************************************************************

PROC assign.shar.var([] BYTE array, INT count)
  INT count1:
  SEQ
    s.var[shar.var.ctr - struct] := count
    count1 := 0
    IF
      ((shar.var.ctr - struct) <> 1)
        SEQ
          SEQ k = 1 FOR ((shar.var.ctr - struct) - 1)
            count1 := s.var[k] + count1
      TRUE
        SKIP
    [shar.vars FROM count1 FOR count] := [array FROM 0 FOR count]
    shar.var.ctr := shar.var.ctr + 1
:

-- ************************************************************************
-- This procedure is used to seperate a particular term of a list and call the appropriate
-- procedure to store the term. This procedure seperates a term by looking for a
-- comma, or a closing square parenthesis, or a closing round parenthesis. The term
-- can be either a variable, ground term or functor of a structure. If the procedure
-- encounters a closing parenthesis it adjusts the list counter so that the next term is
-- stored in the appropriate list.
-- ************************************************************************

PROC sep.char()
  [50] BYTE s:
  INT count,count1,ind:
  BOOL splt:
  SEQ
    splt :=FALSE
    count := 0
    attribute.term := unknown
    WHILE (a[j] <> ',') AND  (a[j] <> ' ') AND (a[j] <> ']')  AND (a[j] <> '(')  AND (a[j]
<> ')')  AND (splt = FALSE)
```

```
SEQ
  IF
    (attribute.term = unknown)
      SEQ
        IF
          (a[j] >= '0') AND (a[j] <= '9')
            attribute.term := dig.const
          (a[j] >= 'A') AND (a[j] <= 'Z')
            attribute.term := var
          (a[j] >= 'a') AND (a[j] <= 'z')
            attribute.term := alph.const
          TRUE
            SKIP
    TRUE
      SKIP
  IF
    (a[j] = '|' )
      SEQ
        splt := TRUE
    TRUE
      SEQ
        s[count] := a[j]
        j := j + 1
        count := count + 1
IF
  (a[j] = '(')
    SEQ
      open.cnt := open.cnt + 1
      temp := current.list
      current.list := open.cnt
      last.list := temp
      i := i + 2
      list.track[i] := last.list
      list.track[i + 1] := current.list
      index := list[last.list][0]
      IF
        (list[last.list][index + 1] = split)
          SEQ
            list[last.list][index + 2] := struct.ctr
            list[last.list][0] := list[last.list][0] + 1
        TRUE
          SEQ
            list[last.list][index + 1] := struct.ctr
            list[last.list][0] := list[last.list][0] + 1
      assign.struct(s,count)
```

```
TRUE
 SEQ
  ind := list[current.list][0]
  IF
   (attribute.term = dig.const) OR (attribute.term = alph.const)
     SEQ
      IF
       (list[current.list][ind + 1] = split)
         SEQ
          list[current.list][ind + 2] := ground.term.ctr
       TRUE
         SEQ
          list[current.list][ind + 1] := ground.term.ctr
     assign.ground.term(s,count)
     list[current.list][0] := list[current.list][0] + 1
   (attribute.term = var)
     SEQ
      IF
       (list[current.list][ind + 1] = split)
         SEQ
          list[current.list][ind + 2] := shar.var.ctr
       TRUE
         SEQ
          list[current.list][ind + 1] := shar.var.ctr
     assign.shar.var(s,count)
     list[current.list][0] := list[current.list][0] + 1
   TRUE
     SKIP

  IF
   (splt = TRUE)
     SEQ
      index := list[current.list][0]
      list[current.list][index + 1] := split
   TRUE
     SKIP
  IF
   (a[j] = ']') AND (j <> (len - 1))
     SEQ
      close := close + 1
      balance := open - close
      i := i - 2
      current.list := list.track[i + 1]
      last.list := list.track[i]
   (a[j] = ']')
```

```
          SEQ
            close := close + 1
            balance := open - close
          (a[j] = ')')
            SEQ
              i := i - 2
              current.list := list.track[i + 1]
              last.list := list.track[i]
          TRUE
            SKIP
    j := j + 1
:

-- ************************************************************************
-- This procedure is used to store a list. It calls the procedure sep.char to store one term
-- of the list. It uses an array list.track in the form of a stack to keep track of the
-- current list being stored and the list stored before the current list. This is useful in
-- storing nested lists.
-- ************************************************************************

PROC assign.list()
  SEQ
    close := 0
    open := 1
    balance := open - close
    open.cnt := list.ctr
    current.list := open.cnt
    last.list := current.list
    list.track[0] := last.list
    list.track[1] := current.list
    i := 0
    WHILE (balance <> 0)
      SEQ
        IF
          (a[j] = ']')
            SEQ
              close := close + 1
              balance := open - close
              j := j + 1
              IF
                (j < (len   1 ))
                  SEQ
                    i := i - 2
                    current.list := list.track[i + 1]
                    last.list := list.track[i]
```

```
            TRUE
              SKIP
          (a[j] = '[')
            SEQ
              open := open + 1
              balance := open - close
              j := j + 1
              open.cnt := open.cnt + 1
              temp := current.list
              current.list := open.cnt
              last.list := temp
              i := i + 2
              list.track[i] := last.list
              list.track[i + 1] := current.list
              index := list[last.list][0]
              IF
                (list[last.list][index + 1] = split)
                  SEQ
                    list[last.list][index + 2] := current.list
                    list[last.list][0] := list[last.list][0] + 1
                TRUE
                  SEQ
                    list[last.list][index + 1] := current.list
                    list[last.list][0] := list[last.list][0] + 1
          (a[j] = ')')
            SEQ
              IF
                (j < (len - 1 ))
                  SEQ
                    i := i - 2
                    current.list := list.track[i + 1]
                    last.list := list.track[i]
                TRUE
                  SKIP
              j := j + 1
          (a[j] = ' ') OR (a[j] = ',')
            j := j + 1
          TRUE
            SEQ
              sep.char()
:

-- ********************************************************************
-- In order to store the input, it is checked to see if it is in the right format. A
-- procedure called check.term performs the operations of checking the input, which
```

```
-- can be any of the data terms described above, except the non shared variables. Non
-- shared variables are generated as the program executes. In the event of an error in
-- the input an error routine is used to inform the user exactly where the error is. If the
-- input is a list, it calls a procedure check.syntax to check the syntax of the list. If the
-- syntax is okay it calls another procedure assign.list to store the terms of the list. If
-- the term is a ground term or variable, it checks their syntax and calls the
-- corresponding procedure to store the terms.
-- ********************************************************************

PROC check.term(INT trm)
  INT count1,temp:
  BYTE op.bracket,cl.bracket :
  SEQ
    error := FALSE
    IF
      (a[0] = '[')
        SEQ
          j := 0
          trm := list.ctr
          check.syntax()
          j := 1
          IF
            (error)
              SKIP
            TRUE
              SEQ
                assign.list()
                list.ctr := open.cnt + 1
      TRUE
        SEQ
          attribute.term := unknown
          j := 0
          WHILE (a[j] <> '(') AND (j < len) AND (error = FALSE)
            SEQ
              IF
                (attribute.term = unknown)
                  SEQ
                    IF
                      (a[j] >= 'A') AND (a[j] <= 'Z')
                        attribute.term := var
                      (a[j] >= '0') AND (a[j] <= '9')
                        attribute.term := dig.const
                      (a[j] >= 'a') AND (a[j] <= 'z')
                        attribute.term := alph.const
                TRUE
```

```
              SKIP
            IF
             (attribute.term = dig.const)
              SEQ
               IF
                ((a[j] >= 'a') AND (a[j] <= 'z')) OR ((a[j] >= 'A') AND (a[j] <= 'Z'))
                  error := TRUE
                TRUE
                  SKIP
             TRUE
              SKIP
            IF
             (a[j] = 'I')
              SEQ
               error := TRUE
               write.error()
             TRUE
              j := j + 1
        IF
         ((j = len) AND (error = FALSE) AND ((attribute.term = dig.const) OR
(attribute.term = alph.const)))
          SEQ
           trm := ground.term.ctr
           assign.ground.term(a,len)
         ((j = len) AND (error = FALSE) AND (attribute.term = var))
          SEQ
           trm := shar.var.ctr
           assign.shar.var(a,len)
         (a[j -1] = ' ')
          SEQ
           error := TRUE
           write.error()
         (error = FALSE) AND (attribute.term <> dig.const) AND (attribute.term <>
var)
          SEQ
           temp := j
           j := j + 1
           op.bracket := '('
           cl.bracket := ')'
           count.brackets(op.bracket,cl.bracket)
           IF
            (error)
             SEQ
              write.error()
            TRUE
```

```
              SEQ
               j := temp
               check.structure()
               IF
                (error)
                 SEQ
                  write.error()
                TRUE
                 SEQ
                  trm := struct.ctr
                  j := temp
                  current.list := list.ctr
                  assign.struct(a,j)
                  a[len-1] := ']'
                  j := temp + 1
                  assign.list()
                  list.ctr := open.cnt + 1
        (error = FALSE) AND ((attribute.term = dig.const) OR (attribute.term = var))
         error := TRUE
        TRUE
         SKIP
:

-- ************************************************************************
-- This procedure is used to print out the functor of a structure. It finds the index into
-- the character array functor from the array functor length. It then prints out each
-- character of the functor.
-- ************************************************************************

PROC print.functor(INT env)
 INT count1:
 SEQ
  count1 := 0
  IF
   ((env - lis) <> 1)
    SEQ
     SEQ k = 1 FOR ((env - lis) - 1)
      count1 := func.length[k] + count1
   TRUE
    SKIP
  SEQ k = 1 FOR func.length[env - lis]
   SEQ
    so.write.char(keyboard,screen,functor[count1])
    count1 := count1 + 1
:
```

```
-- ************************************************************************
-- This procedure is used to print out a shared variable. It calculates the index into the
-- character array shar.var from the array shar.var.length. It then prints out each
-- character of the shared variable.
-- ************************************************************************
PROC print.shar.var(INT env)
  INT count1:
  SEQ
    count1 := 0
    IF
      ((env - struct) <> 1)
        SEQ
          SEQ k = 1 FOR ((env - struct) - 1)
            count1 := s.var[k] + count1
      TRUE
        SKIP
    SEQ k = 1 FOR s.var[env - struct]
      SEQ
        so.write.char(keyboard,screen,shar.vars[count1])
        count1 := count1 + 1
:
-- ************************************************************************
-- This procedure is used to print out a ground term. It calculates the index into the
-- character array ground.terms from the array ground.term.length. It then prints out
-- each character of the shared variable.
-- ************************************************************************

PROC print.ground.term(INT env)
  INT count1:
  SEQ
    count1 := 0
    IF
      ((env - shar.var) <> 1)
        SEQ
          SEQ k = 1 FOR ((env - shar.var) - 1)
            count1 := g.terms[k] + count1
      TRUE
        SKIP
    SEQ k = 1 FOR g.terms[env - shar.var]
      SEQ
        so.write.char(keyboard,screen,ground.terms[count1])
        count1 := count1 + 1
:
```

```
-- ********************************************************************
-- This procedure prints out the contents of the the array passed to it. Lists are stored
-- as integers in an array. In order to print out a result in the form of a list, the
-- appropriate parenthesis commas and spaces have to be inserted. The array passed to
-- this procedure contains the elements of the list with the appropriate parenthesis
-- commas and spaces. Since the array is an integer array, it will contain some integer
-- values that represent parenthesis and commas. This procedure will convert the
-- integer into the appropriate character and print it out.
-- ********************************************************************

PROC print.val([] INT temp,INT count)
  SEQ
    SEQ i = 0 FOR count
      SEQ
        IF
          (temp[i] = uninstant)
            SEQ
              so.write.string(keyboard,screen,"_0")
              IF
                ((temp[i + 1] <> cl.sq.brack) AND (temp[i + 1] <> cl.brack) AND (temp[i +
1] <> split) AND (i <> (count - 1)))
                  so.write.string(keyboard,screen,",")
                TRUE
                  SKIP
          (temp[i] = op.sq.brack)
            SEQ
              so.write.string(keyboard,screen,"[")
          (temp[i] = cl.sq.brack)
            SEQ
              so.write.string(keyboard,screen,"]")
              IF
                ((temp[i + 1] <> cl.sq.brack) AND (temp[i + 1] <> cl.brack) AND (temp[i +
1] <> split) AND (i < (count - 1)) AND (i <> count) )
                  SEQ
                    so.write.string(keyboard,screen,",")
                TRUE
                  SKIP
          (temp[i] = op.brack)
            SEQ
              so.write.string(keyboard,screen,"(")
          (temp[i] = cl.brack)
            SEQ
              so.write.string(keyboard,screen,")")
              IF
```

```
            ((temp[i + 1] <> cl.sq.brack) AND (temp[i + 1] <> cl.brack) AND (temp[i +
1] <> split) AND (i <> (count - 1)))
              so.write.string(keyboard,screen,",")
            TRUE
              SKIP
          (temp[i] = split)
            so.write.string(keyboard,screen,"|")
          (temp[i] = empty)
            SEQ
              so.write.string(keyboard,screen,"[]")
          (temp[i] > lis) AND (temp[i] < struct)
            print.functor(temp[i])
          (temp[i] > shar.var) AND (temp[i] < ground.term)
            SEQ
              print.ground.term(temp[i])
              IF
                ((temp[i + 1] <> cl.sq.brack) AND (temp[i + 1] <> cl.brack) AND (temp[i +
1] <> split) AND (i <> (count - 1)))
                  so.write.string(keyboard,screen,",")
                TRUE
                  SKIP

          (temp[i] > struct) AND (temp[i] < shar.var)
            SEQ
              print.shar.var(temp[i])
          TRUE
            SKIP

  :

  -- ********************************************************************
  -- This function replaces those lists on the queue with their components. It is called by
  -- function print.list. It places the formated list in the array temp.
  -- ********************************************************************

  PROC list.repl([400]INT temp,INT end.store.ctr,count1,count,[400]INT queue,INT
count3,process.num,[]INT store.process,store.va,store.envir )
    [400]INT tem2:
    INT num.terms,count2,k,found,end:
    SEQ
      count2 := count1
      IF
        (list[temp[count1]][list[temp[count1]][0] ] = split)
          num.terms := list[temp[count1]][0] + 1
        TRUE
```

```
        num.terms := list[temp[count1]][0]
      [tem2 FROM 0 FOR count] := [temp FROM 0 FOR count]
      SEQ i = 1 FOR num.terms
        SEQ
          IF
            (list[tem2[count1]][i] = empty) AND (list[tem2[count1]][i - 1] = split)
              SKIP
            (list[tem2[count1]][i] = split) AND (list[tem2[count1]][i + 1] = empty)
              SKIP
            (list[tem2[count1]][i] < lis) AND (list[tem2[count1]][i - 1] <> split)
              SEQ
                temp[count2] := op.sq.brack
                temp[count2 +1] := list[tem2[count1]][i]
                temp[count2 + 2] := cl.sq.brack
                count2 := count2 +  3
                queue[count3] := list[tem2[count1]][i]
                count3 := count3 + 1
                queue[count3] := end.queue
            (list[tem2[count1]][i] < lis)
              SEQ
                temp[count2] := list[tem2[count1]][i]
                count2 := count2 + 1
                queue[count3] := list[tem2[count1]][i]
                count3 := count3 + 1
                queue[count3] := end.queue

            (list[tem2[count1]][i] > lis) AND (list[tem2[count1]][i] < struct)

              SEQ
                temp[count2] := list[tem2[count1]][i]
                temp[count2 + 1] := op.brack
                temp[count2 + 2] := func.list[(list[tem2[count1]][i] - lis)]
                temp[count2 + 3] := cl.brack
                count2 := count2 + 4
                queue[count3] := list[tem2[count1]][i]
                count3 := count3 + 1
                queue[count3] := end.queue
            ((list[tem2[count1]][i] > struct) AND (list[tem2[count1]][i] < shar.var)) OR
((list[tem2[count1]][i] > ground.term) AND (list[tem2[count1]][i] < non.shar.var))
              SEQ
                IF
                  (process.num = 1)
                    SEQ
                      end := end.store.ctr
                      k := 0
```

```
  TRUE
    SEQ
      end := store.process[process.num] + 1
      k := store.process[process.num - 1] + 1
found := 0
WHILE (found <> 1 ) AND (k <> end)
  SEQ
    check.var(store.va[k],list[tem2[count1]][i],found)
    IF
      (found = 1 )
        SKIP
      TRUE
        k := k + 1
IF
  (found = 1)
    SEQ
      IF
        (store.envir[k] < lis) AND (list[tem2[count1]][i   1] <> split)

          SEQ
            temp[count2] := op.sq.brack
            temp[count2 +1] := store.envir[k]
            temp[count2 + 2] := cl.sq.brack
            count2 := count2 +  3
            queue[count3] := store.envir[k]
            count3 := count3 + 1
            queue[count3] := end.queue
        (store.envir[k] < lis)
          SEQ
            temp[count2] := store.envir[k]
            count2 := count2 + 1
            queue[count3] := store.envir[k]
            count3 := count3 + 1
            queue[count3] := end.queue

        (store.envir[k] < struct) AND (store.envir[k] > lis)
          SEQ
            temp[count2] := store.envir[k]
            temp[count2 + 1] := op.brack
            temp[count2 + 2] := func.list[(store.envir[k] - lis)]
            temp[count2 + 3] := cl.brack
            count2 := count2 + 4
            queue[count3] := store.envir[k]
            count3 := count3 + 1
            queue[count3] := end.queue
```

```
                TRUE
                  SEQ
                    temp[count2] := store.envir[k]
                    count2 := count2 + 1
              TRUE
                SEQ
                  temp[count2] := uninstant
                  count2 := count2 + 1
          (list[tem2[count1]][i] = split) AND (list[tem2[count1]][i + 1] < lis)
            SKIP
          (list[tem2[count1]][i] = split) AND (((list[tem2[count1]][i + 1] < shar.var ) AND
(list[tem2[count1]][i + 1] > struct)) OR ((list[tem2[count1]][i +1] > ground.term) AND
(list[tem2[count1]][i + 1] < non.shar.var)))
            SEQ
              IF
                (process.num = 1)
                  SEQ
                    k := 0
                    end := end.store.ctr
                TRUE
                  SEQ
                    k := store.process[process.num - 1] + 1
                    end := store.process[process.num] + 1
              found := 0
              WHILE (found <> 1) AND (k <> end)
                SEQ
                  check.var(list[tem2[count1]][i + 1],store.va[k],found)
                  IF
                    (found = 1)
                      SKIP
                    TRUE
                      k := k + 1
              IF
                (found = 1)
                  SEQ
                    IF
                      (store.envir[k] < lis)
                        SKIP
                      TRUE
                        SEQ
                          temp[count2] := list[tem2[count1]][i]
                          count2 := count2 + 1
                TRUE
                  SEQ
                    temp[count2] := list[tem2[count1]][i]
```

```
            count2 := count2 + 1
        TRUE
          SEQ
            temp[count2] := list[tem2[count1]][i]
            count2 := count2 + 1
    [temp FROM count2 FOR (count - (count1 + 1))] := [tem2 FROM (count1 + 1)
FOR (count - (count1 + 1))]
    count := count + (count2 - (count1 + 1))
  :


  -- ***************************************************************
  -- This procedure is used to print out a list. A list is stored as integers in an array. To
  -- print out the list, the appropriate parenthesis, commas and spaces have to be
  -- inserted. The integers have to be replaced by their proper value. If the list contains
  -- nested lists or structures, then their components have to be printed out when the
  -- original list is being printed out. This procedure performs these operations. It places
  -- the parenthesis and commas. In case of nested lists, it places the parenthesis but dose
  -- not replace the list with its components. The list to be replaced is put in a queue and
  -- at the end of this procedure another procedure list.repl is called to replace the nested
  -- lists with their components. If the component of the list is a variable, it is replaced
  -- by its binding in the stored binding environment. The newly formated list is placed
  -- in another integer array. A function called print.val is called to print out the contents
  -- of this array.
  -- ***************************************************************

  PROC print.list(INT env,end.store.ctr,process.num,[]INT
store.process,store.va,store.envir )
    [400] INT temp:
    [400] INT queue:
    INT end,found,k,count,num.terms,count1,count3,count4:
    SEQ
      count := 0
      count3 := 0
      count4 := 0
      queue[count4] := end.queue
      IF
        (list[env][list[env][0] ] = split)
          SEQ
            num.terms := list[env][0] + 1
        TRUE
          SEQ
            num.terms := list[env][0]
      SEQ i = 1 FOR num.terms
        SEQ
```

```
          IF
           (list[env][i] < lis) AND (list[env][i-1] <> split)
             SEQ
              temp[count] := op.sq.brack
              temp[count + 1] := list[env][i]
              temp[count + 2] := cl.sq.brack
              count := count + 3
              queue[count3] := list[env][i]
              count3 := count3 + 1
              queue[count3] := end.queue
           (list[env][i] < lis)
             SEQ
              temp[count] := list[env][i]
              count := count + 1
              queue[count3] := list[env][i]
              count3 := count3 + 1
              queue[count3] := end.queue
           (list[env][i] > lis) AND (list[env][i] < struct)
             SEQ
              temp[count] := list[env][i]
              temp[count + 1] := op.brack
              temp[count + 2] := func.list[(list[env][i] - lis)]
              temp[count + 3] := cl.brack
              count := count + 4
              queue[count3] := list[env][i]
              count3 := count3 + 1
              queue[count3] := end.queue
           ((list[env][i] < shar.var) AND (list[env][i] > struct))  OR ((list[env][i] >
ground.term) AND (list[env][i] < non.shar.var))
             SEQ
              IF
               (process.num = 1)
                 SEQ
                  end := end.store.ctr
                  k := 0
               TRUE
                 SEQ
                  k := store.process[process.num   1] + 1
                  end := store.process[process.num] + 1
              found := 0
              WHILE(found <> 1) AND (k <> end)
               SEQ
                check.var(list[env][i],store.va[k],found)
                IF
                 (found = 1 )
```

```
          SKIP
        TRUE
          k := k + 1
    IF
     (found = 1)
      SEQ
       IF
        (store.envir[k] < lis) AND ((list[env][i - 1]) <> split)
          SEQ
            temp[count] := op.sq.brack
            temp[count + 1] := store.envir[k]
            temp[count + 2] := cl.sq.brack
            count := count + 3
            queue[count3] := store.envir[k]
            count3 := count3 + 1
            queue[count3] := end.queue
        (store.envir[k] < lis)
          SEQ
            temp[count] := store.envir[k]
            count := count + 1
            queue[count3] := store.envir[k]
            count3 := count3 + 1
            queue[count3] := end.queue

        (store.envir[k] > lis) AND (store.envir[k] < struct)
          SEQ
            temp[count] := store.envir[k]
            temp[count + 1] := op.brack
            temp[count + 2] := func.list[(store.envir[k] - lis)]
            temp[count + 3] := cl.brack
            count := count + 4
            queue[count3] := store.envir[k]
            count3 := count3 + 1
            queue[count3] := end.queue
        TRUE
          SEQ
            temp[count] := store.envir[k]
            count := count + 1
    TRUE
      SEQ
       temp[count] := uninstant
       count := count + 1

(list[env][i] = empty) AND (list[env][i-1] = split)
  SKIP
```

```
(list[env][i] = split) AND (list[env][i+1] = empty)
  SKIP
(list[env][i] = split) AND (list[env][i+1] < lis)
  SKIP
(list[env][i] = split) AND (((list[env][i + 1] < shar.var) AND (list[env][i + 1] >
struct)) OR ((list[env][i + 1] > ground.term) AND (list[env][i +1] < non.shar.var)))
  SEQ
   IF
    (process.num = 1)
     SEQ
      k := 0
      end := end.store.ctr
    TRUE
     SEQ
      k := store.process[process.num   1] + 1
      end := store.process[process.num] + 1
   found := 0
   WHILE(k <> end) AND (found <> 1)
    SEQ
     check.var(list[env][i +1],store.va[k],found)
     IF
      (found = 1)
       SKIP
      TRUE
       k := k + 1
   IF
    (found = 1)

     SEQ
      IF
       (store.envir[k] < lis)
        SKIP
       TRUE
        SEQ
         temp[count] := list[env][i]
         count := count + 1
     TRUE
      SEQ
       temp[count] := list[env][i]
       count := count + 1

  TRUE
   SEQ
    temp[count] := list[env][i]
    count := count + 1
```

```
      count1 := 0
      WHILE (queue[count4] <> end.queue)
        SEQ
          IF
            (temp[count1] < lis)
              SEQ
                count4 := count4 + 1

list.repl(temp,end.store.ctr,count1,count,queue,count3,process.num,store.process,store.va
,store.envir)
            TRUE
              count1 := count1 + 1
      print.val(temp,count)
  :

  PROC print.var(INT var.no)
    INT count1,k:
    [40]BYTE temp:
    SEQ
      count1 := 0
      IF
        ((var.no - struct) <> 1)
          SEQ
            SEQ k = 1 FOR ((var.no - struct) - 1)
              count1 := s.var[k] + count1
        TRUE
          SKIP
      [temp FROM 0 FOR s.var[(var.no - struct)]] := [shar.vars FROM count1 FOR
s.var[(var.no - struct)]]
      SEQ k = 0 FOR s.var[(var.no - struct)]
        so.write.char(keyboard,screen,temp[k])
  :

  -- ********************************************************************
  -- This procedure is used to print out the binding of a particular variable. If the binding
  -- is a list, the procedure prints out the opening parenthesis and calls the function
  -- print.list to print out the components of the list. It then prints out the closing
  -- parenthesis. If the binding is a structure the procedure prints out the functor and then
  -- calls the procedure print.list to print out the components of the structure. For any
  -- other term, the appropriate procedure is called to print out the value.
  -- ********************************************************************

  PROC print.environ(INT end.store.ctr,k,process.num,[]INT
store.process,store.va,store.envir )
    INT envir,env,p,found,temp,end :
```

```
    SEQ
     env := store.envir[k]
     IF
      (env < lis)
        SEQ
         IF
          (list[env][1] = split)
            SEQ
             IF
              (list[env][2] < lis)
                SEQ
                 so.write.string(keyboard,screen,"[")
                 temp := list[env][2]

print.list(temp,end.store.ctr,process.num,store.process,store.va,store.envir )
                 so.write.string(keyboard,screen,"]")
              (list[env][2] > lis) AND (list[env][2] < struct)
                SEQ
                 temp := list[env][2]
                 print.functor(temp)
                 so.write.string(keyboard,screen,"(")
                 envir := func.list[list[env][2] - lis]
                 print.list(envir
,end.store.ctr,process.num,store.process,store.va,store.envir )
                 so.write.string(keyboard,screen,")")
              ((list[env][2] < shar.var) AND (list[env][2] > struct)) OR ((list[env][2] >
ground.term) AND (list[env][2] < non.shar.var))
                SEQ
                 IF
                  (process.num = 1)
                    SEQ
                     end := end.store.ctr
                     p := 0
                  TRUE
                    SEQ
                     end := store.process[process.num] + 1
                     p := store.process[process.num   1] + 1
                 found := 0
                 WHILE(found <> 1) AND (p <> end)
                   SEQ
                    check.var(store.va[p],list[env][2],found)
                    IF
                     (found = 1)
                              SKIP
                     TRUE
```

```
                                p := p + 1
                        IF
                         (found = 1)
                           SEQ
                            IF
                             (store.envir[p] < lis)
                               SEQ
                                so.write.string(keyboard,screen,"[")
                                temp := store.envir[p]

print.list(temp,end.store.ctr,process.num,store.process,store.va,store.envir )
                                so.write.string(keyboard,screen,"]")
                             (store.envir[p]  > lis) AND (store.envir[p] < struct)
                               SEQ
                                print.functor(store.envir[p])
                                so.write.string(keyboard,screen,"(")
                                envir := func.list[store.envir[p]    lis]
                                print.list(envir
,end.store.ctr,process.num,store.process,store.va,store.envir )
                                so.write.string(keyboard,screen,")")
                             (store.envir[p] > struct) AND (store.envir[p] < ground.term)
                               SEQ
                                        print.ground.term(store.envir[p])
                             TRUE
                               SKIP
                          TRUE
                           so.write.string(keyboard,screen,"_0")

                    (list[env][2] > shar.var) AND (list[env][2] < ground.term)
                      SEQ
                       temp := list[env][2]
                       print.ground.term(temp )
                     TRUE
                      SKIP

                  TRUE
                   SEQ
                    so.write.string(keyboard,screen,"[")
                    print.list(env,end.store.ctr,process.num,store.process,store.va,store.envir )
                    so.write.string(keyboard,screen,"]")
               (env > lis) AND (env < struct)
                 SEQ
                  print.functor(env)
                  so.write.string(keyboard,screen,"(")
                  envir := func.list[env - lis]
```

```
          print.list(envir ,end.store.ctr,process.num,store.process,store.va,store.envir )
          so.write.string(keyboard,screen,")")
        (env > shar.var) AND (env < ground.term)
          SEQ
            print.ground.term(env)
        (env = uninstant)
          so.write.string(keyboard,screen,"_0")
        (env = empty)
          so.write.string(keyboard,screen,"[]")
        (env < shar.var) AND (env > struct)
          print.var(env)
        TRUE
          SKIP
:
PROC check.print.queue(INT var,[]INT print.queue,INT find)
  INT found,k:
  SEQ
    k := 0
    WHILE (print.queue[k] <> end.queue)
      SEQ
        found := 0
        check.var(var,print.queue[k],found)
        IF
          (found = 1)
            SEQ
              find := 1
          TRUE
            SKIP
        k := k + 1
:


  PROC print.list.queue.check(INT end.store.ctr,pro.num,[]INT sto.proc,
list.queue,store.envir,store.va,INT queue.ptr,[]INT print.queue,INT queue.pt)
    INT end,ptr,num.list,num.terms,k,found,find:
    BOOL got:
    SEQ
      ptr := 0
      WHILE (list.queue[ptr] <> end.queue)
        SEQ
          IF
            (list[list.queue[ptr]][list[list.queue[ptr]][0]]  = split)
              num.terms := list[list.queue[ptr]][0] + 1
            TRUE
              num.terms := list[list.queue[ptr]][0]
```

```
SEQ i = 1 FOR num.terms
 IF
  (list[list.queue[ptr]][i] < lis)
    SEQ
     list.queue[queue.ptr] := list[list.queue[ptr]][i]
     list.queue[queue.ptr + 1] := end.queue
     queue.ptr := queue.ptr + 1
  (list[list.queue[ptr]][i] < struct) AND (list[list.queue[ptr]][i] > lis)
    SEQ
     num.list := list[list.queue[ptr]][i] - lis
     list.queue[queue.ptr] := func.list[num.list]
     list.queue[queue.ptr + 1] := end.queue
     queue.ptr := queue.ptr + 1
  ((list[list.queue[ptr]][i] < shar.var)  AND (list[list.queue[ptr]][i] > struct))
    SEQ
     IF
      (pro.num = 1)
        SEQ
         end := end.store.ctr
         k := 0
      TRUE
        SEQ
         end := sto.proc[pro.num] + 1
         k := sto.proc[pro.num - 1] + 1
     got := FALSE
     found := 0
     WHILE (k <> end) AND (got <> TRUE)
      SEQ
       check.var(list[list.queue[ptr]][i],store.va[k],found)
       IF
        (found = 1)
          SEQ
           IF
            (store.envir[k] < lis)
              SEQ
               list.queue[queue.ptr] := store.envir[k]
               list.queue[queue.ptr + 1] := end.queue
               queue.ptr := queue.ptr + 1
            (store.envir[k] > lis) AND (store.envir[k] < struct)
              SEQ
               num.list := store.envir[k] - lis
               list.queue[queue.ptr] := func.list[num.list]
               list.queue[queue.ptr + 1] := end.queue
               queue.ptr := queue.ptr + 1
            TRUE
```

```
                    SKIP
                  got := TRUE
                  find := 0
                  check.print.queue(list[list.queue[ptr]][i],print.queue,find)
                  IF
                   (find = 0)
                    SEQ
                     print.queue[queue.pt] := list[list.queue[ptr]][i]
                     print.queue[queue.pt + 1] := end.queue
                     queue.pt := queue.pt + 1
                     so.write.nl(keyboard,screen)
                     print.shar.var(list[list.queue[ptr]][i])
                     so.write.string(keyboard,screen," = ")
                     print.environ(end.store.ctr,k,pro.num,sto.proc,store.va,store.envir

                   TRUE
                    SKIP
                  k := end
                TRUE
                 k := k + 1
          ((list[list.queue[ptr]][i] < non.shar.var)  AND (list[list.queue[ptr]][i] >
ground.term))
           SEQ
            IF
             (pro.num = 1)
              SEQ
               end := end.store.ctr
               k := 0
             TRUE
              SEQ
               end := sto.proc[pro.num] + 1
               k := sto.proc[pro.num - 1] + 1
            found := 0
            WHILE(k <> end ) AND (found <> 1)
             SEQ
              IF
               (store.va[k] = list[list.queue[ptr]][i])
                SEQ
                 found := 1
               TRUE
                k := k + 1
            IF
             (found = 0) AND (list[list.queue[ptr]][i - 1] = split)
              SEQ
               list[list.queue[ptr]][i] := 0
```

```
                list[list.queue[ptr]][i - 1] := 0
                list[list.queue[ptr]][0] := list[list.queue[ptr]][0] - 1
              (found = 0)
                SKIP

              TRUE
                IF
                  (store.envir[k] < lis)
                    SEQ
                      list.queue[queue.ptr] := store.envir[k]
                      list.queue[queue.ptr + 1] := end.queue
                      queue.ptr := queue.ptr + 1
                  (store.envir[k] < struct) AND (store.envir[k] > lis)
                    SEQ
                      num.list := store.envir[k] - lis
                      list.queue[queue.ptr] := func.list[num.list]
                      list.queue[queue.ptr+ 1] := end.queue
                      queue.ptr := queue.ptr + 1
                  TRUE
                    SKIP
            TRUE
              SKIP
        ptr := ptr + 1
:

  PROC list.queue.check(INT end.store.ctr,pro.num,[]INT sto.proc,
list.queue,store.envir,store.va,INT queue.ptr,[]INT print.queue,INT queue.pt)
    INT ptr,num.list,num.terms,k,found,find:
    BOOL got:
    SEQ
      ptr := 0
      WHILE (list.queue[ptr] <> end.queue)
        SEQ
          IF
            (list[list.queue[ptr]][list[list.queue[ptr]][0]]  = split)
              num.terms := list[list.queue[ptr]][0] + 1
            TRUE
              num.terms := list[list.queue[ptr]][0]
          SEQ i = 1 FOR num.terms
            IF
              (list[list.queue[ptr]][i] < lis)
                SEQ
                  list.queue[queue.ptr] := list[list.queue[ptr]][i]
                  list.queue[queue.ptr + 1] := end.queue
                  queue.ptr := queue.ptr + 1
```

```
(list[list.queue[ptr]][i] < struct) AND (list[list.queue[ptr]][i] > lis)
  SEQ
   num.list := list[list.queue[ptr]][i] - lis
   list.queue[queue.ptr] := func.list[num.list]
   list.queue[queue.ptr + 1] := end.queue
   queue.ptr := queue.ptr + 1
((list[list.queue[ptr]][i] < shar.var)  AND (list[list.queue[ptr]][i] > struct))
  SEQ
   IF
    (pro.num = 1)
     k := 0
    TRUE
     SEQ
      k := sto.proc[pro.num - 1] + 1
   got := FALSE
   found := 0
   WHILE (k <> end.store.ctr) AND (got <> TRUE)
    SEQ
     check.var(list[list.queue[ptr]][i],store.va[k],found)
     IF
      (found = 1)
       SEQ
        IF
         (store.envir[k] < lis)
          SEQ
           list.queue[queue.ptr] := store.envir[k]
           list.queue[queue.ptr + 1] := end.queue
           queue.ptr := queue.ptr + 1
         (store.envir[k] > lis) AND (store.envir[k] < struct)
          SEQ
           num.list := store.envir[k] - lis
           list.queue[queue.ptr] := func.list[num.list]
           list.queue[queue.ptr + 1] := end.queue
           queue.ptr := queue.ptr + 1
         TRUE
          SKIP
        k := end.store.ctr
        got := TRUE
        find := 0
       check.print.queue(list[list.queue[ptr]][i],print.queue,find)
        IF
         (find = 0)
          SEQ
           print.queue[queue.pt] := list[list.queue[ptr]][i]
           print.queue[queue.pt + 1] := end.queue
```

```
                    queue.pt := queue.pt + 1
                  TRUE
                    SKIP
                TRUE
                  k := k + 1
              IF
                (got = FALSE)
                  SEQ
                    find := 0
                    check.print.queue(list[list.queue[ptr]][i],print.queue,find)
                    IF
                      (find = 0)
                        SEQ
                          print.queue[queue.pt]:= list[list.queue[ptr]][i]
                          print.queue[queue.pt + 1] := end.queue
                          queue.pt := queue.pt + 1
                          store.va[end.store.ctr] := list[list.queue[ptr]][i]
                          store.envir[end.store.ctr] := uninstant
                          end.store.ctr := end.store.ctr + 1

                      TRUE
                        SKIP
                  TRUE
                    SKIP
          ((list[list.queue[ptr]][i] < non.shar.var)  AND (list[list.queue[ptr]][i] >
ground.term))
            SEQ
              IF
                (pro.num = 1)
                  k := 0
                TRUE
                  k := sto.proc[pro.num   1] + 1
              found := 0
              WHILE(k <> end.store.ctr ) AND (found <> 1)
                SEQ
                  IF
                    (store.va[k] = list[list.queue[ptr]][i])
                      SEQ
                        list[list.queue[ptr]][i] := store.envir[k]
                        found := 1
                    TRUE
                      k := k + 1
              IF
                (found = 0) AND (list[list.queue[ptr]][i - 1] = split)
                  SEQ
```

```
                list[list.queue[ptr]][i] := 0
                list[list.queue[ptr]][i - 1] := 0
                list[list.queue[ptr]][0] := list[list.queue[ptr]][0] - 1
              (found = 0)
                SKIP

              TRUE
                IF
                  (store.envir[k] < lis)
                    SEQ
                      list.queue[queue.ptr] := store.envir[k]
                      list.queue[queue.ptr + 1] := end.queue
                      queue.ptr := queue.ptr + 1
                  (store.envir[k] < struct) AND (store.envir[k] > lis)
                    SEQ
                      num.list := store.envir[k] - lis
                      list.queue[queue.ptr] := func.list[num.list]
                      list.queue[queue.ptr+ 1] := end.queue
                      queue.ptr := queue.ptr + 1
                  TRUE
                    SKIP
            TRUE
              SKIP
          ptr := ptr + 1
  :

  -- ************************************************************************
  -- This procedure is used to instantiate the components of nested lists or structures.
  -- ************************************************************************

  PROC print.list.queue.check2(INT end.store.ctr,pro.num,[]INT sto.proc,
list.queue,store.envir,store.va,INT queue.ptr,[]INT print.queue,INT queue.pt)
    INT end,p,find,found,ptr,num.lis,num.list,num.terms,k:
    BOOL got,get:
    SEQ
      ptr := 0
      WHILE(list.queue[ptr] <> end.queue)
        SEQ
          num.lis := list.queue[ptr]
          IF
            (list[num.lis][list[num.lis][0]] = split)
              num.terms := list[num.lis][0] + 1
            TRUE
              num.terms := list[num.lis][0]
          SEQ j = 1 FOR num.terms
```

```
IF
 (list[num.lis][j] < lis)
   SEQ
    list.queue[queue.ptr] := list[num.lis][j]
    list.queue[queue.ptr + 1] := end.queue
    queue.ptr := queue.ptr + 1
 ( list[num.lis][j] > lis) AND (list[num.lis][j] <  struct)
   SEQ
    num.list := list[num.lis][j] - lis
    list.queue[queue.ptr] := func.list[num.list]
    list.queue[queue.ptr + 1] := end.queue
    queue.ptr := queue.ptr + 1
 (list[num.lis][j] > struct) AND (list[num.lis][j] < shar.var)
   SEQ
    IF
     (pro.num = 1)
       SEQ
        end := end.store.ctr
        k := 0
     TRUE
       SEQ
        end := sto.proc[pro.num] + 1
        k := sto.proc[pro.num   1] + 1
    got := FALSE
    found := 0
    WHILE (k <> end)  AND (got <> TRUE)
      SEQ
       check.var(list[num.lis][j],store.va[k],found)
       IF
        (found = 1)
          SEQ
           find := 0
           check.print.queue(list[num.lis][j],print.queue,find)
           IF
            (find = 0)
              SEQ
               print.queue[queue.pt] := list[num.lis][j]
               print.queue[queue.pt + 1] :=end.queue
               queue.pt := queue.pt + 1
               IF
                (store.envir[k] > struct) AND (store.envir[k] < shar.var)
                  SEQ
                   IF
                    (pro.num = 1)
                      p := 0
```

```
                                TRUE
                                  p := sto.proc[pro.num   1] + 1
                                        get := FALSE
                                        found := 0
                                WHILE (p <> end)
                                  SEQ
                                   check.var(store.va[p],store.envir[k],found)
                                   IF
                                    (found = 1)
                                      SEQ
                                       so.write.nl(keyboard,screen)
                                       print.shar.var(list[num.lis][j])
                                       so.write.string(keyboard,screen," = ")

print.environ(end.store.ctr,p,pro.num,sto.proc,store.va,store.envir )
                                        p := end
                                        get := TRUE
                                      TRUE
                                        p := p + 1
                              TRUE
                               SEQ
                                so.write.nl(keyboard,screen)
                                print.shar.var(list[num.lis][j])
                                so.write.string(keyboard,screen," = ")

print.environ(end.store.ctr,k,pro.num,sto.proc,store.va,store.envir )
                         TRUE
                           SKIP
                        got := TRUE
                        k := end
                      TRUE
                        k := k + 1
                IF
                 (got <> TRUE)
                   SEQ
                    so.write.nl(keyboard,screen)
                    print.shar.var(list[num.lis][j])
                    so.write.string(keyboard,screen," = ")
                    so.write.string(keyboard,screen,"_0")
                 TRUE
                   SKIP
             TRUE
               SKIP
          ptr := ptr + 1
  :
```

```
  PROC list.queue.check2(INT end.store.ctr,pro.num,[]INT sto.proc,
list.queue,store.envir,store.va,INT queue.ptr,[]INT print.queue,INT queue.pt)
    INT p,find,found,ptr,num.lis,num.list,num.terms,k:
    BOOL got,get:
    SEQ
     ptr := 0
     WHILE(list.queue[ptr] <> end.queue)
      SEQ
       num.lis := list.queue[ptr]
       IF
        (list[num.lis][list[num.lis][0]] = split)
         num.terms := list[num.lis][0] + 1
        TRUE
         num.terms := list[num.lis][0]
       SEQ j = 1 FOR num.terms
        IF
         (list[num.lis][j] < lis)
           SEQ
            list.queue[queue.ptr] := list[num.lis][j]
            list.queue[queue.ptr + 1] := end.queue
            queue.ptr := queue.ptr + 1
         ( list[num.lis][j] > lis) AND (list[num.lis][j] <  struct)
           SEQ
            num.list := list[num.lis][j] - lis
            list.queue[queue.ptr] := func.list[num.list]
            list.queue[queue.ptr + 1] := end.queue
            queue.ptr := queue.ptr + 1
         (list[num.lis][j] > struct) AND (list[num.lis][j] < shar.var)
           SEQ
            IF
             (pro.num = 1)
               k := 0
             TRUE
               SEQ
                 k := sto.proc[pro.num - 1] + 1
            got := FALSE
            found := 0
            WHILE (k <> end.store.ctr)  AND (got <> TRUE)
             SEQ
              check.var(list[num.lis][j],store.va[k],found)
              IF
               (found = 1)
                 SEQ
                  find := 0
```

```
check.print.queue(list[num.lis][j],print.queue,find)
 IF
  (find = 0)
    SEQ
     print.queue[queue.pt] := list[num.lis][j]
     print.queue[queue.pt + 1] :=end.queue
     queue.pt := queue.pt + 1
     IF
      (store.envir[k] > struct) AND (store.envir[k] < shar.var)
        SEQ
         IF
          (pro.num = 1)
            p := 0
          TRUE
            p := sto.proc[pro.num - 1] + 1
                    get := FALSE
                    found := 0
         WHILE (p <> end.store.ctr)
          SEQ
           check.var(store.va[p],store.envir[k],found)
           IF
            (found = 1)
              SEQ
                p := end.store.ctr
                get := TRUE
            TRUE
              p := p + 1
         IF
          (get = FALSE)
            SEQ
             store.va[end.store.ctr] := store.va[k]
             store.envir[end.store.ctr] := uninstant
             end.store.ctr := end.store.ctr + 1
             store.envir[k] := uninstant
            TRUE
             SKIP
       TRUE
        SEQ
         SKIP
   TRUE
    SKIP
  got := TRUE
  k := end.store.ctr
TRUE
 k := k + 1
```

```
            IF
              (got = FALSE)
                SEQ
                  find := 0
                  check.print.queue(list[num.lis][j],print.queue,find)
                  IF
                    (find = 0)
                      SEQ
                        print.queue[queue.pt] := list[num.lis][j]
                        print.queue[queue.pt + 1] := end.queue
                        queue.pt := queue.pt + 1
                        store.va[end.store.ctr] := list[num.lis][j]
                        store.envir[end.store.ctr] := uninstant
                        end.store.ctr := end.store.ctr + 1
                    TRUE
                      SKIP
                TRUE
                  SKIP
          TRUE
            SKIP
        ptr := ptr + 1
  :

  -- ***************************************************************************
  -- This instruction receives a term and the stored binding environment. If the input
  -- term is a ground term the instruction does nothing but output the same term,
  -- otherwise, the instruction instantiates the variables occurring in the term according
  -- to their binding instances in the stored binding environment and then outputs the
  -- instantiated term. If the term is a list each component of the list has to be
  -- instantiated. If the list contains a nested list or a structure then their components will
  -- also have to be instantiated. Since Occam does not support recursion, these nested
  -- lists and structures are not instantiated at this point, but are put on a queue and
  -- another procedure called print.list.queue.check2 is called to instantiate them. If the
  -- component of the list is a variable, it is replaced by its binding from the binding
  -- envrionment. If the term to be instantiated is a variable, then its value from the
  -- binding environment is found. If it does not have a value then the variable is
  -- declared as uninstantiated. If it has a value, then the value from the binding
  -- environment is instantiated.If the binding of the variable is a list, then the
  -- components of the list are instantiated. To instantiate nested lists another procedure
  -- called print.list.queue.check is called. this procedure also prints the value of the
  -- variable instantiated.
  -- ***************************************************************************

  PROC print.instantiate(INT end.store.ctr,pro.num,[]INT
sto.proc,store.envir,store.var,INT term,[]INT print.queue,INT queue.pt)
```

```
INT end,queue.ptr,i,num.lis,temp,num.list,num.terms,k,found,p,find,queue.point:
[400] INT list.queue:
BOOL got,get:
SEQ
 IF
  (term < struct)
   SEQ
    queue.ptr := 0
    IF
     (term < lis)
      num.lis := term
     TRUE
      num.lis := func.list[term - lis]
    IF
     (list[num.lis][list[num.lis][0]] = split)
      num.terms := list[num.lis][0] + 1
     TRUE
      num.terms := list[num.lis][0]
    SEQ j = 1 FOR num.terms
     IF
      (list[num.lis][j] < lis)
       SEQ
        list.queue[queue.ptr] := list[num.lis][j]
        list.queue[queue.ptr + 1] := end.queue
        queue.ptr := queue.ptr + 1
      (list[num.lis][j] > lis) AND (list[num.lis][j] <  struct)
       SEQ
        num.list := list[num.lis][j] - lis
        list.queue[queue.ptr] := func.list[num.list]
        list.queue[queue.ptr + 1] := end.queue
        queue.ptr := queue.ptr + 1
      (list[num.lis][j] > struct) AND (list[num.lis][j] < shar.var)
       SEQ
        IF
         (pro.num = 1)
          SEQ
           k := 0
           end := end.store.ctr
         TRUE
          SEQ
           k := sto.proc[pro.num   1] + 1
           end := sto.proc[pro.num] + 1

        queue.point := queue.ptr
        got := FALSE
```

```
                found := 0
                WHILE (k <> end) AND (got <> TRUE)
                  SEQ
                   check.var(list[num.lis][j],store.var[k],found)
                    IF
                     (found = 1 )
                       SEQ
                        find := 0
                        check.print.queue(list[num.lis][j],print.queue,find)
                         IF
                          (find = 0)
                            SEQ
                             print.queue[queue.pt] := list[num.lis][j]
                             print.queue[queue.pt + 1] := end.queue
                             queue.pt := queue.pt + 1
                             IF
                              (store.envir[k] > struct) AND (store.envir[k] < shar.var)
                                SEQ
                                 IF
                                  pro.num = 1
                                    p := 0
                                  TRUE
                                    SEQ
                                      p := sto.proc[pro.num - 1] + 1
                                get := FALSE
                                found := 0
                                WHILE (p <> end) AND (get <> TRUE)
                                  SEQ
                                   check.var(store.var[p],store.envir[k],found)
                                   IF
                                    (found = 1)
                                      SEQ
                                       so.write.nl(keyboard,screen)
                                       print.shar.var(list[num.lis][j])
                                       so.write.string(keyboard,screen," = ")

print.environ(end.store.ctr,p,pro.num,sto.proc,store.var,store.envir )
                                        p := end
                                        get := TRUE
                                      TRUE
                                        p := p + 1
                             (store.envir[k] < lis)
                               SEQ
                                list.queue[queue.ptr] := store.envir[k]
                                list.queue[queue.ptr + 1] := end.queue
```

```
                    queue.ptr := queue.ptr + 1
                  (store.envir[k] < struct)
                    SEQ
                     num.list := store.envir[k] - lis
                     list.queue[queue.ptr] := func.list[num.list]
                     list.queue[queue.ptr] := end.queue
                     queue.ptr := queue.ptr + 1
                   TRUE
                    SEQ
                     so.write.nl(keyboard,screen)
                     print.shar.var(list[num.lis][j])
                     so.write.string(keyboard,screen," = ")

print.environ(end.store.ctr,k,pro.num,sto.proc,store.var,store.envir )
                  TRUE
                    SKIP
                 IF
                  (queue.ptr <> queue.point)
                    SEQ

print.list.queue.check2(end.store.ctr,pro.num,sto.proc,list.queue,store.envir,store.var,queu
e.ptr,print.queue,queue.pt)
                     so.write.nl(keyboard,screen)
                     print.shar.var(list[num.lis][j])
                     so.write.string(keyboard,screen," = ")

print.environ(end.store.ctr,k,pro.num,sto.proc,store.var,store.envir )
                  TRUE
                    SKIP
                 got := TRUE
                 k := end
               TRUE
                 k := k + 1
            IF
             (got <> TRUE)
               SEQ
                so.write.nl(keyboard,screen)
                print.shar.var(list[num.lis][j])
                so.write.string(keyboard,screen," = ")
                so.write.string(keyboard,screen,"_0")
             TRUE
               SKIP

          TRUE
            SKIP
```

```
          IF
            (queue.ptr <> 0)
              SEQ

print.list.queue.check2(end.store.ctr,pro.num,sto.proc,list.queue,store.envir,store.var,queu
e.ptr,print.queue,queue.pt)
            TRUE
              SKIP
      ((term > struct) AND (term < shar.var)) OR ((term >ground.term) AND (term<
non.shar.var))
          SEQ
            IF
              (pro.num = 1)
                SEQ
                  i := 0
                  end := end.store.ctr
              TRUE
                SEQ
                  i := sto.proc[pro.num - 1] + 1
                  end := sto.proc[pro.num] + 1
            got := FALSE
            WHILE  ( i <> end)  AND (got <> TRUE)
              SEQ
                IF
                  (store.var[i] = term)

                    got := TRUE
                  TRUE
                    i := i + 1
            IF
              (got <>  TRUE )
                SEQ
                  so.write.nl(keyboard,screen)
                  print.shar.var(term)
                  so.write.string(keyboard,screen," = ")
                  so.write.string(keyboard,screen," _0  ")
                  print.queue[queue.pt] := term
                  print.queue[queue.pt + 1] := end.queue
                  queue.pt := queue.pt + 1
              TRUE
                SEQ
                  queue.ptr := 0
                  IF
                    (store.envir[i] < lis) OR (store.envir[i] < struct)
                      SEQ
```

```
IF
 (store.envir[i] < lis)
  num.lis := store.envir[i]
 TRUE
  num.lis := func.list[store.envir[i] - lis]
temp := list[num.lis][0]
IF
 (list[num.lis][temp] = split)
  num.terms := temp + 1
 TRUE
  num.terms := temp
SEQ j = 1 FOR num.terms
 IF
  (list[num.lis][j]  < lis)
    SEQ
     list.queue[queue.ptr] := list[num.lis][j]
     list.queue[queue.ptr + 1] := end.queue
     queue.ptr := queue.ptr + 1
  (list[num.lis][j] > lis) AND (list[num.lis][j] < struct)
    SEQ
     num.list := list[num.lis][j] - lis
     list.queue[queue.ptr] := func.list[num.list]
     list.queue[queue.ptr + 1] := end.queue
     queue.ptr := queue.ptr + 1
  ((list[num.lis][j] > struct) AND (list[num.lis][j] < shar.var))
    SEQ
     IF
      pro.num = 1
        SEQ
         end := end.store.ctr
         k := 0
      TRUE
        SEQ
         end := sto.proc[pro.num] + 1
         k := sto.proc[pro.num - 1] + 1
     got := FALSE
     found := 0
     WHILE (k <> end) AND (got <> TRUE)
       SEQ
        check.var(list[num.lis][j],store.var[k],found)
        IF
         (found = 1)
           SEQ
            k := end
            got := TRUE
```

```
      TRUE
        k := k + 1
  IF
    (got = FALSE)
      SEQ
        find := 0
        check.print.queue(list[num.lis][j],print.queue,find)
        IF
          (find = 0)
            SEQ
              print.queue[queue.pt] := list[num.lis][j]
              print.queue[queue.pt + 1] := end.queue
              queue.pt := queue.pt + 1
              so.write.nl(keyboard,screen)
              print.shar.var(list[num.lis][j])
              so.write.string(keyboard,screen," = ")
              so.write.string(keyboard,screen," _0 ")
          TRUE
            SKIP
      TRUE
        SEQ
          IF
            (store.envir[k] < lis)
              SEQ
                list.queue[queue.ptr] := store.envir[k]
                list.queue[queue.ptr + 1] := end.queue
                queue.ptr := queue.ptr + 1
            (store.envir[k] > lis) AND (store.envir[k] < struct)
              SEQ
                num.list := store.envir[k] - lis
                list.queue[queue.ptr] := func.list[num.list]
                list.queue[queue.ptr + 1] := end.queue
                queue.ptr := queue.ptr + 1
            TRUE
                        SKIP
(list[num.lis][j] > ground.term) AND (list[num.lis][j] < non.shar.var)

  SEQ
    IF
      (pro.num = 1)
        SEQ
          end := end.store.ctr
          k := 0
      TRUE
        SEQ
```

```
        end := sto.proc[pro.num] + 1
        k := sto.proc[pro.num - 1] + 1
    found := 0
    WHILE(k <> end ) AND (found <> 1)
      SEQ
       IF
        (store.var[k] = list[num.lis][j])
          SEQ
           found := 1
        TRUE
          k := k + 1
    IF
     (found = 0) AND (list[num.lis][j-1] = split)
       SEQ
        list[num.lis][j - 1] := 0
        list[num.lis][j] := 0
        list[num.lis][0] := list[num.lis][0] - 1
     (found = 0)
       SKIP
     TRUE
       IF
        (store.envir[k] < lis)
          SEQ
           list.queue[queue.ptr] := store.envir[k]
           list.queue[queue.ptr + 1] := end.queue
           queue.ptr := queue.ptr + 1
        (store.envir[k] > lis) AND (store.envir[k] < struct)
          SEQ
           num.list := store.envir[k] - lis
           list.queue[queue.ptr] := func.list[num.list]
           list.queue[queue.ptr + 1] := end.queue
           queue.ptr := queue.ptr + 1
        TRUE
          SKIP
(list[num.lis][j] > struct) AND (list[num.lis][j] < shar.var)
  SEQ
   got := FALSE
   IF
    (pro.num = 1)
      SEQ
       end := end.store.ctr
       k := 0
    TRUE
      SEQ
       end := sto.proc[pro.num] + 1
```

```
                    k := sto.proc[pro.num   1] + 1
                  found := 0
                  WHILE (k <> end ) AND (got <> TRUE)
                    SEQ
                     check.var(list[num.lis][j],store.var[k],found)
                     IF
                      (found = 1)
                        SEQ
                         find := 0
                         check.print.queue(store.var[k],print.queue,find)
                         IF
                          (find = 0 )
                            SEQ
                             print.queue[queue.pt] := store.var[k]
                             print.queue[queue.pt + 1] := end.queue
                             queue.pt := queue.pt + 1
                             print.shar.var(store.var[k])
                             so.write.string(keyboard,screen," = ")

print.environ(end.store.ctr,k,pro.num,sto.proc,store.var,store.envir )
                                TRUE
                                 SKIP
                               k := end
                               got := TRUE
                             TRUE
                              k := k + 1

                      TRUE
                        SKIP
                    IF
                     (queue.ptr <> 0)

print.list.queue.check(end.store.ctr,pro.num,sto.proc,list.queue,store.envir,store.var,queue
.ptr,print.queue,queue.pt)
                      TRUE
                       SKIP
                  (store.envir[i] > struct) AND (store.envir[i] < shar.var)
                    SEQ
                     got := FALSE
                     IF
                      (pro.num = 1)
                        SEQ
                          k := 0
                         end := end.store.ctr
                      TRUE
```

```
                SEQ
                  k := sto.proc[pro.num - 1] + 1
                  end := sto.proc[pro.num] + 1
              found := 0
              WHILE (k <> end)
                SEQ
                 check.var(store.var[k],store.envir[i],found)
                 IF
                  (found = 1)
                    SEQ
                     store.envir[i] := store.envir[k]
                     got := TRUE
                     k := end
                  TRUE
                    k := k +1
              IF
                (got = FALSE)
                  SEQ
                   store.envir[i] := uninstant
                TRUE
                  SKIP

            TRUE
              SEQ
                SKIP
          found := 0
          check.print.queue(term,print.queue,found)
          IF
            (found <> 1)
              SEQ
                print.queue[queue.pt] := term
                print.queue[queue.pt + 1] := end.queue
                queue.pt := queue.pt + 1
                so.write.nl(keyboard,screen)
                print.shar.var(term)
                so.write.string(keyboard,screen," = ")
                print.environ(end.store.ctr,i,pro.num,sto.proc,store.var,store.envir)
            TRUE
              SKIP
    TRUE
      SKIP
:

-- ***************************************************************
-- This procedure is used ot instantiate lists or variables, but it does not print their
```

```
-- values.
-- ********************************************************************

  PROC instantiate(INT end.store.ctr,pro.num,[]INT sto.proc,store.envir,store.var,INT
term,[]INT print.queue,INT queue.pt)
    INT queue.ptr,i,num.lis,temp,num.list,num.terms,k,found,p,find,queue.point:
    [400] INT list.queue:
    BOOL got,get:
    SEQ
     IF
      (term < struct)
       SEQ
        queue.ptr := 0
        IF
         (term < lis)
          num.lis := term
         TRUE
          num.lis := func.list[term - lis]
        IF
         (list[num.lis][list[num.lis][0]] = split)
          num.terms := list[num.lis][0] + 1
         TRUE
          num.terms := list[num.lis][0]
        SEQ j = 1 FOR num.terms
         IF
          (list[num.lis][j] < lis)
            SEQ
             list.queue[queue.ptr] := list[num.lis][j]
             list.queue[queue.ptr + 1] := end.queue
             queue.ptr := queue.ptr + 1
          (list[num.lis][j] > lis) AND (list[num.lis][j] <  struct)
            SEQ
             num.list := list[num.lis][j] - lis
             list.queue[queue.ptr] := func.list[num.list]
             list.queue[queue.ptr + 1] := end.queue
             queue.ptr := queue.ptr + 1
          (list[num.lis][j] > struct) AND (list[num.lis][j] < shar.var)
            SEQ
             IF
              (pro.num = 1)
               k := 0
              TRUE
               SEQ
                k := sto.proc[pro.num - 1] + 1
```

```
queue.point := queue.ptr
got := FALSE
found := 0
WHILE (k <> end.store.ctr) AND (got <> TRUE)
 SEQ
  check.var(list[num.lis][j],store.var[k],found)
  IF
   (found = 1 )
    SEQ
     find := 0
     check.print.queue(list[num.lis][j],print.queue,find)
     IF
      (find = 0)
       SEQ
        print.queue[queue.pt] := list[num.lis][j]
        print.queue[queue.pt + 1] := end.queue
        queue.pt := queue.pt + 1
        IF
         (store.envir[k] > struct) AND (store.envir[k] < shar.var)
          SEQ
           IF
            pro.num = 1
             p := 0
            TRUE
             SEQ
              p := sto.proc[pro.num - 1] + 1
           get := FALSE
           found := 0
           WHILE (p <> end.store.ctr) AND (get <> TRUE)
            SEQ
             check.var(store.var[p],store.envir[k],found)
             IF
              (found = 1)
               SEQ
                p := end.store.ctr
                get := TRUE
              TRUE
               p := p + 1
           IF
            (get = FALSE)
             SEQ
              store.var[end.store.ctr] := store.var[k]
              store.envir[end.store.ctr] := uninstant
              end.store.ctr := end.store.ctr + 1
              store.envir[k] := uninstant
```

```
                    TRUE
                      SKIP
                  (store.envir[k] < lis)
                    SEQ
                     list.queue[queue.ptr] := store.envir[k]
                     list.queue[queue.ptr + 1] := end.queue
                     queue.ptr := queue.ptr + 1
                  (store.envir[k] < struct)
                    SEQ
                     num.list := store.envir[k]   lis
                     list.queue[queue.ptr] := func.list[num.list]
                     list.queue[queue.ptr] := end.queue
                     queue.ptr := queue.ptr + 1
                  TRUE
                    SEQ
                     SKIP
               TRUE
                 SKIP
              IF
               (queue.ptr <> queue.point)
                 SEQ

list.queue.check2(end.store.ctr,pro.num,sto.proc,list.queue,store.envir,store.var,queue.ptr,
print.queue,queue.pt)
                    TRUE
                      SKIP
                   got := TRUE
                   k := end.store.ctr
                TRUE
                  k := k + 1
           IF
            (got = FALSE)
              SEQ
               find := 0
               check.print.queue(list[num.lis][j],print.queue,find)
               IF
                (find = 0 )
                  SEQ
                   print.queue[queue.pt] := list[num.lis][j]
                   print.queue[queue.pt + 1] := end.queue
                   queue.pt := queue.pt + 1
                   store.var[end.store.ctr] := list[num.lis][j]
                   store.envir[end.store.ctr] := uninstant
                   end.store.ctr := end.store.ctr + 1
                TRUE
```

```
                SKIP
            TRUE
              SKIP
          TRUE
            SKIP
        IF
          (queue.ptr <> 0)
            SEQ

list.queue.check2(end.store.ctr,pro.num,sto.proc,list.queue,store.envir,store.var,queue.ptr,
print.queue,queue.pt)
          TRUE
            SKIP
      ((term > struct) AND (term < shar.var)) OR ((term >ground.term) AND (term<
non.shar.var))
        SEQ
          IF
            (pro.num = 1)
              i := 0
            TRUE
              SEQ
                i := sto.proc[pro.num - 1] + 1
          got := FALSE
          WHILE  ( i <> end.store.ctr)  AND (got <> TRUE)
            SEQ
              IF
                (store.var[i] = term)
                  got := TRUE
                TRUE
                  i := i + 1
          IF
            (got <>  TRUE )
              SEQ
                store.var[end.store.ctr] := term
                store.envir[end.store.ctr] := uninstant
                end.store.ctr := end.store.ctr + 1
                print.queue[queue.pt] := term
                print.queue[queue.pt + 1] := end.queue
                queue.pt := queue.pt + 1
            TRUE
              SEQ
                queue.ptr := 0
                IF
                  (store.envir[i] < lis) OR (store.envir[i] < struct)
                    SEQ
```

```
IF
 (store.envir[i] < lis)
  num.lis := store.envir[i]
 TRUE
  num.lis := func.list[store.envir[i] - lis]
temp := list[num.lis][0]
IF
 (list[num.lis][temp] = split)
  num.terms := temp + 1
 TRUE
  num.terms := temp
SEQ j = 1 FOR num.terms
 IF
  (list[num.lis][j]  < lis)
    SEQ
     list.queue[queue.ptr] := list[num.lis][j]
     list.queue[queue.ptr + 1] := end.queue
     queue.ptr := queue.ptr + 1
  (list[num.lis][j] > lis) AND (list[num.lis][j] < struct)
    SEQ
     num.list := list[num.lis][j]   lis
     list.queue[queue.ptr] := func.list[num.list]
     list.queue[queue.ptr + 1] := end.queue
     queue.ptr := queue.ptr + 1
  ((list[num.lis][j] > struct) AND (list[num.lis][j] < shar.var))
    SEQ
     IF
      pro.num = 1
       k := 0
      TRUE
       SEQ
        k := sto.proc[pro.num - 1] + 1
     got := FALSE
     found := 0
     WHILE (k <> end.store.ctr) AND (got <> TRUE)
      SEQ
       check.var(list[num.lis][j],store.var[k],found)
       IF
        (found = 1)
          SEQ
           k := end.store.ctr
           got := TRUE
        TRUE
          k := k + 1
     IF
```

```
      (got = FALSE)
        SEQ
          find := 0
          check.print.queue(list[num.lis][j],print.queue,find)
          IF
            (find = 0)
              SEQ
                print.queue[queue.pt] := list[num.lis][j]
                print.queue[queue.pt + 1] := end.queue
                queue.pt := queue.pt + 1
                store.var[end.store.ctr] := list[num.lis][j]
                store.envir[end.store.ctr] := uninstant
                end.store.ctr := end.store.ctr+  1
            TRUE
              SKIP
      TRUE
        SEQ
          IF
            (store.envir[k] < lis)
              SEQ
                list.queue[queue.ptr] := store.envir[k]
                list.queue[queue.ptr + 1] := end.queue
                queue.ptr := queue.ptr + 1
            (store.envir[k] > lis) AND (store.envir[k] < struct)
              SEQ
                num.list := store.envir[k] - lis
                list.queue[queue.ptr] := func.list[num.list]
                list.queue[queue.ptr + 1] := end.queue
                queue.ptr := queue.ptr + 1
            TRUE
              SKIP
(list[num.lis][j] > ground.term) AND (list[num.lis][j] < non.shar.var)
  SEQ
    IF
      (pro.num = 1)
        k := 0
      TRUE
        SEQ
          k := sto.proc[pro.num   1] + 1
    found := 0
    WHILE(k <> end.store.ctr ) AND (found <> 1)
      SEQ
        IF
          (store.var[k] = list[num.lis][j])
            SEQ
```

```
            list[num.lis][j] := store.envir[k]
            found := 1
          TRUE
            k := k + 1
      IF
       (found = 0) AND (list[num.lis][j-1] = split)
         SEQ
          list[num.lis][j - 1] := 0
          list[num.lis][j] := 0
          list[num.lis][0] := list[num.lis][0]   1
       (found = 0)
         SKIP
       TRUE
         IF
          (store.envir[k] < lis)
            SEQ
             list.queue[queue.ptr] := store.envir[k]
             list.queue[queue.ptr + 1] := end.queue
             queue.ptr := queue.ptr + 1
          (store.envir[k] > lis) AND (store.envir[k] < struct)
            SEQ
             num.list := store.envir[k] - lis
             list.queue[queue.ptr] := func.list[num.list]
             list.queue[queue.ptr + 1] := end.queue
             queue.ptr := queue.ptr + 1
          TRUE
            SKIP
(list[num.lis][j] > struct) AND (list[num.lis][j] < shar.var)
  SEQ
   got := FALSE
   IF
    (pro.num = 1)
      k := 0
    TRUE
      SEQ
       k := sto.proc[pro.num   1] + 1
   found := 0
   WHILE (k <> end.store.ctr) AND (got <> TRUE)
    SEQ
     check.var(list[num.lis][j],store.var[k],found)
     IF
      (found = 1)
        SEQ
         find := 0
         check.print.queue(store.var[k],print.queue,find)
```

```
                        IF
                         (find = 0 )
                           SEQ
                            print.queue[queue.pt] := store.var[k]
                            print.queue[queue.pt + 1] := end.queue
                            queue.pt := queue.pt + 1
                          TRUE
                           SKIP
                        k := end.store.ctr
                        got := TRUE
                      TRUE
                       k := k + 1
                  IF
                   (got = FALSE)
                     SEQ
                      find := 0
                      check.print.queue(list[num.lis][j],print.queue,find)
                       IF
                        (find = 0 )
                          SEQ
                           print.queue[queue.pt] := list[num.lis][j]
                           print.queue[queue.pt + 1] := end.queue
                           queue.pt := queue.pt + 1
                           store.var[end.store.ctr] := list[num.lis][j]
                           store.envir[end.store.ctr] := uninstant
                           end.store.ctr := end.store.ctr + 1
                         TRUE
                          SKIP
                     TRUE
                      SKIP

                TRUE
                 SKIP
            IF
             (queue.ptr <> 0)

list.queue.check(end.store.ctr,pro.num,sto.proc,list.queue,store.envir,store.var,queue.ptr,p
rint.queue,queue.pt)
              TRUE
               SKIP
          (store.envir[i] > struct) AND (store.envir[i] < shar.var)
            SEQ
             got := FALSE
             IF
              (pro.num = 1)
```

```
                k := 0
              TRUE
                SEQ
                  k := sto.proc[pro.num - 1] + 1
            found := 0
            WHILE (k <>end.store.ctr)
              SEQ
               check.var(store.var[k],store.envir[i],found)
               IF
                 (found = 1)
                   SEQ
                    store.envir[i] := store.envir[k]
                    got := TRUE
                    k := end.store.ctr
                 TRUE
                   k := k +1
            IF
              (got = FALSE)
                SEQ
                 store.envir[i] := uninstant
              TRUE
                SKIP

          TRUE
            SEQ
              print.queue[queue.pt] := store.envir[i]
              print.queue[queue.pt + 1] := end.queue
              queue.pt := queue.pt + 1
    TRUE
      SKIP
:
PROC construct(INT num.terms,lis2,lis3,trm2,trm3)
 INT num.lis,trm,lit,term1,tem  :
 SEQ
  IF
    (lis3 < lis)
      SEQ
        trm := trm3
        lit := lis3
        trm2 := non.shar.var.ctr
        tem := trm2

    (lis2 < lis)
      SEQ
        trm := trm2
```

```
          lit := lis2
          trm3 := non.shar.var.ctr
          tem := trm3
        TRUE
          SKIP
      IF
        (list[lit][list[lit][0]] = split) AND (trm <> empty) AND (list[trm][list[trm][0]] =
split)
          SEQ
            list[list.ctr][0] := num.terms + 1
            [list[list.ctr] FROM 1 FOR num.terms] := [list[lit] FROM 1 FOR num.terms]
            list[list.ctr][num.terms + 1] := split
            list[list.ctr][num.terms + 2] := tem
        (list[lit][list[lit][0]]  = split) AND (list[lit][(list[lit][0] + 1) ] < lis)
          SEQ
            list[list.ctr][0] := num.terms + 1
            num.lis := list[ lit][(list[lit][0] + 1)]
            [list[list.ctr] FROM 1 FOR (list[lit][0] - 1)] := [list[lit] FROM 1 FOR (list[lit][0]
 1)]
            [list[list.ctr] FROM list[lit][0] FOR ((num.terms - list[lit][0]) + 1) ] :=
[list[num.lis] FROM 1 FOR ((num.terms - list[lit][0])  + 1)]
            list[list.ctr][num.terms + 1] := split
            list[list.ctr][num.terms + 2] := tem

        TRUE
          SEQ
            list[list.ctr][0] := num.terms + 1
            [list[list.ctr] FROM 1 FOR num.terms] := [list[lit] FROM 1 FOR num.terms]
            list[list.ctr][num.terms + 1] := split
            list[list.ctr][num.terms + 2] := tem
      non.shar.var.ctr := non.shar.var.ctr + 1

  :
  PROC reset(INT trm2,lis2,trm3,lis3,[]INT store.envir,store.va,INT
proc.num,end.store.ctr,[]INT sto.proc,BOOL run)
    INT num.terms,num.trms,num.lis,k :
    BOOL done,got :
    SEQ
      IF

        ((trm2 <> empty) AND (trm2 < lis )) AND (((trm3 < shar.var) AND (trm3 >
struct)) OR ((trm3 > ground.term) AND (trm3 < non.shar.var)))
          SEQ
            IF
              (list[lis2][list[lis2][0]] = split) AND (list[trm2][list[trm2][0]] = split)
```

```
            SEQ
              num.terms := list[lis2][0] - list[trm2][0]
            (list[lis2][list[lis2][0]] = split) AND (list[lis2][(list[lis2][0] + 1) ] < lis)
              SEQ
               num.lis := list[lis2][(list[lis2][0] + 1)]
               num.trms := (list[lis2][0] + list[num.lis][0]) - 1
               num.terms := num.trms - list[trm2][0]
            TRUE
              num.terms := list[lis2][0] - list[trm2][0]

          construct(num.terms,lis2,lis3,trm2,trm3)
          store.va[end.store.ctr] := lis3
          store.envir[end.store.ctr] := list.ctr
          list.ctr := list.ctr + 1
          end.store.ctr := end.store.ctr + 1
          start.store.ctr := end.store.ctr
       ((trm3 < lis)   AND (trm3 <> empty) ) AND (((trm2 > struct ) AND (trm2 <
shar.var)) OR ((trm2 > ground.term) AND (trm2 < non.shar.var)))
         SEQ
          decompose(store.envir,store.va,sto.proc,end.store.ctr,proc.num,trm3,head3,run)
          IF
           (trm3 = empty) AND (list[lis3][list[lis3][0]] = split) AND (list[lis3][(list[lis3][0]
+ 1)] < lis)
             SEQ
              num.lis := list[lis3][(list[lis3][0] +1)]
              num.terms := (list[lis3][0] + list[num.lis][0]) - 1
           (trm3 = empty) AND (list[lis3][list[lis3][0]] = split) AND (list[lis3][(list[lis3][0]
+ 1)] > lis  )
             SEQ
              num.terms := 0
              run := FALSE
              find := FALSE
            (trm3 = empty)
             SEQ
              num.terms := list[lis3][0]
            (list[lis3][list[lis3][0]] = split) AND (list[trm3][list[trm3][0]] = split)
             SEQ
              num.terms := list[lis3][0] - list[trm3][0]
            (list[lis3][list[lis3][0]] = split) AND (list[lis3][(list[lis3][0] + 1) ] < lis)
             SEQ
              num.lis := list[lis3][(list[lis3][0] + 1)]
              num.trms := (list[lis3][0] + list[num.lis][0]) - 1
              num.terms := num.trms - list[trm3][0]
            TRUE
             num.terms := list[lis3][0] - list[trm3][0]
```

```
          IF
           (num.terms <> 0)
            SEQ
             construct(num.terms,lis2,lis3,trm2,trm3)
             store.va[end.store.ctr] := lis2
             store.envir[end.store.ctr] := list.ctr
             list.ctr := list.ctr + 1
             end.store.ctr := end.store.ctr + 1
             start.store.ctr := end.store.ctr
            TRUE
             SKIP
       (trm2 = empty)
         SEQ
          run := FALSE
       ((trm3 < shar.var ) AND (trm3 > struct)) OR ((trm3 < non.shar.var) AND (trm3 >
ground.term))
         SEQ
          IF
           (trm3 < shar.var)
            SKIP
           TRUE
            SKIP
          decompose(store.envir,store.va,sto.proc,end.store.ctr,proc.num,trm3,head3,run)
          num.terms := 0
          done := FALSE
          k := 0
          WHILE(store.va[k] <> lis3)
           k := k + 1
          num.lis := store.envir[k]
          WHILE(done <> TRUE)
           SEQ
            k := 1
            WHILE(list[num.lis][k] <> split)
             SEQ
              num.terms := num.terms + 1
              list[list.ctr][num.terms] := list[num.lis][k]
              k := k + 1
            k := 0
            got := FALSE
            IF
             (list[num.lis][(list[num.lis][0] + 1)] < lis)
              num.lis := list[num.lis][(list[num.lis][0] + 1)]
             (list[num.lis][(list[num.lis][0] + 1)] < non.shar.var ) AND
(list[num.lis][(list[num.lis][0] + 1)] > ground.term)
              SEQ
```

```
            WHILE(k <> end.store.ctr) AND (got <> TRUE)
              SEQ
                IF
                 (store.va[k] = list[num.lis][(list[num.lis][0] + 1)] )
                   SEQ
                     got := TRUE
                     num.lis := store.envir[k]
                  TRUE
                    k := k + 1
              IF
                (got = TRUE)
                  SEQ
                    SKIP
                TRUE
                  done := TRUE
            TRUE
              SKIP
        list[list.ctr][0] := num.terms + 1
        list[list.ctr][(num.terms + 1)] := split
        list[list.ctr][(num.terms + 2)] := non.shar.var.ctr
        trm2 := non.shar.var.ctr
        non.shar.var.ctr := non.shar.var.ctr + 1
        store.va[end.store.ctr] := lis2
        store.envir[end.store.ctr] := list.ctr
        check.dec(lis2,list.ctr,end.store.ctr,proc.num,sto.proc,store.envir,store.va)
        list.ctr := list.ctr + 1
        end.store.ctr := end.store.ctr + 1
        start.store.ctr := end.store.ctr
      TRUE
        SEQ
          run := FALSE
          find := FALSE
:
PROC reset.list(INT temp,trm2,lis2)
  INT num,num.terms :
  SEQ
    IF
      (((list[lis2][0] - list[trm2][0]) - 1) = 0)
        temp := trm2
      TRUE
        SEQ
          num := (list[lis2][0] - list[trm2][0]) - 1
          num.terms := num + list[trm2][0]
          [list[list.ctr] FROM 1 FOR num] := [list[lis2] FROM 1 FOR num]
```

```
        [list[list.ctr] FROM (num + 1) FOR list[trm2][0]] := [list[trm2] FROM 1 FOR
list[trm2][0]]
        list[list.ctr][0] := num.terms
        temp := list.ctr
        list.ctr := list.ctr + 1
  :

  -- ************************************************************************
  -- This procedure is used for recursive functions that call other recursive functions.
  -- The structure of the solutions of such functions is stored in the form a tree. This
  -- procedure is used to traverse the tree to go from one solution to the next.
  -- ************************************************************************

  PROC tree.traversal(INT t.ctr,[100][100]INT t.structure,INT c.num,c.proc.num)
    BOOL got:
    SEQ
      t.ctr := t.ctr - 1
      got := FALSE
      WHILE (t.ctr <> 0) AND (got <> TRUE)
        SEQ
          t.structure[t.ctr][num.child.trav] := t.structure[t.ctr][num.child.trav] + 1
          IF
            ((t.structure[t.ctr][num.child] - t.structure[t.ctr][num.child.trav]) = 0)
              SEQ
                t.ctr := t.ctr - 1
                c.proc.num := c.proc.num   1
            TRUE
              SEQ
                got := TRUE
                c.num := t.structure[t.ctr][(t.structure[t.ctr][num.child.trav] + 3)]
  :
  PROC flatten.list(INT ptr)
    INT num.terms :
    BOOL got :
    SEQ
      num.terms := 1
      got := FALSE
      WHILE (got <> TRUE)
        SEQ
          SEQ j = 1 FOR (list[ptr][0]   1)
            SEQ
              list[list.ctr][num.terms] := list[ptr][j]
              num.terms := num.terms + 1
          ptr := list[ptr][(list[ptr][0] + 1)]
          IF
```

```
        (list[ptr][list[ptr][0]] = split)
          SKIP
        TRUE
          SEQ
            got := TRUE
            SEQ j = 1 FOR list[ptr][0]
              SEQ
                list[list.ctr][num.terms] := list[ptr][j]
                num.terms := num.terms + 1
            list[list.ctr][0] := num.terms - 1
            ptr := list.ctr
            list.ctr := list.ctr + 1
  :

  -- ***************************************************************************
  -- This function is the virtual machine instructions implementation of the Prolog
  -- procedure concatenate. This function conactenates two lists to form a third list.
  -- ***************************************************************************

  PROC conc(INT c.ctr,e.store.ctr,[]INT store.env,store.var,store.process,INT
process.num,term1,term2,term3,BOOL run1)
    INT head2,head1,list1,list2,list3,temp,succ1 :
    INT env ,e ,queue.pt,k,succ1,succ2:
    BOOL run2:
    [100] INT print.queue:
    SEQ
      list1 := term1
      list2 := term2
      list3 := term3
      run2 := TRUE
      start.store.ctr := e.store.ctr
      WHILE run2
        SEQ
          unify(term1,empty,succ1,var1,envir1)
          IF
            (succ1 <> 1) AND ((term2 < lis) AND (term3 < lis))
              SEQ
                unify(term2,term3,succ2,var2,envir2)
                check(e.store.ctr,process.num,store.process,store.env,store.var,var2,envir2)
                IF
                  (list[list3][0] = list[list2][0])
                    SEQ
                      store.var[e.store.ctr] := term1
                      store.env[e.store.ctr] := empty
                      e.store.ctr := e.store.ctr + 1
```

```
            TRUE
              SKIP
          TRUE
            SKIP
        IF
          (succ1 = 1) OR (succ2 = 1)
            SEQ
              check(e.store.ctr,process.num,store.process,store.env,store.var,var1,envir1)
              unify(term2,term3,succ1,var2,envir2)
              check(e.store.ctr,process.num,store.process,store.env,store.var,var2,envir2)
              IF
                (succ1 = 1) OR (term1 = empty) OR (term3 = empty)
                  SEQ
                    print.queue[0] := end.queue
                    queue.pt := 0
                    c.ctr := c.ctr + 1

instantiate(e.store.ctr,process.num,store.process,store.env,store.var,list1,print.queue,queu
e.pt)

instantiate(e.store.ctr,process.num,store.process,store.env,store.var,list2,print.queue,queu
e.pt)

instantiate(e.store.ctr,process.num,store.process,store.env,store.var,list3,print.queue,queu
e.pt)
                  IF
                    (e.store.ctr = 0)
                      SEQ
                        find := TRUE
                        run2 := FALSE
                    TRUE
                      SEQ
                        store.process[process.num] := e.store.ctr - 1
                        process.num := process.num + 1


reset(term2,list2,term3,list3,store.env,store.var,process.num,e.store.ctr,store.process,run2
)
                        find := TRUE
                        IF
                          ((term1 < non.shar.var) AND (term1 > ground.term)) AND ((term3
< non.shar.var) AND (term3 > ground.term))
                            SEQ

decompose(store.env,store.var,store.process,e.store.ctr,process.num,term1,head1,run2)
```

```
decompose(store.env,store.var,store.process,e.store.ctr,process.num,term3,head3,run2)
                    unify(head1,head3,succ1,var1,envir1)

check(e.store.ctr,process.num,store.process,store.env,store.var,var1,envir1)
                  TRUE
                   SEQ
                    SKIP
            TRUE
             SEQ
              find := FALSE
              run2 := FALSE
        TRUE
         SEQ
          IF
           (term3 < lis) OR ((term3 < shar.var) AND (term3 > struct)) OR ((term3 >
ground.term) AND (term3 < non.shar.var)) OR (term3 = empty)
            SEQ

decompose(store.env,store.var,store.process,e.store.ctr,process.num,term1,head1,run2)
              IF
               (run2)
                SEQ

decompose(store.env,store.var,store.process,e.store.ctr,process.num,term3,head3,run2)
                IF
                 (run2)
                  SEQ
                   unify(head1,head3,succ1,var1,envir1)
                    IF
                     (succ1 = 1 )

check(e.store.ctr,process.num,store.process,store.env,store.var,var1,envir1)
                      TRUE
                       SEQ
                        run2 := FALSE
                        find := FALSE
                  TRUE
                   SKIP
              TRUE
               SKIP
          ((term1 > struct) AND (term1 < shar.var))
           SEQ
            store.var[e.store.ctr] := list1
            store.env[e.store.ctr] := empty
```

```
            e.store.ctr := e.store.ctr + 1
            term1 := empty
          TRUE
           SKIP
  IF
   (find)
    SKIP
   TRUE
    SEQ
     run1 := FALSE
:
PROC check.envir(INT term,end.ctr,[]INT store.va,store.env)
 INT j,found:
 SEQ
  j := 0
  WHILE ( j < end.ctr )
   SEQ
    found := 0
    check.var(term,store.va[j],found)
    IF
     (found = 1)
      SEQ
       term := store.env[j]
     TRUE
      SKIP
    j := j + 1
:

-- ********************************************************************
-- This procedure is used to check a given file which contains input data. Thus the data
-- that has to be input to a Prolog procedure can be present in a file. The user is not
-- required to input the data everytime.
-- ********************************************************************

PROC check.file(INT length,[]BYTE b,INT e.ctr,[]INT store.va,store.env)
 INT found,j,temp,count,tem:
 [250] BYTE c:
 BOOL got:
 SEQ
  j := 0
  WHILE (b[j] = ' ')
   j := j + 1
  temp := j
  IF
   (b[j] >= 'A') AND (b[j] <='Z')
```

```
SEQ
 WHILE(b[j] <> ' ')
  SEQ
   j := j + 1
 count := j - temp
 [c FROM 0 FOR count] := [b FROM temp FOR count]
 assign.shar.var(c,count)
 tem := shar.var.ctr - 1
 WHILE(b[j] = ' ')
  j := j + 1
 IF
  (b[j] = '=')
   SEQ
    j := j + 1
    WHILE (b[j] = ' ')
     j := j + 1
    [a FROM 0 FOR (length - j)  ] := [b FROM j FOR (length - j) ]
    len := length - j
    check.term(term)
    IF
     (error = FALSE)
      SEQ
       got := FALSE
       j := 0
       WHILE(j < e.ctr) AND (got = FALSE)
        SEQ
         found := 0
         temp := shar.var.ctr - 1
         check.var(store.va[j],temp,found)
         IF
          (found = 1)
           SEQ
            store.env[j] := term
            got := TRUE
          TRUE
           j := j + 1
       IF
        (got = FALSE)
         SEQ
          store.va[e.ctr] := tem
          store.env[e.ctr] := term
          e.ctr := e.ctr + 1
        TRUE
         SKIP
     TRUE
```

```
            SKIP
          TRUE
            SKIP
        TRUE
          SKIP
  :

  -- ***************************************************************
  -- This is the Occam implementation of the Prolog procedure premutation. It is
  -- implemented with the virtual machine instructions.
  -- ***************************************************************
  INT
head2,end.current.ctr,e.store.ctr,child.ctr,process.num,current.node.number,current.proce
ss.num,tree.ctr,list1,list2,temp,succ1,succ2,inst.num,count,start,index,tem,queue.pt,end,te
mp2,temp3,length,e.ctr:
  INT32 stream.id:
  [256]BYTE b:
  BYTE res:
  [800]INT
store.process,store.env,store.var,current.store.env,current.store.var,print.queue,s.proc:
  [800][2]INT current.store.process,variable.store:
  [100][100]INT tree.structure:
  BOOL run1,got :
  SEQ
    current.node.number := 0
    tree.ctr := 1
    process.num := 1
    child.ctr := 1
    current.process.num := 1
    end.current.ctr := 0
    e.store.ctr := 0
    run1 := TRUE
    e.ctr := 0
    initialise.variables()
    so.write.string(keyboard,screen,"Enter the data file to be loaded ")
    so.read.echo.line(keyboard,screen,len,a,result)
    so.open(keyboard,screen,[a FROM 0 FOR len],spt.text,spm.input,stream.id,res)
    IF
      (res = spr.bad.name)
        SEQ
          so.write.nl(keyboard,screen)
          so.write.string(keyboard,screen,"Invalid file name ")
      (res <> spr.ok)
        SEQ
          so.write.nl(keyboard,screen)
```

```
        so.write.string(keyboard,screen,"File does not exist ")
      TRUE
       SEQ
        so.gets(keyboard,screen,stream.id,length,b,res)
        IF
         (length <> 0)
          check.file(length,b,e.ctr,var1 ,envir1)
         TRUE
          SKIP
        WHILE (length <> 0)
         SEQ
          so.gets(keyboard,screen,stream.id,length,b,res)
          IF
           (length <> 0)
            check.file(length,b,e.ctr,var1,envir1)
           TRUE
            SKIP
       so.close(keyboard,screen,stream.id,res)
    so.write.nl(keyboard,screen)
    so.write.string(keyboard,screen,"Enter the first term  ")
    so.read.echo.line(keyboard,screen,len,a,result)
    check.term(term1)
    WHILE (error <> FALSE)
     SEQ
      so.write.nl(keyboard,screen)
      so.write.string(keyboard,screen,"Enter the first term  ")
      so.read.echo.line(keyboard,screen,len,a,result)
      check.term(term1)
    IF
     (term1 < shar.var) AND (term1 > struct)
      SEQ
       check.envir(term1,e.ctr,var1,envir1)
     TRUE
      SKIP
    list1 := term1
    so.write.int(keyboard,screen,term1,8)
    WHILE run1
     SEQ
      unify(term1,empty,succ1,var1,envir1)
      IF
       ((succ1 = 1) OR (((term1 < shar.var) AND (term1 > struct)) OR ((term1 <
non.shar.var) AND (term1 > ground.term))))
         SEQ
          tem := 1
```

```
check(end.current.ctr,tem,store.process,current.store.env,current.store.var,var1,envir1)
          print.queue[0] := end.queue
          queue.pt := 0
          s.proc[0] := end.current.ctr - 1

print.instantiate(end.current.ctr,tem,s.proc,current.store.env,current.store.var,list1,print.qu
eue,queue.pt)
          IF
           (queue.pt = 0)
            SEQ
             run1 := FALSE
             find := TRUE
           TRUE
            SEQ
             so.read.echo.line(keyboard,screen,len,a,result)
             IF
              (a[0] = ';')
               SEQ

tree.traversal(tree.ctr,tree.structure,current.node.number,current.process.num)
                  IF
                   (tree.ctr = 0)
                    SEQ
                     run1 := FALSE
                     find := FALSE
                   TRUE
                    SEQ
                     tree.ctr := tree.ctr + 1
                     current.process.num := current.process.num - 1
                     IF
                      (current.node.number = 1)
                        SEQ
                         count := store.process[current.node.number] + 1
                         index := 0
                      TRUE
                        SEQ
                         count := store.process[current.node.number]
store.process[current.node.number - 1]
                         index := store.process[current.node.number - 1] + 1
                     start := current.store.process[current.process.num][0]
                     SEQ j = 0 FOR count
                       SEQ
                        IF
                         (store.env[(index + j)] <> uninstant)
```

```
                        SEQ
                         current.store.var[start] := store.var[(index + j)]
                         current.store.env[start] := store.env[(index + j)]
                         start := start + 1
                       TRUE
                        SKIP
                  current.store.process[current.process.num][1] := start - 1
                  end.current.ctr := start
                  term1 := variable.store[current.process.num][0]
                  current.process.num := current.process.num + 1

            TRUE
             SEQ
              run1 := FALSE
              find := TRUE
      TRUE
       SEQ
        tem := 1

decompose(current.store.env,current.store.var,store.process,end.current.ctr,tem,term1,hea
d1,run1)
        IF
         (run1) AND (term1 = empty)
          SKIP
         (run1)
          SEQ
           temp := non.shar.var.ctr
           temp3 := temp
           non.shar.var.ctr := non.shar.var.ctr + 1
           child.ctr := 1
           list[list.ctr][0] := 1
           list[list.ctr][1] := head1
           temp2 := list.ctr
           list.ctr := list.ctr + 1

conc(child.ctr,e.store.ctr,store.env,store.var,store.process,process.num,temp,temp2,term1,
run1)
           IF
            (run1)
             SEQ
              child.ctr := child.ctr - 1
              tree.structure[tree.ctr][node.number] := current.node.number
              tree.structure[tree.ctr][num.child] := child.ctr
              tree.structure[tree.ctr][num.child.trav] := 0
              tem := child.ctr
```

```
SEQ j = 3 FOR tem
  SEQ
    tree.structure[tree.ctr][j] := process.num - child.ctr
    child.ctr := child.ctr - 1
inst.num := tree.structure[tree.ctr][3]
current.node.number := tree.structure[tree.ctr][3]
IF
  (inst.num = 1)
    SEQ
      count := store.process[inst.num] + 1
      index := 0
  TRUE
    SEQ
      count := store.process[inst.num] - store.process[inst.num - 1]
      index := store.process[inst.num - 1] + 1
current.store.process[current.process.num][0] := end.current.ctr
SEQ j = 0 FOR count
  SEQ
    IF
      (store.env[index + j] <> uninstant)
        SEQ
          current.store.var[end.current.ctr] := store.var[index + j]
          current.store.env[end.current.ctr] := store.env[index + j]
          end.current.ctr := end.current.ctr + 1
      TRUE
        SKIP
current.store.process[current.process.num][1] := end.current.ctr -
start := current.store.process[current.process.num][0]
end := current.store.process[current.process.num][1]
got := FALSE
SEQ j = start FOR ((end - start) + 1)
  IF
    (current.store.var[j] = temp3)
      SEQ
        term1 := current.store.env[j]
        got := TRUE
    TRUE
      SKIP
IF
  (got = TRUE)
    SKIP
  TRUE
    term1 := empty
IF
  (term1 = empty)
```

```
                SKIP
              (list[term1][list[term1][0]] = split)
                flatten.list(term1)
              TRUE
                SKIP
            variable.store[current.process.num][0] := term1
            current.process.num := current.process.num + 1
            tree.ctr := tree.ctr + 1

          TRUE
            find := FALSE
        TRUE
          find := FALSE

IF
  (find)
    SEQ
      so.write.nl(keyboard,screen)
      so.write.string(keyboard,screen,"Yes")
  TRUE
    SEQ
      so.write.nl(keyboard,screen)
      so.write.string(keyboard,screen,"No")
so.exit(keyboard,screen,sps.success)
```

Bibliography

[1] Ivan Bratko, "Prolog Programming For Artificial Intelligence", Addison-Wesley Publishers, 1986.

[2] Sterling L., Shapiro E., "The Art of Prolog", The MIT Press, 1983.

[3] C.A.R. Hoare, Series Editor, "INMOS Limited, Occam 2 Reference Manual", Prentice Hall, 1988.

[4] DeGroot D., "Restricted AND Parallelism", Proceedings of the International Conference on Fifth Generation Computer Systems, pp 471-478, 1984.

[5] Dobry T., Despain A., Patt Y., "Performance Studies of a Prolog Machine Architecture", Proceedings of the 12th Annual International Symposium on Computer Architecture, pp 179-190, June 1985.

[6] Fagin B.S., Despain A.M., "Performance Studies of a Parallel Prolog Architecture", Proceedings of the 14th Annual International Symposium on Computer Architecture, pp 108-116, June 1987.

[7] Wise M.J., "EPILOG = PROLOG + DATAFLOW", SIGPLAN Notices, 17 (12), pp 80 - 86, 1982.

[8] Nakashima, Tomura S., Ueda K., "What is a variable in Prolog?", Proceedings of the 1984 International Conference on Fifth Generation Computer Systems.

[9] Conery J.S., Kibler D.F., "AND Parallelism in Logic Programs", Proceedings of the 1983 International Joint Conference on Artificial Intelligence.

[10] Shapiro E.Y., "Concurrent Prolog Papers", Cambridge, Mass.: MIT Press, v.1- pp 477-506, v.2 - pp 605-634, 1987.

[11] Wexler John., "Concurrent programming in Occam2", Chichester: E.Horwood; New York: Halsted Press, 1989.

[12] Brookes Graham R., "Introduction to Occam2 on the Transputer", Basingstoke: Macmillan, 1989.

[13] Jones Geraint., "Programming in Occam2", New York: Prentice Hall, 1988.