OPTIMIZATION ANALYSIS OF PROGRAMS IN LANGUAGES WITH POINTER VARIABLES

by

Hendrik Jacobus Boom

Research Report CS-76-32

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

June, 1976

OPTIMIZATION ANALYSIS OF PROGRAMS IN LANGUAGES WITH POINTER VARIABLES.

by Hendrik Jacobus Boom

A thesis submitted in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

at the

University of Waterloo

Waterloo, Ontario

Department of Applied Analysis and Computer Science

1974 July

Kevin Pammett, AA&CS, U of W. Nov 1975.

I would like to thank Professor Florentin and Professor Cowan for helping to teach me to formulate my thoughts.

Abstract

A technique is presented for analysing programs to determine information about the data structures they create and maintain. It identifies the variables that can be treated by standard optimizing techniques, despite the presence of pointer variables. It emphasizes the difference between values, models of values at compile time, and the names used for values in the program. The feasibility of the technique has been demonstrated by an implementation. The analysis technique may be useful for designing optimizing compilers and for proving program correctness.

Computing Reviews categories: 4.12, 5.24.

Key words and phrases: ALGOL 68, pointer variables, optimizing compilers, compilers, proving program correctness.

Table of contents.

Chapter 1. Introduction.

1.1 Introduction to the introduction.

Optimizing compilers have been found useful for programming languages like FORTRAN. Inexperienced programmers, however, tend to have excessive faith in optimization, and fail to write efficient code themselves. They believe, rather naively, that the machine will compensate for their own bad code. Experienced programmers, who are properly sceptical of mechanical abilities, can use optimizing compilers to good effect. They let the optimizer use machine-dependent techniques to generate good object code, techniques which high-level language programs cannot express directly. They do not expect the optimizer to replace their algorithm by a better one, nor to make other major changes in their source code. The chief constraint on optimizing compilers is the language they translate. It is folly to expect an optimizing compiler to detect the circumlocutions used to express operations not provided in the language, and then compile efficiently the program that the programmer would have written had the language permitted it.

On the other hand, new features, such as dynamic storage allocation and pointer variables, that allow a programmer to state his intentions clearly, can also hinder an optimizing compiler, by interfering with the principles by which present optimizers work.

Much previous work has been done on optimizing compilers, most of it for Fortran-like languages. [Cocke 1] contains a rather extensive coverage of the state of the art as it is applied in production compilers. [Allen 1] contains a method for handling pointers which is different from the one presented here. It is briefly mentioned in section 5.3.3.

Existing optimizing compilers need to know when the values of variables are set and used. Without this, it is impossible to know when to store variables in fast registers or when to perform common subexpression elimination. If a variable is set via a pointer, it may be very difficult for an optimizing compiler to know whether that variable is also involved, possibly under a different name, in a common subexpression.

Consider the following fragments from an ALGOL 68 program:

```
real x, t, u;
ref real yy;

...

u := x * 2;
ref real( yy ) := 6;
```

```
t := x * 2;

...
```

In this example, "x", "t", and "u" are identifiers for real variables. "yy" is the identifier for a "reference-to-real" variable; i.e. a variable whose values are pointers to real variables. The line "ref real(yy) := 6" specifies that the real variable pointed to by the pointer in "yy" is to receive the value 6.

Is the second computation of "x * 2" redundant? One cannot tell without knowing whether the pointer variable yy points to x. If it does, a new value is referred to by the variable x in the second occurrence of "x * 2"; if not, the second computation is redundant.

Some of these problems also obstruct the verification of program correctness. The presence of pointer variables can make programs exhibit extreme combinatorial complexity. Nonetheless, analysis algorithms may yield information that is useful for verification. If, for example, an algorithm could detect that a certain group of variables were unchanged in a segment of code, any predicate that involves only those variables and is true at the beginning of the segment will still be true at the end of that segment. A detailed formal proof, which might be extremely complicated, would no longer be needed. Algorithms developed for program optimization might well uncover reliable and useful information which can aid the process of verification.

1.2 Prospectus.

This thesis sets forth a technique for the analysis of programs with pointer variables. It distinguishes a class of variables for which information can be obtained by examining source text. A simple programming language with pointer variables is presented as an example, and the algorithm for analysis of its programs is also presented. The correctness of this algorithm is demonstrated by a proof, and its feasibility is demonstrated by an implementation.

The rest of Chapter 1 discusses the problems encountered, and informally suggests some of the techniques used. Chapters 2 and 3 describe the simple programming language, and specify in detail how these techniques apply to it. Chapter 4 describes an ALGOL W [Wirth 1, Sites 1] program that implements the algorithms of Chapters 2 and 3. Chapter 5 describes extensions that might be made to these analysis techniques.

1.3 Representatives.

The first thing to recognize is that the program analyser in an optimizing compiler does not deal with the values (e.g. integers, real numbers, and pointers) and variables (i.e. pieces of storage) themselves, since they exist only at run time, but with (compile-time) 'representatives' of (run-time) values and variables. A representative of a variable consists of the information the compiler uses at compile time to keep track of the variable that will exist at run time. This might consist of a symbol-table entry, the location and data type of a compiler-generated temporary, the fact that several identifiers possess overlapping storage (e.g., the FORTRAN equivalence statement, or the ALGOL 68 identity declaration), information indicating how a value is to be found at run time, etc.

A single compile-time representative of a variable might well represent different variables at different moments during execution (variables are created and destroyed at block entry and exit), and may even represent several different variables at one time (recursion).

The distinction between variables and representatives of variables is of little practical importance for languages like FORTRAN and BASIC. Since they are usually implemented with completely static storage allocation, there can be a one-to-one correspondence between variables and representatives of variables. In this thesis, which discusses more general languages, the distinction between things and representatives of things will be crucial for clear discussion.

The prefix 'm-' will be used as a linguistic tool to indicate representatives. Thus, an 'mvalue' is a representative of a value.

ALGOL 68 deliberately blurs the distinction between variables and values. It treats the class of variables as a subset of the class of values. It does this by identifying a variable with the pointer pointing to it. Variables can thus be treated as freely as any other kind of value: pointers are objects that can be passed as parameters, copied, and so forth. Many older languages already pass a variable as a parameter to a subroutine by passing a pointer. Variables will often be called 'references'. The ALGOL 68 Report confuses the issue by calling them 'names'. They can be assigned, passed as parameters, compared for equality, etc. To reflect this simplification of concept, we shall henceforth consider representatives of values to include representatives of variables as a special case. Thus, every mvariable is an mvalue.

Let us consider an extended example:

```
real x;
real y;
```

These are variable declarations. Each of them creates a variable whose values must be real numbers; and gives it an identifier. One variable is named "x", and the other is named "y". We sometimes also say that such a variable is 'possessed' by the identifier.

```
    ref real z = x;
```

This is an identity declaration, which defines "z" to be synonymous with "x". "z" and "x" will be two names for the same piece of storage. At this point, there are two variables, and three names for them.

```
    real count = 6 + y;
```

This is another identity declaration. It causes six to be added to the current value of y. The sum is permanently (well, until block exit) given the name "count". The value of "count" may not hereafter be altered; it is a constant. The identifier "count" possesses a real value, not a variable.

```
    proc p = (ref real u, real v) void:
         begin real a;
         y := u := v - 1;
         if v > 0
    . then p(a, v - 1)
      fi
      end;
```

This is a recursive (and not very useful) procedure. A 'routine' is created, consisting of some executable code, and an environment; the environment will enable the routine to access the values of the nonlocal identifiers "y" and "p".

```
p(x,6)
```

The routine is called, providing it with a real variable and the value six as parameters. Note that a single variable is now named by the three identifiers "u", "x", and "z".

After p calls itself once, the identifiers "a" and "u" will each possess two variables. Thus,

A single value can be named by several identifiers.

As a special case, a single variable can be named by several identifiers.

An identifier can name several values (even at the same time).

Existing optimization techniques do not clearly distinguish between identifiers (the strings of letters and digits coded by a programmer to denote objects in the run-time machine), the variables and other values possessed by identifiers, and the values referred to by these variables. Thus, in a FORTRAN program, the sequence of characters "ABC" might be considered to be

1) a name chosen by the programmer, possibly for mnemonic reasons,

2) a piece of storage in the computer, or

3) any of the values residing in that piece of storage from time to time.

This semantic confusion certainly does not help an analyser attempting to analyse a program that uses pointers. I shall maintain a sharper distinction between these meanings. The name coded by the programmer will be called an 'identifier'; the piece of storage (which I identify with a pointer) will be called a 'variable' or a 'reference'; the values residing in that piece of storage are said to be 'referred to' by the variable.

Furthermore, it will be clear that compilers manipulate <u>models</u> of run-time objects, and not the objects themselves.

1.4 Individuality.

Once we have recognized the difference between values and mvalues, two possibilities come to mind. A value may be represented by one mvalue, or by more than one mvalue.

For example, consider the procedure declared below:

```
proc p = (ref real x, y) real:
     begin real v;
     v := x + 2;
     y := v;
     v + (x + 2)
     end
```

This procedure receives two real variables as parameters, and assigns to one of them. The outcome of the procedure depends on whether the two variables are the same, as in the procedure call:

```
real q;
p(q, q)
```

An optimizing compiler will have to take great care before treating the second occurrence of "x + 2" as redundant. Undetected synonymy, such as between x and y above, can cause severe troubles for an optimization algorithm. Synonymy can also arise (and be even harder to detect) through the use of pointer variables. It is important to find methods for discovering the amount of synonymy that is possible in a particular program.

If the analyser knows that a value is represented by only one mvalue, and (except for recursion) that this mvalue represents only one value, we call the value and mvalue 'individual', and otherwise, 'nonindividual'. Individual values have the property that the analyser knows, at compile time, when and how they are accessed. Once we have discovered which mvalues are individual, we can apply standard optimizing techniques to the individual mvalues. One of the main themes in this thesis is an attempt to reach a precise understanding of individuality and related concepts.

First, we may notice that individuality is of no importance for some data types. The value '65' may appear only once, or at many places in the run-time machine. It makes no difference whether it is represented by a single mvalue or by many, nor whether it occurs in the source program as an explicit number or is computed at run time. Each occurrence of '65' is, so to speak, independent of all the others. Performing operations on one of them has no effect on the others.

For some data types, this is not the case. For example, individuality is important for references and semaphores[Dijkstra 1]. If a value is assigned to a variable at one point in a program, it certainly does affect the value obtained by using that variable at another point in the program; dependence exists whether the compiler can detect it or not. For semaphores, such dependence is their very function, to control communication between different processes.

Individuality is a simple property of values and mvalues. More complex situations are possible. For example, if certain list structures are considered to be single values (like SNOBOL's patterns) and are represented by single mvalues, one might care whether two mvalues represent the same list structure, partially overlapping list structures, disjoint list structures, whether one is embedded in the other or whether they contain cycles. The complex and interesting situations of this paragraph will not be discussed further in this thesis.

1.5 Mstates.

For each programming language, one can construct an abstract machine which is 'ideal' for that language, in that it provides just those data structures and operations which are necessary in any implementation of the language. Abstract machines of this sort are quite useful in defining programming languages [Landin 1, Wirth 1, Lucas 1, Van Wijngaarden 1]. An implementation is obliged to provide these constructs in some form. Often, these abstract data structures and operations may appear quite different in their concrete representation because of various optimizations that the compiler may perform.

These abstract machines are quite different for different programming languages. The abstract machines for LISP, SNOBOL, and ALGOL 68 resemble each other in very few ways. They can nevertheless all be represented concretely on a single real machine.

The analysis algorithm presented in later chapters constructs 'mstates', i.e., representatives of the states of the run-time machine, and 'attaches' them at various points in the program. These mstates will consist of representatives of the values, of representatives of the relations between values, of representatives of the naming conventions, of representatives of parts of the control structure, and of models of other relevant abstract components of the run-time machine. An mstate for an

ALGOL machine might well have representatives for procedures, variables, values, and the relation between identifiers and their variables. These representatives will of course be called 'mprocedures', 'mvariables', etc.

As in any simulation problem, the representatives need not be exact replicas of the things represented; they need only contain an amount of information sufficient for the analysis.

We say that the representatives 'represent' the things being represented. An mstate attached at a point in the program must properly represent the state of the abstract machine at each moment that control passes that point of the program. In the coming chapters, representation is a dynamic relation, defined between static representatives constructed by the program analyser and the dynamic, ever-changing, run-time state of the abstract machine.

Because the analysis algorithm constructs representatives for abstract machines, and the abstract machine depends on the programming language, the details of the analysis algorithm necessarily depend on the language. In the next few chapters, a simple programming language is presented as an example to illustrate the principles behind the analysis techniques.

1.6 An example.

Consider the following ALGOL 68 real closed clause:

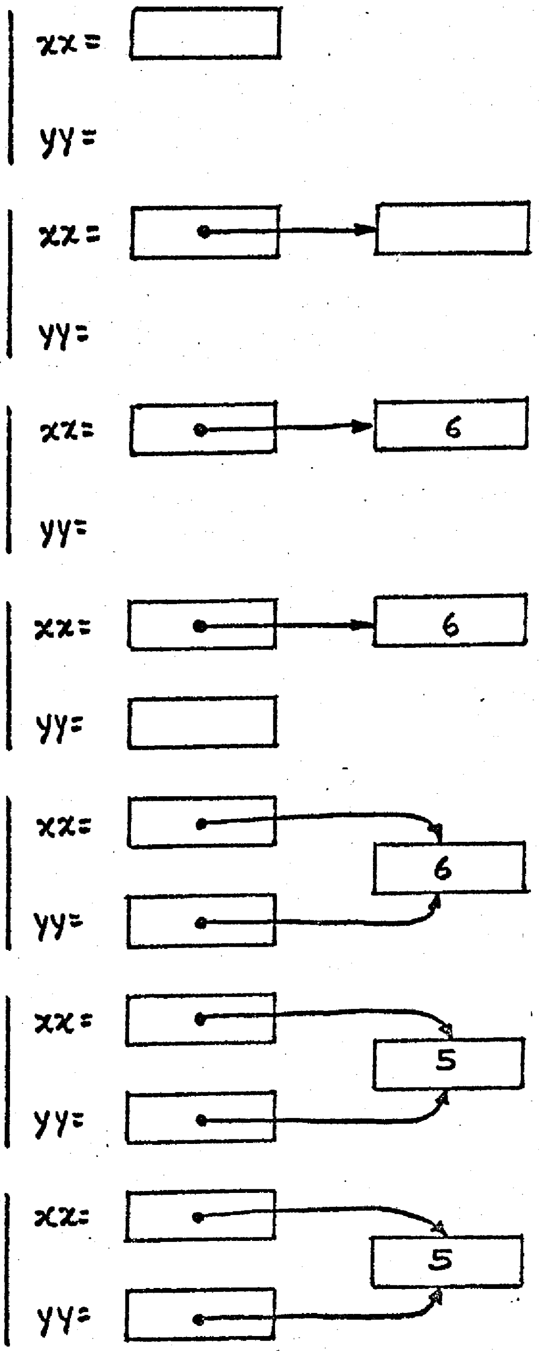
```
( ref real xx;
  xx := heap real;
  ref real(xx) := 6;
  ref real yy;
  yy := xx;
  ref real(xx) := 5;
  real(yy)
)
```

If we execute this clause, we pass through a succession of states. The next page illustrates an execution of the program. In a case as simple as this one, states and mstates are isomorphic. Therefore, only the mstate is drawn.

program

mstates between lines (the same as the states of the machine, in this simple case)

```
(
  ref real xx;

  xx := heap real ;

  ref real (xx) := 6;

  ref real  yy;

  yy := xx;

  ref  real (xx) := 5

  real  (yy)

)
```



▭ ..... piece of storage

●→ ..... pointer

5, 6 ..... integral values

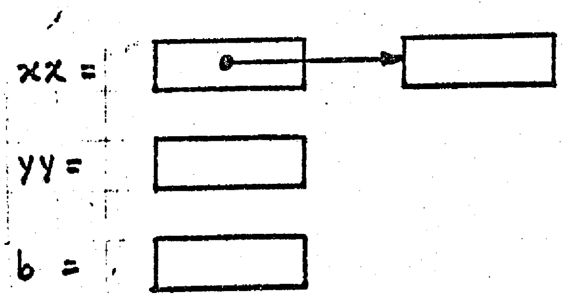
Note that it was possible to detect that the value of the expression obtained via yy was the same 5 assigned in an earlier line, even though the variable to which the 6 and the 5 were both assigned does not have an identifier of its own, and is accessed via pointers from two different identifiers. Such detection is beyond the ability of most optimizing algorithms, since they equate identifiers with mvalues. The algorithm in this thesis does detect this. This might enable an optimizer to replace the entire program segment by a 5.

1.7 Conditionals and merging.

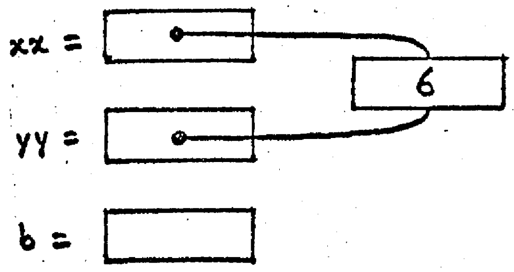
Consider

```
( ref real xx, yy; bool b;
  xx := heap real;
  ref real(xx) := 6 ;
  read(b);
  if b
     then yy := xx
     else yy := heap real fi;
  ref real(yy) := 5;
  real(xx)
)
```

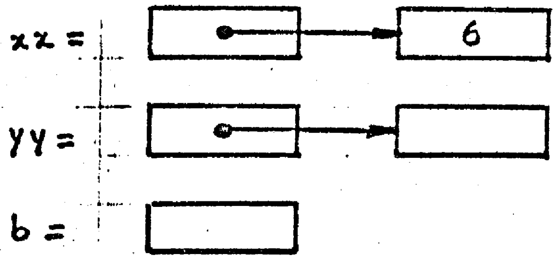
Matters go as in the last example until we get to the conditional clause. A picture of the mstate before the conditional clause is



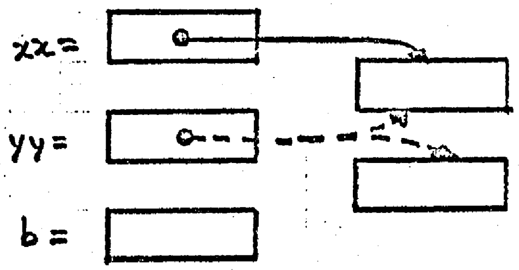
Note that the value read into b is unknown, and so is not included. After the then clause we have



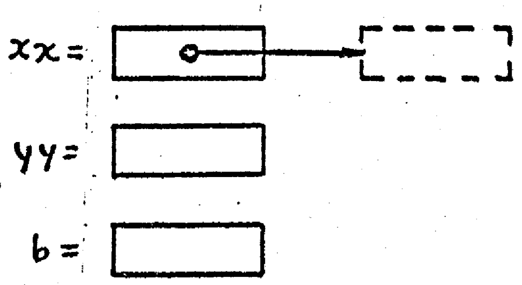
After the else clause we get



instead. These are not the same. Yet we must find some mstate to place after the entire conditional. We might like to get



The dotted lines indicate possible, instead of certain, truth. The algorithms presented in this thesis will, however, give only



The dotted lines here indicate that the mvariable has become nonindividual; i.e., in principle, at some time during execution, the variable might independently become represented by another mvariable, but the analyser will not know this. It can no longer tell when the value referred to by the variable that the mvalue represents is changed.

This new mstate is called the result of 'merging' the mstates obtained from the then and else clauses. An mstate obtained by merging other mstates must be capable of representing all machine states that can be represented by any of the other mstates. In general, merging will destroy information from the original mstates.

1.8 Loops.

A loop, such as the one in

```
A;
while B do C od;
D
```

causes difficulty for the attempt to construct representatives. The mstate after C is most naturally computed from that before C, which is most naturally computed from that after B, which ... is computed from those after A and C. The circularity here results from suppressing the time dimension. It is possible to handle this case by assuming no information at all in the mstate before B. Unfortunately, this is unsatisfactory, since most programs contain loops. Chapter 3 outlines a better algorithm for finding mstates that satisfy the correctness conditions presented in Chapter 2, despite such circularities.

1.9 Jumps.

The go to statement is a common method for constructing loops, although its extravagant use is rightly deplored by many[Dijkstra 1]. A go to causes a few problems for analysis, because it terminates execution of all parts of the program between it and its target. Somehow, the mstate before the go to statement must be transmitted to the target of the go to.

A systematic method is used in this thesis. When a go to statement is executed, a special kind of value, a 'jump', is placed on the run-time stack. Jumps are recognized as special cases and are simply passed outwards through syntactic levels, with the stack and/or the environment suitably trimmed, until the jump finally reaches the clause containing the jump target. Normal execution resumes at the jump target.

Analysis is analogous. When analysing a go to statement, the analyser places a representative for the jump on the mstack. The resulting mstate is recognized as a special case, and is passed outwards, with its mstack and menvironment suitably trimmed, until it reaches the jump target. This is implemented by attaching more than one mstate after some phrases in the program -- one for the normal flow of control, and one for each go to statement that might terminate execution of that phrase by jumping out of it. For this reason, we shall distinguish between 'mstates', which we have been discussing so far, and 'mstate sets', which are disjunctions of mstates. Normally, one of these mstates will correspond to normal flow of control, and the others to paths of control created by <u>go</u> <u>to</u> statements. An mstate set will be considered to describe the state of the run-time machine iff at least one of its mstates does. It is in fact mstate sets that are attached to points in the program.

1.10 Procedures.

Procedures cause severe problems. A procedure behaves like a piece of code that may be inserted at many points throughout the program, including within itself. One might try to analyse procedures by attaching at the beginning of the procedure body an mstate obtained by merging all mstates appearing before calls to the procedure, and placing after each call a copy of the mstate at the end of the procedure. However, this technique destroys information correlated with the point of call, a quite undesirable effect.

For example, consider:

```
proc p = void: skip;
     # end of p, which is the null procedure #
v := 1;
p;
a := v;
v := 2;
p;
a := a + v
```

If the above approach were used, merging would destroy the information that at one call to p, v was 1, but at the other call, v was 2. The mstate at the start of the routine, and also those placed after each call, would have no information about the value of v, despite the fact that v is not altered by p.

We must find some other scheme. A procedure will be analysed, independently of any calls to it, to produce an 'effects directory'. The effects directory describes the ways that the procedure can alter nonlocal variables. Each possible alteration is described by an 'effect'. An effect consists of an identifier, which may be a nonlocal identifier or a formal parameter identifier, a route through the run-time data structure starting at the identifier, and an action performed on the value at the end of the route. The action may be 'set' or 'escape'. If the action is 'set', then the value may be the destination of an assignment. If the action is 'escape', then the value escapes; i.e., it can no longer be represented by an individual mvalue.

Consider the following procedure:

```
proc p = (ref real x, ref ref real y) void:
     # 'x' and 'y' are parameters.
          'x' is a real variable.
          'y' is a variable whose values are real variables
          #
     begin
     ref real z; # the values referred to by z are real
          variables #
```

```
    z := x;
    ref real(z) := 3;
    ref real(y) := 3
    end # end of p #
```

The effects directory of this procedure contains

```
set x
set dereference(y)
```

i. e., the variable x and the variable referred to by y are both assigned to.

The identifiers used in these effects are either the formal parameters of the procedure, or identifiers global to the procedure. Since, as demanded in Chapter 2, procedures may not be actual parameters and may not be referred to by variables, there is no way for a procedure to be called from a point in the program where its global identifiers are unknown. Thus, the effects are in a notation meaningful at every point of call.

In order to compute these effects, each mvalue has associated with it a 'history', which describes all the ways that the run-time machine might be considered to have reached the value represented by the mvalue. A history is a set of 'tracks'. Each track describes one of these ways, and consists of an identifier and a sequence of primitive language operators that describe a route through the run-time data structure. In the above example, the destination of the last assignment would have the history 'dereference(y)'.

These effects are applied to the mstate at each point of call.

Chapter 2. Description of the language and presentation of the consistency conditions.

2.0 Introduction.

'Analysis' is the attaching of an mstate set to each point of a program, by examining the program's source text. These mstates can be used directly for, for example, the constant-propagation optimization (This optimization can be of great value when applied to program text resulting from macro processing or from in-line expansion of procedures). In constructing these mstates, the analysis algorithm also isolates a class of variables, the individual variables, whose interaction with pointer variables is sufficiently simple that they can be treated by conventional optimization techniques. It will likely be found that the majority of variables occurring in ordinary programs will be individual.

These methods will also be of use with other optimizing techniques, since the problem of side effects of procedures is settled, at least for the individual variables. Once this is done, there is sufficient information for most conventional optimization analyses to be performed.

In order to be certain that these mstate sets correctly represent the corresponding states at run time, they are required to satisfy certain conditions, called consistency conditions .

A very simple programming language is used in order to make discussion concrete. One can infer the treatment of other languages from this example. The concrete and abstract syntax of the language is presented, and the semantics of the language is described using an abstract machine.

The reader may well find his mind boggling at the number and complexity of the definitions necessary to define the many concepts in this chapter. If this happens, he is advised to examine the lengthy examples in section 2.4, which provide the concrete details needed by the intuition to grasp the abstract ideas involved. The mstate sets and their precise correspondence to the states of the abstract machine are also presented. Section 2.9 describes the execution of programs and the consistency conditions on mstate sets. A number of theorems are proved, to show that the consistency conditions imply correctness of the mstate sets. The use of the consistency conditions in performing the analysis is explained in Chapter 3.

2.1 Purpose of the simple language, restrictions.

The simple language was designed to illustrate the analysis algorithm. To this end, any language feature that would complicate the analysis without providing additional insight was omitted. The following language features have been provided:

procedures and procedure calls

boolean values

pointers and variables

while -- do -- od

if -- then -- else -- fi

identity declarations

block structure

go to and labels

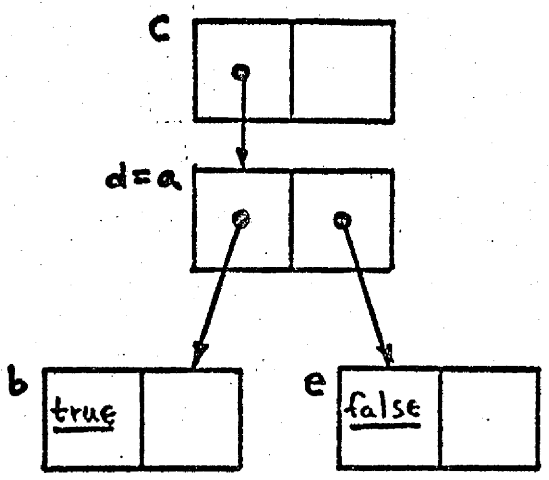
Example.

Here is a sample program:

```
( let makelist =
      ( proc ( a, b, e):
         ( a := b;
            a setlink e;
            b := true;
            e := false;
            c := a));
   let c = gen;
.   let d = gen;
    makelist(d, gen, gen)
)
```

In this program, "makelist" is the identifier of a procedure with three parameters, "a", "b", and "e", and one nonlocal identifier, "c". It chains "a", "b", and "e" into a small tree whose leaves contain the values "true" and "false". "c" and "d" are declared as new variables: the identity declaration for "c"; "let c = gen", declares "c" to possess the value of the expression after the equals sign. In this declaration, that expression is a generator, which creates a new variable by allocating storage and yielding a pointer to it. Thus "c" will possess this new variable. The declaration of "d" is similar.

The variables in this language have two fields each, a 'val' field, and a 'link' field. Each field may contain either a pointer or a boolean value. The val field is set by the assignment operator, and the link field by the 'setlink' operator. The list structure constructed by the call to makelist is:



It is unnecessary to provide additional primitive data types, such as integer, character, etc., since they can be handled like boolean.

Label variables (such as those in PL/1) have not been provided, because they greatly confuse control flow. There is, therefore, no data type '<u>label</u>'.

There is one major restriction. Procedures cannot have procedures as parameters, nor may procedures be referred to by variables. If these were permitted, global identifiers used in a procedure call might be unknown at the point of call. The following ALGOL 60 program violates this restriction:

```
begin
   procedure p(q1);
      procedure q1;
      q1;
   begin
```

```
            integer x;
            procedure q;
               x := 6;
            p (q)
         end
      end
```

The procedure 'p' is one which, when called, calls its argument, 'q1', which is a procedure. However, 'p' is called from a block where an extra identifier, 'x', is known. 'q', the actual parameter, can, and does, know this extra identifier.

At the syntactic point where q1 (which equals q at run time) is called from p, this nonlocal identifier x is not known, even though it is still meaningful at a lower level on the run-time stack. It is difficult to summarize the effects of calling 'q1' exclusively in terms of identifiers known at the point of call. This is why the restriction was imposed.

Thus, in the simple programming language, we have reasonable control structure, adequate for most normal programming, and we have pointers, but we have a severe scarcity of data types.

It is not intended to define the language with the rigour that would be required of an international standard, since that is not necessary for this exposition.

2.2 Syntax of the simple programming language.

2.2.1 Abstract syntax.

This section contains the abstract syntax of the simple language, in a notation reminiscent of that used by Peter Landin [Landin 1].

A program is an expression in which each identifier is defined in a declaration or as a formal parameter. We say that an identifier 'identifies' its defining declaration.

In this syntax, 'identifier', 'label', 'completer', 'true', and 'false' are primitive notions in terms of which the others are defined.

An expression is
    a serial (sometimes called a block), or
    a conditional, or
    a call, or
    an assignment, or
    a link setting, or
    an iteration, or
    a go to, or
    a dereferencing, or
    a delinking, or
    a generator, or
    an identifier, or

a routine denotation, or

a boolean denotation.

A declaration has

a formal parameter, which is an identifier, and

an actual parameter, which is an expression.

Without loss of generality, it is required that all declarations and formal parameters declare different identifiers.

Examples:

```
      let makelist =
          ( proc ( a, b, e):
              ( a := b;
                  a setlink e;
                  b := true;
                  e := false;
                  c := a));
```

This declaration declares 'makelist' to possess a routine.

```
   let d = gen
```

This declaration declares 'd' to possess a new variable.

A serial is a sequence of phrases, label definitions, and completers.

Example:

```
let c = gen; if e then go to l fi; e := false; e. l: c :=
     true; goto l
```

A completer causes premature termination of execution of a serial. The value of the serial is the value of the phrase before the completer. In the above example, the value yielded by the serial is the variable 'e', unless the completer is bypassed by 'go to l'. If the completer is bypassed, this particular serial will loop.

A phrase is a declaration or an expression.

A conditional has
- a condition, which is an expression,
- a then part, which is an expression, and
- an else part, which is an expression.

Example:

```
if a then b else c fi
```

A call has
- a function, which is an expression, and
- a sequence of actual parameters, which are expressions.

Example:

makelist(d, gen)

if foo then makelist else destroylist fi (d, gen)

Note that the choice of which procedure to call may be made dynamically.

An assignment has

a destination, which is an expression, and

a source, which is an expression.

Example: a := b

In this example, 'a' is the destination, and 'b' the source.

A link setting has

a destination, which is an expression, and

a source, which is an expression.

Example; a setlink b

Assignments and link settings are identical, except that assignment changes the 'val' field of a variable, whereas link setting changes the 'link' field.

An iteration has

a condition, which is an expression, and

a body, which is an expression.

Example: while b do c od

A go to has a label.

Example: <u>go</u> <u>to</u> 1

A dereferencing has an argument, which is an expression.

Example: <u>val</u> b

'<u>val</u>' is used to extract a value that has been assigned to a variable by ':='.

A delinking has an argument, which is an expression.

Example: <u>delink</u> b

'<u>delink</u>' is used to extract a value that has been assigned to a variable by '<u>setlink</u>'.

A generator is used to produce a new variable. There is only one way of writing a generator.

Example: <u>gen</u>

A boolean denotation may be

<u>true</u>, or

<u>false</u>.

A routine denotation (the text of a routine) has

a sequence of formal parameters, which are identifiers, and

a body, which is an expression.

Example:

```
(proc (a, b, e): (a := b; a setlink e; b := true; e :=
    false; c := a))
```

2.2.2 Concrete syntax.

Abstract syntax is convenient when one discusses semantics: it eliminates a great deal of confusing verbiage about positions of semicolons, commas, etc. In order to be able to write programs on paper, however, some concrete syntax must be selected for representing abstract programs.

This section contains the concrete syntax of the simple language.

```
<program>
     ::= <expression>
<expression>
     ::= <serial>
      | <phrase>
<serial>
     ::= <serial> ; <phrase>
      | <serial> ; <label place>
      | <serial> <completer> <phrase>
      | <phrase> ; <phrase>
      | <phrase> ; <label place>
      | <phrase> <completer> <phrase>
<label definition>
      ::= <identifier> :
```

```
<completer>
     ::= .
<phrase>
     ::= <unit>
     | <declaration>
<declaration>
     ::= let <identifier> = <unit>
<unit>
     ::= <secondary>
     | <assignment>
     | <link setting>
<assignment>
     ::= <primary> := <unit>
<link setting>
     ::= <primary> setlink <unit>
<secondary>
     ::= <primary>
     | <dereferencing>
     | <delinking>
     | <go to>
     | <call>
<dereferencing>
     ::= val <secondary>
```

```
<delinking>
     ::= delink <secondary>
<go to>
     ::= go <identifier>
<call>
     ::= <primary> <actual parameters>
<actual parameters>
     ::= ( <unit list> )
     | ( )
<unit list>
     ::= <unit>
     | <unit list> , <unit>
<primary>
     ::= <conditional>
     | <loop>
     | <denotation>
     | <closure>
     | <identifier>
     | <generator>
<conditional>
     ::= if <expression> then <expression> else <expression> fi
<loop>
     ::= while <expression> do <expression> od
```

```
<denotation>
     ::= <boolean denotation>
     | <routine denotation>
<boolean denotation>
     ::= true
     | false
<routine denotation>
     ::= ( proc <parameters> : <unit> )
<parameters>
     ::= <empty>
     | ( <parameter list> )
<parameter list>
     ::= <identifier>
     | <parameter list> , <identifier>
<closure>
     ::= ( <expression> )
<generator>
     ::= gen
```

2.3 The state of the machine.

The semantics of the simple programming language will be described by specifying its interpretation by an abstract machine [Landin 1, Lucas 1]. This section describes the architecture of the abstract machine. The semantics is described in sections 2.9.x.1, x = 1,...,11.

When a program is being executed, the machine passes through a succession of states, each determined by the previous state and the program. In this section, we describe the state of the abstract machine. The state of the machine, as described below, does not include control structure; that is handled separately.

A state has

- a stack of levels,
- a collection of values (which contains all the values contained in all parts of the state), and
- two mappings, called 'refer' and 'link', from a subset of the collection of values into the collection of values.

Each level corresponds to a procedure currently being executed, and contains its parameters, local and nonlocal identifiers, and other information.

'refer' and 'link' model the two fields of a variable, and will be described presently.

A level has

a stack of values (commonly called the stack of compiler-generated temporaries), and

an environment.

An environment is a set of pairs, pairing identifiers with values.

The environment holds the values of local and nonlocal identifiers. The set of pairs can be considered as a mapping which maps each identifier to its value. An identifier is said to 'possess' its value.

We shall often speak of 'the top of the stack', meaning the top of the value stack at the top of the level stack.

A value is

the undefined value, or

a reference value, or

a routine value, or

a boolean value, or

a jump.

The undefined value is used whenever a value has not yet been provided; for example, newly created variables refer to the undefined value.

Reference values have been provided in the simple language in order to model both the names (references) of ALGOL 68 and the two-field cells of LISP. For the ALGOL 68 analogue, one can consider 'refer' to model the mapping that maps a variable onto its current value, or maps a pointer onto the value it points to. For the LISP analogue, the 'refer' and 'link' mappings model the mappings 'car' and 'cdr'.

A reference value may refer and/or link to a value.

In the rest of this thesis, we shall often discuss the 'refer' mapping only, since the 'link' mapping is entirely analogous.

A routine value has

a routine denotation, and

an environment.

The routine denotation contains the code to be executed when the procedure is called, and the environment associates the nonlocal identifiers with their values.

A boolean value is "_true_" or "_false_".

A jump has

a label.

When a go to statement is executed, it leaves a special kind of value, a jump, on the stack.

2.4 Examples.

This section contains a number of examples intended to help explicate some of the technical terms used in this thesis.

2.4.1 Straight line code.

We shall consider the following program:

```
(
   let xx = gen;
      xx := gen;
   val xx := true;
   let yy = gen;
   yy := val xx;
   val xx := false;
   xx
)
```

This program is similar to the one in section 1.6. Its execution is just like that in section 1.6, except that "true" is used instead of "6", and "false" instead of "5". Here is a picture of the state of the abstract machine at the moment when both operands of the assignment 'yy := val xx' have been executed, but the assignment itself has not been performed.

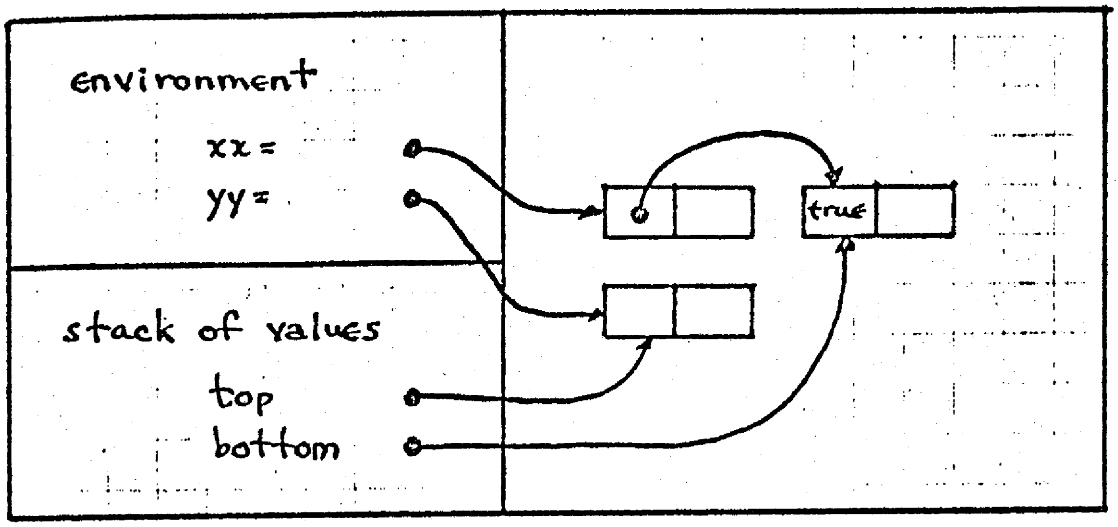


Fig 2.4.1(1)

On the stack have been placed, in order, the value of the source '<u>val</u> xx' of the assignment, and the value of the destination 'yy' of the assignment. All that remains to do in order to complete the assignment is to copy a pointer into the first field of the destination, and to replace the top two values on the stack by the value of the assignment (which is the destination), thus:

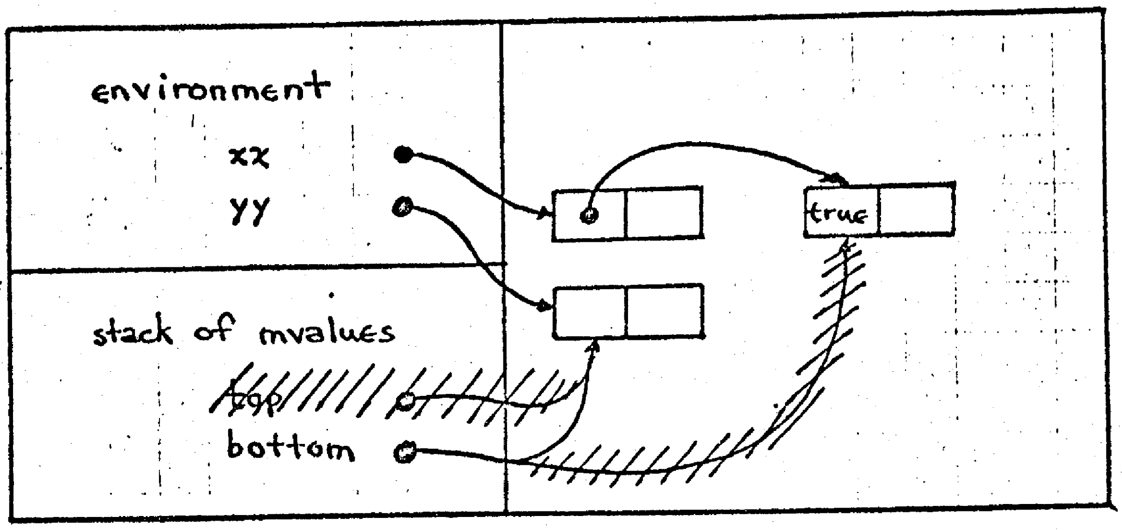


Fig 2.4.1(2)

According to the rules for analysis, mstate sets are prefixed and postfixed to each phrase in the program, in the places indicated by the pillows below:

```
■(■
   ■let xx = ■gen■■;
   ■■xx■ := ■gen■■;
   ■■val ■xx■■ := ■true■■;
   ■let yy = ■gen■■;
   ■■yy■ := ■val ■xx■■■;
   ■■val ■xx■■ := ■false■■;
   ■xx■
■)■
```

Mstates are compile-time objects which contain information known to the compiler about run-time machine states. The formal definition of mstates is given in section 2.5.

For obvious reasons, all of these mstates will not be drawn. Instead, let us examine one of them, namely, that indicated by the lozenge below:

```
■▪yy□ := ▪val ▪xx▪▪▪;
```

In an example as simple as this one, each mstate set has only one mstate. Because the program is so simple, everything about it can easily be discovered by the analyser. The picture of an mstate will be identical to that of the state, except for the prefix 'm'. Here is the mstate (at the lozenge) that represents the machine state:

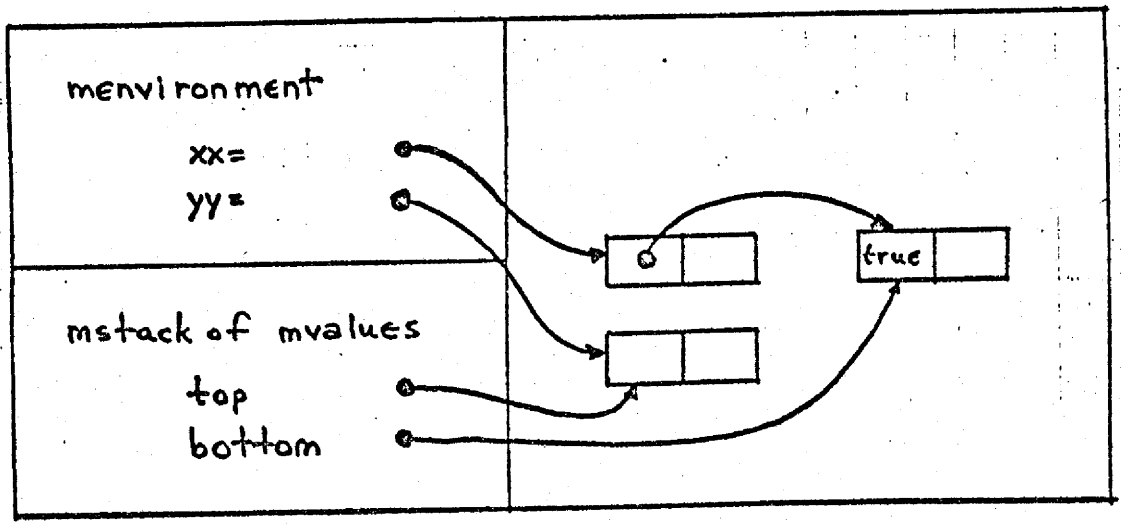


Fig 2.4.1(3)

'xx' and 'yy' are identifiers, which mpossess two mvalues. The mvalue mpossessed by 'xx' is an mvariable, and mrefers to another mvariable, which mrefers to the mvalue 'true'.

The parts of the mstate represent the corresponding parts of the machine state in figure 2.4.1(1).

2.4.2 A loop.

The purpose of the following routine is to search the linked list 'list' for a cell whose val field does not contain 'true'. When one is found, the previous cell in the list is returned in the variable 'v' as result. There must be at least one such cell for the routine to terminate. If the first cell is the one found, a new cell must be created to serve as previous cell. In addition, all cells before the one found have their 'val' fields set to 'false'.

```
(proc (list, v):
   (
      let current = gen;
      let previous = gen;
      let newcell = gen;
      newcell := false; newcell linkset list;
      previous := newcell;
      current := list;
      while val val current
      do val current := false;
         previous := val current;
         current := link val current
      od;
      v := val previous
   )
)
```

The following mstate is the only one in the mstate set prefixed to the while statement:

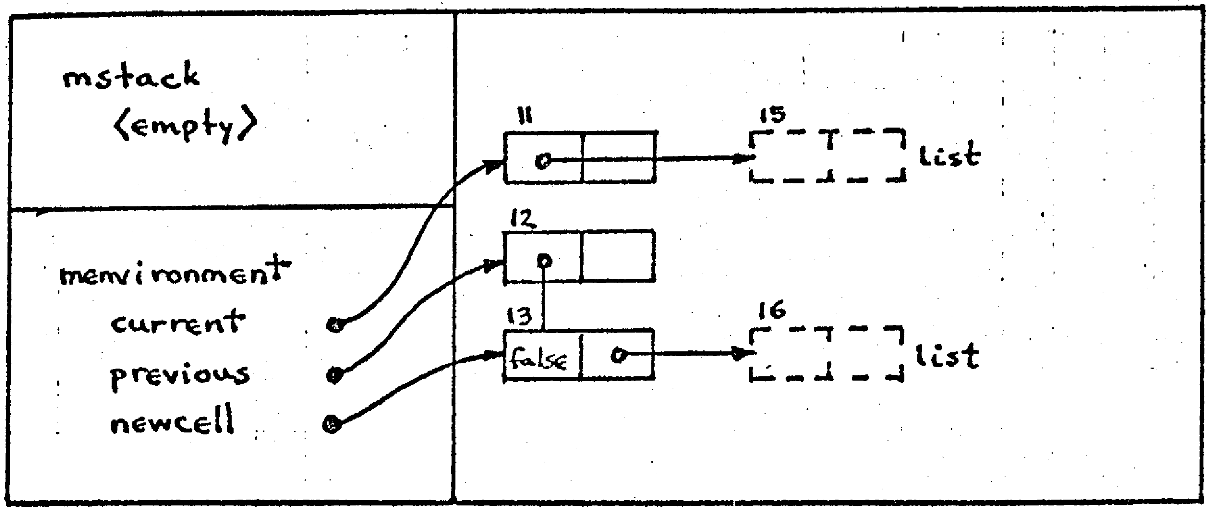


Fig 2.4.2(1)

In this mstate, two mvalues represent the variable 'list'. The analyser is unfortunately too stupid to recognize that they represent the same variable. Fortunately, it knows that it is too stupid, and therefore the two mvalues are nonindividual. This is indicated by the dotted lines. Each mvalue has a history beside it; only the nonindividual mvalues in this mstate have a nontrivial history, however. This history consists of a single track. This single track has a start, 'list', and the empty sequence of operators.

```
                 list
L---------J  L-----J
 operators    start
L-----------------J
       track
L-----------------J
      history
```

The history indicates that the mvalue may represent, at run time, a value which is possessed by the nonlocal identifier 'list'. Notice that the other mvalues have no histories; their origins are purely local. Since histories are used by the analyser only for the discovery of the nonlocal effects of a procedure, there is no point in providing any history.

After computing mstate sets for all places in the program up to the start of the loop, the analyser is faced with a problem: two paths of control converge. It appears that it cannot find the mstate set to prefix to the condition of the loop without knowing the mstate sets along both of these paths. Furthermore, the mstate sets from the path of control from the end of the loop will not be known until the entire loop has been analysed.

The solution used is this:

The analyser blithely ignores the extra path of control until it has an mstate set for it. When it finds one, the analyser performs the analysis of the loop again. If this causes the mstate set on the extra path to change, it performs it yet again. Iteration continues until the mstate sets cease to change.

Therefore, as a first try, the mstate prefixed to the condition is the same as that prefixed to the loop, the one in figure 2.4.2(1).

Mstate sets are then computed for the rest of the loop, giving the following for the end of the body of the loop:

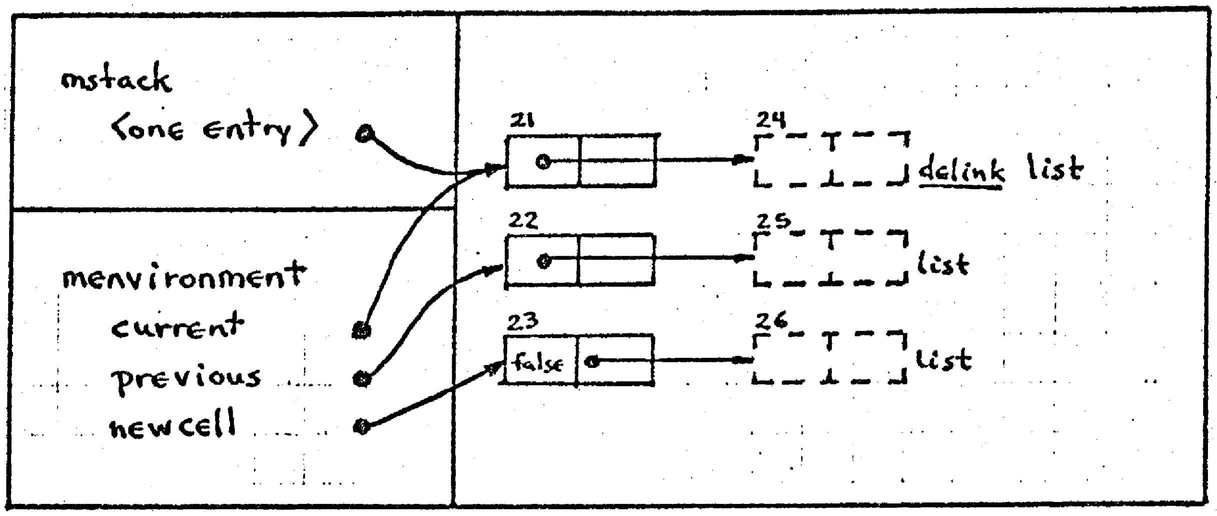


Fig 2.4.2(2)

Notice the history of mvalue 24:

$$\underbrace{\underline{\text{delink}}}_{\substack{\text{sequence}\\\text{with only}\\\text{one operator}}}\quad\underbrace{\text{list}}_{\text{start}}$$

The 'delink' results from delinking the old value of 'current' to obtain a new one.

This mstate must now be used in constructing a new mstate set for the start of the loop. The new mstate set is computed using an algorithm called 'merging', which is defined precisely in section 2.10. 'Merging' is the construction, given two or more 'original' mstates, of a new mstate that can represent all the machine states that either of the original mstates can. In this example, the mstates merged are the one in figure 2.4.2(2) after one entry has been removed from its mstack, and the one in figure 2.4.2(1). The resulting merger has one mstate:

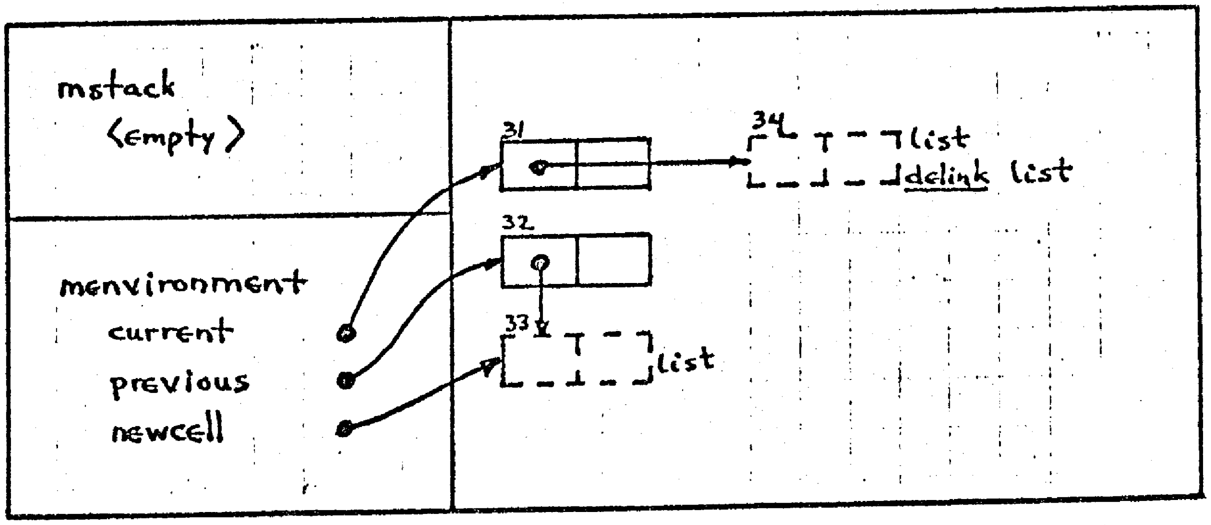


Fig 2.4.2(3)

Mvalues in the old and new mstates correspond in the following manner:

from: Fig. 2.4.2(1)	Fig 2.4.2(2)	to: Fig 2.4.2(3)
11	21	31
12	22	32

13	23,25	33
15	24	34
16	26	absent

This merging causes the following effect to be placed in the effects directory of the routine:

$$\underbrace{\underbrace{\underline{\text{escape}}}_{\text{action}}\quad \underbrace{\underbrace{\qquad\qquad}_{\substack{\text{sequence}\\ \text{of operators}}}\ \underbrace{\text{list}}_{\text{start}}}_{\text{track}}}_{\text{effect}}$$

(Effects are further discussed in the next example. This effect is mentioned here for completeness.) This effect appears because mvalues 16 and 26 have no corresponding mvalue in 2.4.2(3) (see condition 2.10.11).

Using this new mstate, new mstates are constructed for all the phrases in the loop, producing the following to suffix to the body of the loop:

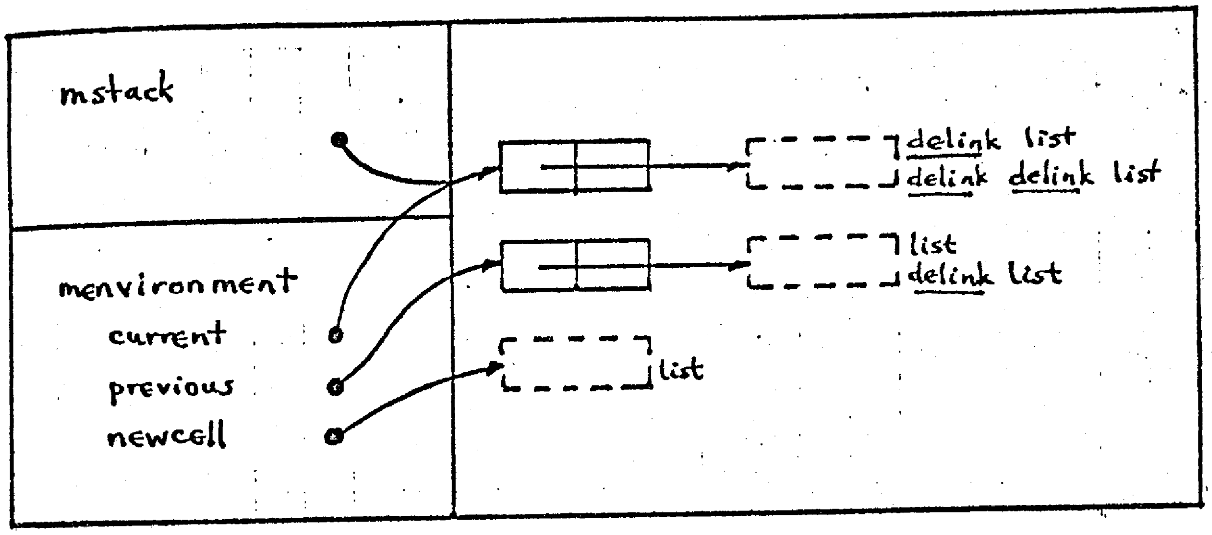


Fig 2.4.2(4)

The history of mvalue 45 has two tracks:

<u>delink</u> list

<u>delink</u> <u>delink</u> list

Merging this new mstate (after deleting the mstack entry) with the one prefixed to the loop (figure 2.4.2(1)), we get:

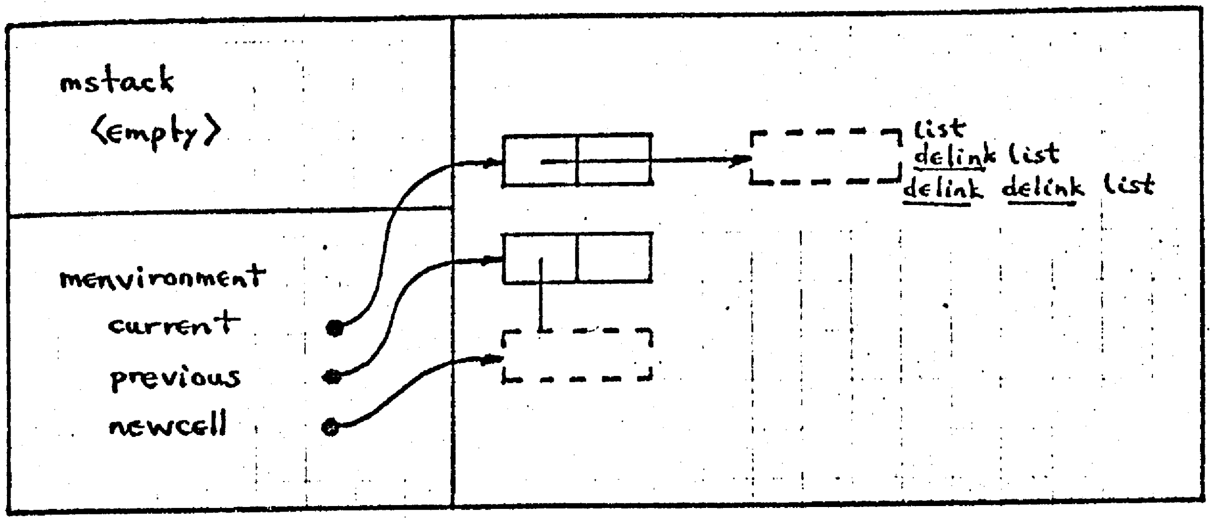


Fig 2.4.2(5)

Except for the history of mvalue 54, this is remarkably similar to 2.4.2(3). It is plain that the analyser could continue forever, merely adding another track to the history of mvalue 54 each time around the loop. Instead, it takes special measures, and sooner or later makes the giant leap to an infinite history. How this is done is described in Chapter 3. The history becomes the regular language defined by the regular expression

<u>delink</u> *   list

The history is the infinite set of tracks:

list

<u>delink</u> list

<u>delink</u> <u>delink</u> list

<u>delink</u> <u>delink</u> <u>delink</u> list

...

2.4.3 Effects.

Let us discuss one more matter regarding this procedure. The procedure has nonlocal effects, namely, setting the fields of the linked list to _false_ (strictly, speaking, it also causes the analyser to forget a variable. This is also an effect). Knowledge of these side effects must be available to the analyser when it analyses any program that calls this procedure.

This is accomplished by an 'effects directory', which summarizes the nonlocal effects of the procedure. These include straightforward effects, such as assignment and linksetting, and also subtler ones, such as 'escaping'. 'Escaping' is, strictly speaking, an artifact of the analysis algorithm. An effect that announces that some variable escapes is a warning to the analyser that it can no longer tell when that variable is used. Mvariables representing such variables are called 'nonindividual'.

To describe an effect, it is necessary to specify what is done, and what it is done to. What is done is specified by the 'action' of an effect, which is '_set_', '_setlink_', or '_escape_'. What it is done to is specified by a 'track'. This track identifies the object of the action by giving an algorithm by which the object may be found. This algorithm is a sequence of operations (each of which is '_dereference_' or '_delink_') applied to some identifier.

Such tracks are obtained from the history of the mvalue to which the action is performed. This history contains a complete record of all dereferencing and delinking used within the procedure to reach the value affected. Since the procedure that performs the action could locate the value by a sequence of operations, so can the caller of the procedure, and therefore the analyser can use the track to find the value when it is analysing the call to the procedure.

Let us reconsider the routine of section 2.4.2. Its effects arise from three causes:

1. An escape results from merging, as already mentioned in section 2.4.2.
2. A class of set effects results from the assignment

   val val current := false.
3. The effect 'set v' results from the assignment

   v := val previous.

Let us examine the second cause.

The mvalue that represents the destination 'val val current' of the assignment is:

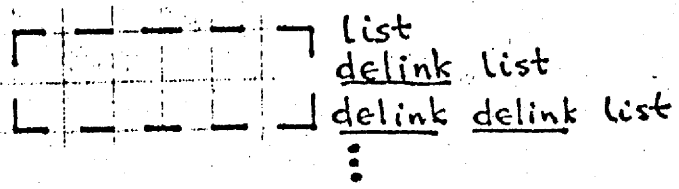


Fig 2.4.2(6)

Since the variable it represents is assigned to, we obtain the effects:

set list

set delink list

set delink delink list

...

This can be summarized finitely as

set delink* list

Thus, the effects of the routine are the following:

escape list

set delink * list

set v

2.4.4 Use of effects.

We shall now consider a program that calls the routine of section 2.4.2:

```
(
   let p =
      (proc (list, v):
         (
            let current = gen;
            let previous = gen;
            let newcell = gen;
            newcell := false; newcell linkset list;
            previous := newcell;
            current := list;
            while val val current
            do val current := false;
               previous := val current;
               current := link val current
            od;
            v := val previous
         )
      );

   let a = gen;
   let b = gen;
let c = gen;
let d = gen;

a := true; b := true; c := false; d := true;

a setlink b;
b setlink c;
c setlink d;

let w = gen;

p(a, w) )
```

This program constructs a linked list to pass to 'p'. The mstate set postfixed to the 'p' in the call corresponds to the moment of execution after the parameters and the function have been executed, but just before the routine is actually invoked. The mstate is:

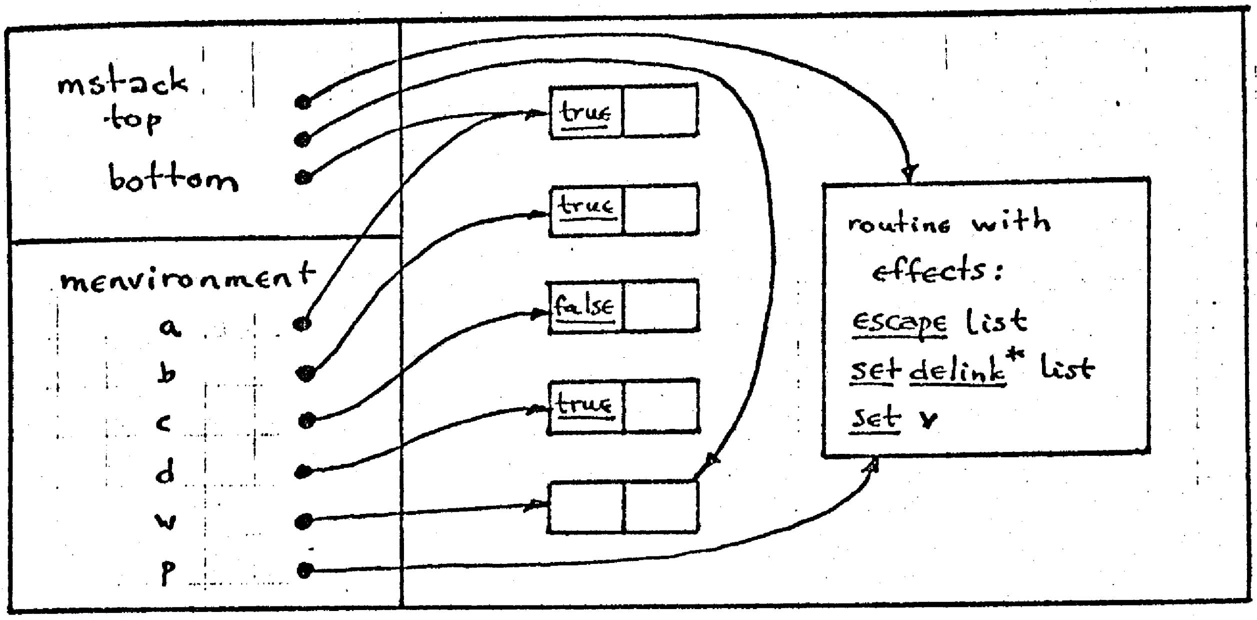


Fig 2.4.4(1)

To compute the mstate postfixed to the call, we must apply the effects of 'p'. These were

<u>set</u> v

<u>escape</u> list

<u>set</u> list

<u>set</u> <u>delink</u> list

<u>set</u> <u>delink</u> <u>delink</u> list

...

Let us consider the effects with action 'set' first.

Consider '<u>set</u> v'. 'v' is a formal parameter, which corresponds to the actual parameter 'w'. 'w' mpossesses an mvalue which will represent the variable to which 'p' may assign a value. This mvalue acquires a 'set' mark (see figure 2.4.4(2)).

The 'set' mark will indicate that the value represented by the mvalue may be assigned to.

Similarly, 'list' corresponds to 'a', and so the mvalue mpossessed by 'a' acquires a 'set' mark.

The effect '<u>set</u> <u>delink</u> l' requires the following. One starts with the mvalue represented by 'list' (because of the 'list'), one finds the mvalue it mlinks to (because of the '<u>delink</u>'), and this mvalue is the one which acquires the '<u>set</u>' mark.

The effect '<u>set</u> <u>delink</u> <u>delink</u> l' requires the following. One starts with the mvalue represented by 'list' (because of the 'list'), one finds the mvalue it mlinks to (because of the '<u>delink</u>'), one finds the mvalue it mlinks to (because of the '<u>delink</u>'), and this mvalue is the one which acquires the '<u>set</u>' mark.

...

This infinite sequence of effects would seem to require an infinite amount of analysis. However, sooner or later, there are no more mvalues left in the mlinked list for the sequence of <u>delink</u>s to be used on, and then the analyser can stop.

Furthermore, the effect '<u>escape</u> list ' causes the mvalue mpossessed by 'a' to acquire an 'escape' mark.

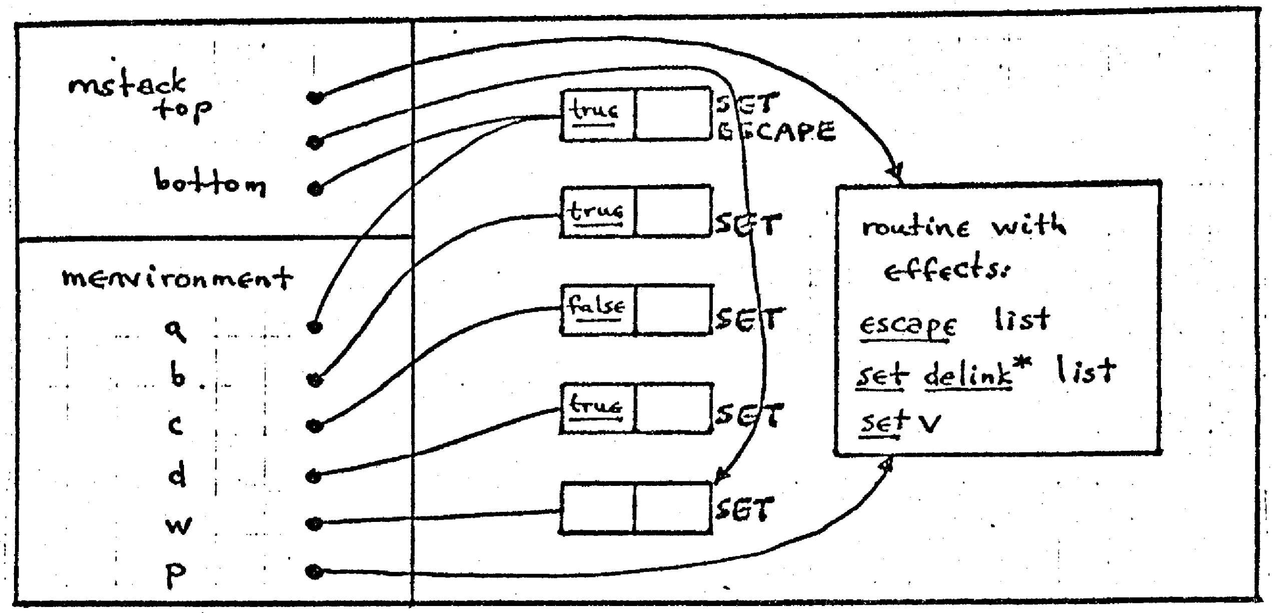


Fig 2.4.4(2)

Now that the necessary mvalues have been located, the actions are performed. Each 'set' causes the '<u>val</u>' fields to be forgotten:

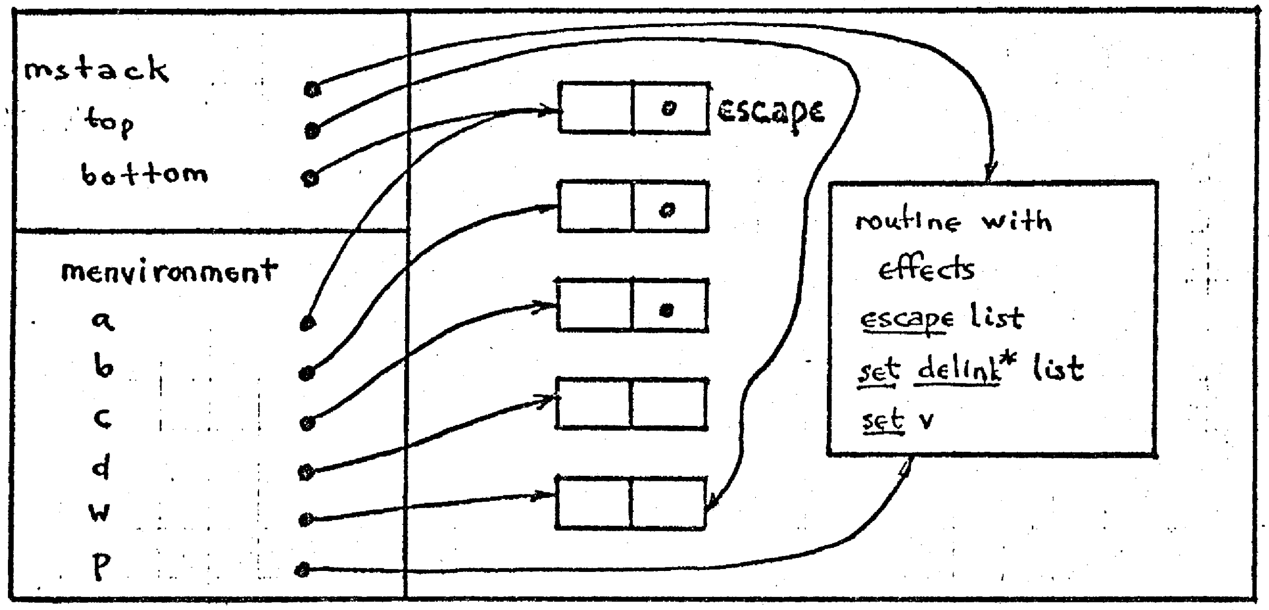


Fig 2.4.4(3)

Each 'escape' causes the mvalue to cease to be individual, causes it no longer to mlink or mrefer to other mvalues, and causes any mvalues formerly mlinked to or mreferred to also to escape.

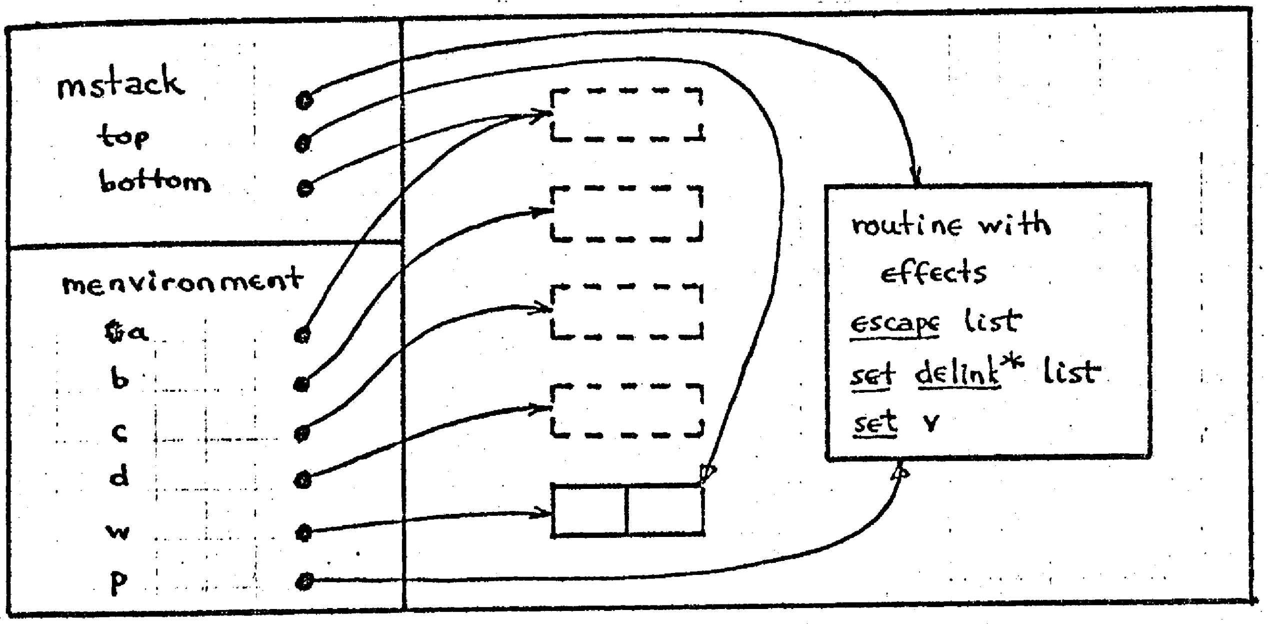


Fig 2.4.4(4)

After deleting the mstack entries corresponding to the parameters and 'p', we have the mstate of the mstate set postfixed to the call.

2.5 Mstate sets.

The analysis algorithm presented in this thesis attaches to each point in the program text an mstate set, which is a partial picture of the state of the abstract machine. Before describing the algorithm, it is necessary to define mstate sets in a rigorous manner, not merely in the informal manner of Chapter 1. The correspondence between mstate sets and states is defined in section 2.6.

An mstate is a (partial) description of that part of the state of the machine accessible from a single level. It thus contains representatives for some values, for the local part of the environment, and for the temporaries stack. Furthermore, it keeps records, called histories, of the means whereby these represented values might have been obtained.

An mstate set consists of a number of 'mstates'.

An mstate consists of

a collection of mvalues,

an mstack, which is a stack of mvalues,

an menvironment, which maps identifiers into the collection of mvalues, and

a predicate, 'individual', specifying which mvalues of type reference are individual.

For the meaning of individuality, see section 1.4. Mvalues representing values freshly created by generators are individual. The analyser attempts to keep as many mvalues as possible individual.

An mvalue consists of

a history, and

a type, which may be 'boolean', 'jump', 'routine', 'reference', or 'unknown'.

A history is a set of 'tracks'.

A history describes all the nonlocal ways that a program might be considered to arrive at a value represented by an mvalue. Each track describes one of these ways.

A track has

a start, which is an identifier, and

a sequence of operations, each of which must be 'link' or 'val'.

The program, in reaching a value, might have started with the 'start', and applied each of the operations in order, until it reached the value.

A boolean mvalue may (but need not) have one of the attributes 'true' or 'false'.

If it has neither attribute, it indicates that the analyser cannot discern whether the mvalue represents the run-time value '_true_' or '_false_'.

A jump mvalue has a label (the jump target).

An mvalue of type 'routine' has an effects directory (see below). This will describe the possible effects of the routine value it represents.

If an mvalue of type 'reference' is not null and is individual, then it may 'mrefer' or 'mlink' to some mvalue.

An unknown mvalue is used when nothing is known about the value it represents.

An effects directory is a set of effects.

An effect consists of

a track, and

an action, which may be 'set' or 'escape'.

The difference between an effects directory and a history is these actions. 'Set' ('linkset') means that the procedure can assign (linkset) to the variable at the end of the track.

'Escape' means that the procedure can perform some deed that makes it impossible for the analyser to continue treating the value as individual.

Tracks might be considered as the footprints left in the data structure by the program in reaching a value. However, instead of rigorously adding overhead to the run-time system to maintain tracks, they are computed as well as is possible at compile time. Thus, a track becomes a way, starting from the 'start' and applying the operations in order, that the analyser thinks the run-time machine might arrive at a value represented by an mvalue.

Tracks are written on paper in the notation usually used for a sequence of unary prefix operators applied to an operand. The operators are written first, in reverse order, followed by the operand. We shall speak of the 'first' and 'last' operators in a track in the order of application, not the order written. Thus, the first (applied) operator will be the textually last one.

Example: '<u>val</u> <u>val</u> i' might be a track of an mvalue representing a value obtained by twice dereferencing the value possessed by the identifier 'i'.

Example: The following declaration is similar to one in section 1.10.

```
let p = (proc (x, y):
   (let z = gen;
   z := x;
   val z := true;
   val y := true) )
```

The destination, 'z', of the assignment 'z := x', has an empty history. This is because it is a local variable, and is not obtained in any nonlocal manner.

The destination of the next assignment will have the history 'x'. This indicates that the value of the destination might be the value possessed by the parameter 'x'. Note that the analyser is capable of distinguishing that 'x' is involved here, even though it is not explicitly mentioned in the statement. There will be an effect 'set(x)' in the effects directory of p, since 'x' is (effectively) the destination of an assignment.

The destination of the third assignment will have the history 'val y', since it is obtained by dereferencing the value possessed by the parameter 'y'. There will be an effect 'set(val(y))' in the effects directory of p.

The effects directory of 'p' is

'set(x)'

'set(val(y))'.

2.6 Precise correspondence between mstate sets and states.

2.6.1 Specifications.

The mstate set can be considered to be a statement about the state. This section describes the precise interpretation of the mstate set. The proofs in the latter part of this chapter will establish inductively that the mstate set associated with a point in the program will be a true statement each time that control passes that point in the program.

At any instant during program execution, the program will have one or more levels on its level stack. Each level corresponds to a procedure being executed (except the bottom level, which corresponds to the so-called main program). Each point of program execution (one at each level) will have a 'current' mstate set attached to it. Since an mstate describes the state from the viewpoint of a single level, we require that each mstate set be a correct description at its own level, i.e., using the environment and stack that belongs to its level.

There may be many mstates in an mstate set, each a possible description of some level of the state. Because of practical limitations that will be discussed in Chapter 3, however, we shall have at most one mstate for the normal flow of control, and at most one for each label. We require that at least one mstate (the one corresponding to the flow of control) hold at any time.

An mstate set is correct at a level of a state iff at least one of its constituent mstates is correct at that level of the state.

An mstate is correct at a level iff the following conditions hold:

1. There is a relation between the mstack entries and mvalues of the mstate, and the stack entries and values of the state. We denote this relation by "represents".

2. The i-th stack entry of the mstack of mvalues represents the i-th stack entry of the stack of values at the level.

3. For every mvalue v there exists a value w such that v represents w.

4. If v is an individual mvalue, and v represents u and w, then u = w .

Let the mvalue v represent the value v', and the mvalue w represent the value w'. Then:

5. If v is individual, and w is an other mvalue, then v' ≠ w'.

6. If v mrefers to w, then v' refers to w'.

7. If v mlinks to w, then v' links to w'.

8. If v is of type boolean (jump, routine, reference), then v' is of type boolean (jump, routine, reference).

9. If v is <u>true</u>, then v' is <u>true</u>.

10. If v is <u>false</u>, then v' is <u>false</u>.

11. If v is a jump mvalue, then v' is a jump with the same label.

12. If v is of type 'routine', then v' consists of a routine denotation and an environment. The effects directory of v must contain the effects directory of the routine denotation.

<u>Note</u>: We shall often speak of an individual or nonindividual value, meaning a value represented or not represented by an individual mvalue (see section 1.4).

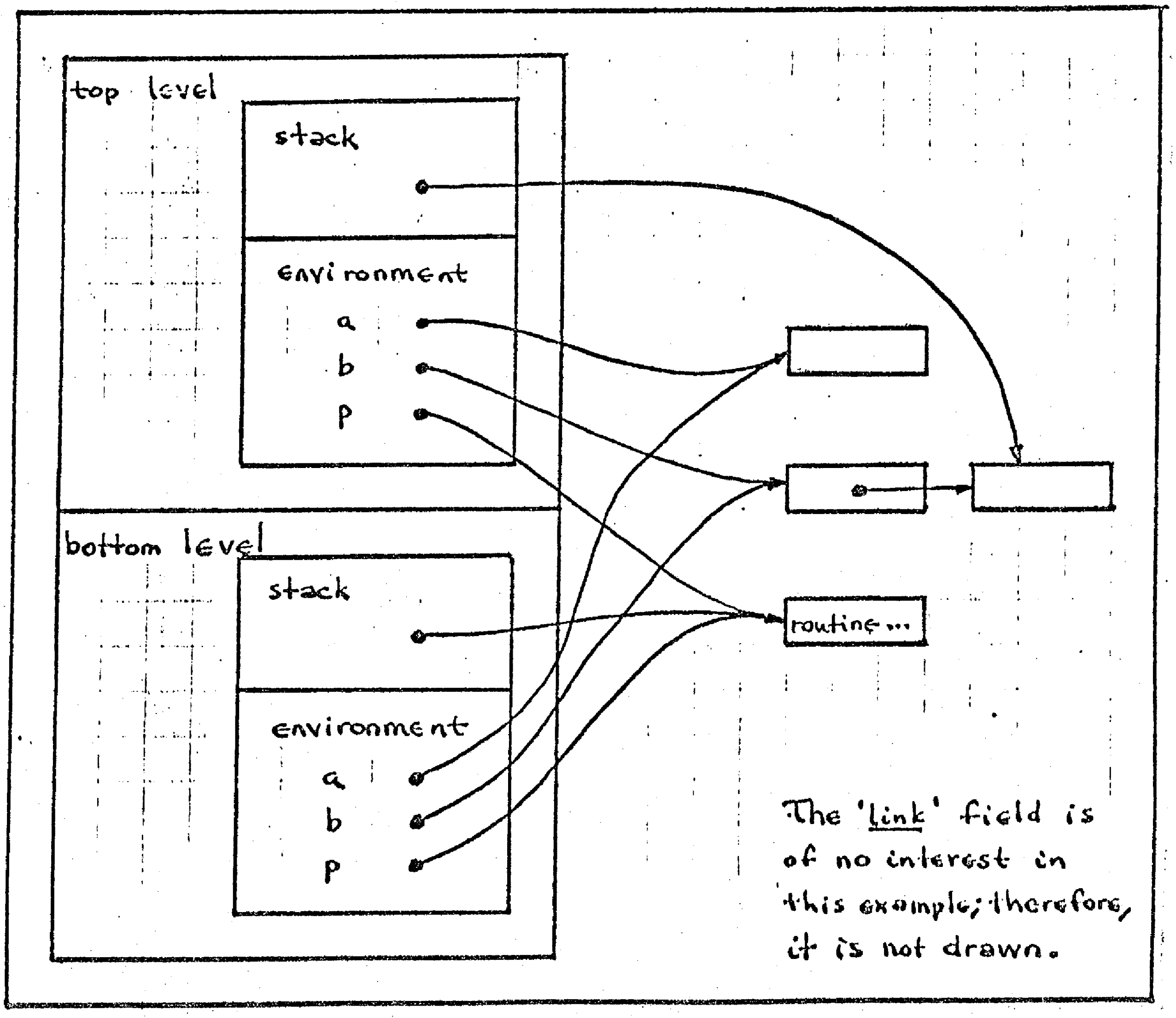
2.6.2 Example.

Consider the following program.

```
                ( let a = gen;
                  let b = gen;
                  let p = ( proc: a := ( val b
(1) ------------------------------------------> *
                                                  ));
                  b := gen;
(2) --------------> *
                  p()
(3) --------------> *
                )
```

2.6.2.1 Execution.

The state of the abstract machine when execution reaches point (1) during execution of the call p() is:


top level
stack
environment
a
b
p
bottom level
stack
environment
a
b
p
routine ...
The 'link' field is of no interest in this example; therefore, it is not drawn.

Let us examine how this state arises. Execution of the program is started. The program contains local declarations of a, b, and p. The identifiers a and b are each made to possess freshly created variables; hence, we have possession pointers from a and b on the bottom level to reference values. The boxes in the diagram for these reference values can be considered as pieces of storage, and a and b possess pointers (these pointers are of course identified with the pieces of storage as usual). 'p' is made to possess a routine. The body of 'p' is not executed at this time. The variable possessed by b is then assigned another freshly created variable, and so the piece of storage possessed by b acquires a pointer to yet another piece of storage.

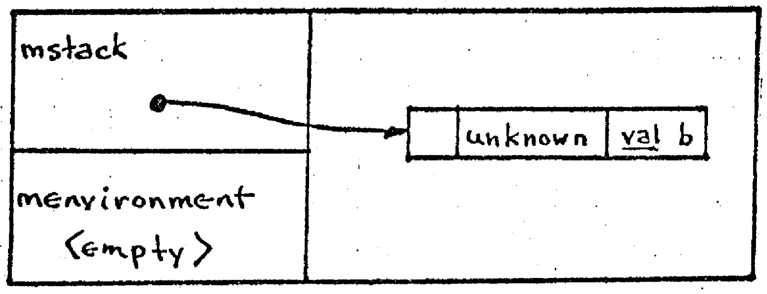
The routine p is then called with no parameters. This produces the top level. The environment of the routine is that of the bottom level; this is copied and becomes the environment at the top level also. Since p contains no local identifiers itself, no new identifiers are added at the top level. At the moment at which the snapshot is taken, execution is at point (1) in the middle of an assignment. At intermediate steps in executing the assignment, the variable possessed by b has been stacked, and then this variable has been dereferenced.

2.6.2.2 Analysis.

In the following diagrams, boxes will represent mvalues. A box will contain 'I' on the left if its mvalue is individual. In the centre of the box is the type of the mvalue, and on the right is the history.

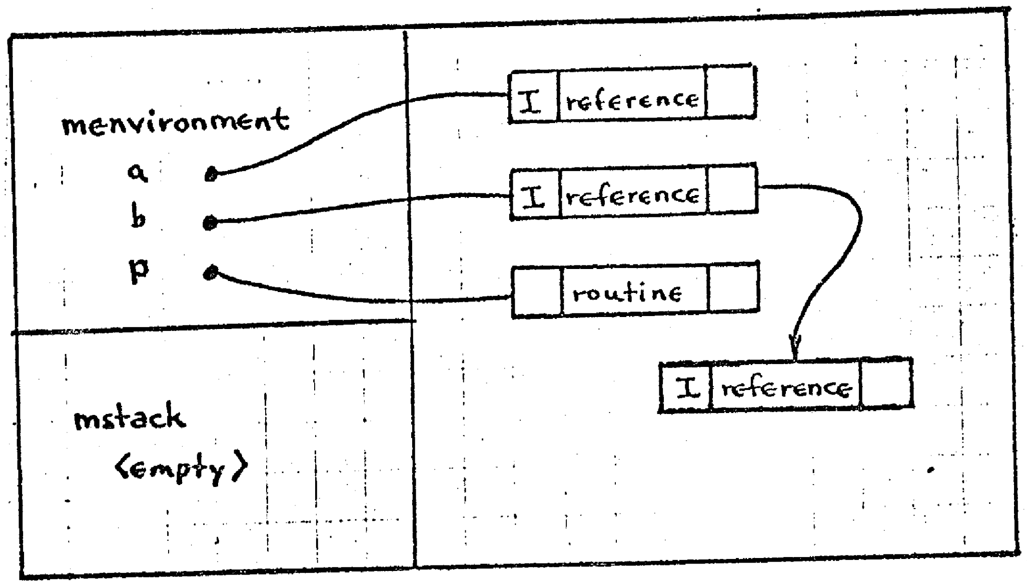
The following mstates might be attached at points (1), (2), and (3) in the program text.

(1)



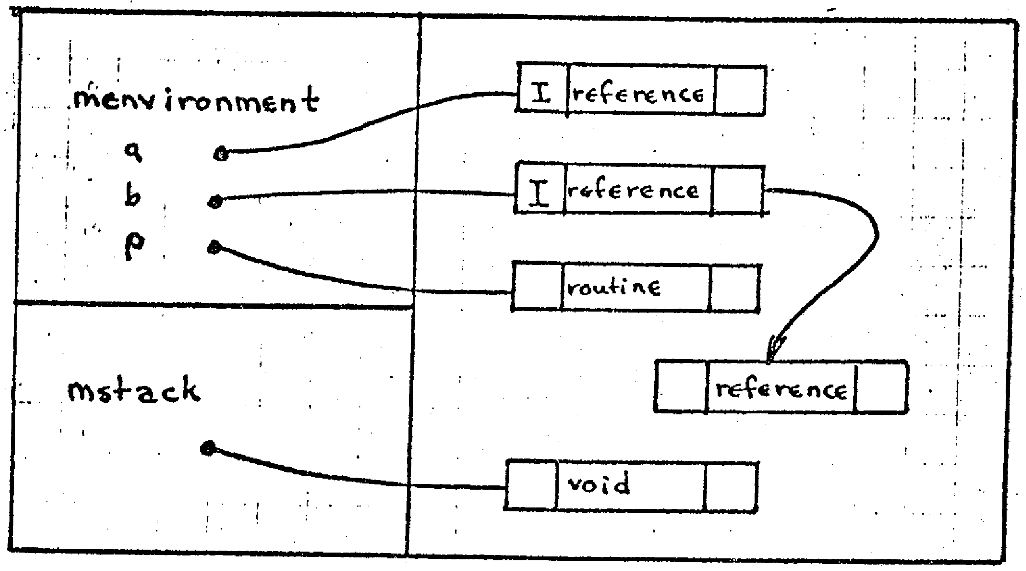
Since procedures are analysed completely independently, the identifiers global to p do not appear in the menvironment. However, one mvariable does appear on the mstack, representing the corresponding variable on the stack of the top level. It is nonindividual here, since its origin lies outside the procedure, and therefore nothing is known about it. Its history, indicated at the right of the box, shows that the it was obtained by dereferencing the mvariable possessed by b.

(2)



The mstack is empty because point (2) is between two statements: there are no temporaries when execution is here.

(3)



These two mstates at points (2) and (3) illustrate the loss of information caused by a procedure call.

Within p, the value obtained by executing '<u>val</u> b' is assigned to 'a'. Because procedures are analysed independently, when the analyser is analysing the procedure 'p', it does not know about 'a', which is declared outside p. Thus '<u>val</u> b' is assigned to a variable about which the analyser knows nothing. This causes the effect '<u>escape</u> <u>val</u> b' to be in the effects directory of p, because the analyser can no longer tell which objects will refer to the value '<u>val</u> b' after a call to p. (e.g. at (3)) The track '<u>val</u> b' allows the analyser to find the mvalue affected at point (2), and the '<u>escape</u>' indicates that the mvalue is no longer individual. The mstate at point (3) reflects this change. Another effect, '<u>set</u> a', which indicates that 'a' was assigned to, was also logged, but it was of little importance in this example. However, this effect would be of interest if the mvariable mpossessed by 'a' had mreferred to some other mvariable. The mreference from 'a' to this other mvalue would have had to be deleted.

2.7 Outline of the complete analysis algorithm, consistency conditions, and proof.

The analysis algorithm constructs, by iteration, a set of mstate sets, one mstate set for the beginning and one mstate set for the end of each expression. The mstate set attached before an expression will be called its 'prefixed' mstate, and the one attached after an expression will be called its 'postfixed' mstate. The prefixed mstate must correctly represent the state of the machine whenever execution of the expression is commenced, and the postfixed mstate must do so whenever execution of the expession is terminated. The algorithm also constructs, by iteration, an effects directory for each routine denotation (i.e., the definition of a routine). The iteration terminates when the mstate sets and effects directories satisfy a certain set of conditions, the 'consistency conditions'. It is these consistency conditions that will guarantee the correctness of the mstate sets obtained by the algorithm.

The consistency conditions are stated in sections 2.9.x.2, for x from 1 to 11. The methods used to ensure termination of the analysis are set forth in Chapter 3.

The consistency conditions are relations that must be satisfied on the prefixed and postfixed mstate sets. Most consistency conditions are expressed as algorithms for computing the postfixed mstate set from the prefixed mstate set.

One thing must be kept in mind throughout the ensuing discussion. It is always possible to discard information. Doing this is simply increasing one's ignorance. It may indeed be necessary to discard information to ensure termination of the analysis. The analyser must not be swamped by infinite amounts of information. This will occasionally lead to the blatant dropping of mvalues from the mstate. However, when representing an infinite class of machine states of unknown complexity, it is necessary to drop information in order to have a finite and practical representation.

The proof of correctness of the consistency conditions takes the following form. Suppose that (somehow) mstate sets have been attached that satisfy the consistency conditions. We must show that these mstate sets correctly describe run-time states. Consider execution of the program. This involves a sequence of execution steps, each of which brings the abstract machine from one state to another. It is proved by induction on the number of execution steps so far executed that each mstate set is a correct (partial) description the state of the machine each time execution passes the point in the program where the mstate set is attached. The proof consists of a number of theorems. In the proofs of these theorems, the preceding theorems are used for execution steps up to and including the 'current' one, and any theorem, even a later one, may be used for execution steps preceding the 'current' one. The circularity here is only apparent. It is precisely the apparent circularity of any proof by induction.

2.8 Preliminaries.

This section contains a number of explanations of concepts that are needed in later sections.

The relations 'passing through', 'quickly reaches', and 'ultimately reaches' are used in applying effects directories to the prefixed mstate set of a procedure call in order to construct the postfixed mstate set. An effects directory describes an effect by specifying a path through run-time data structure and an action to be performed on the mvalue at the end of the path (The action 'escape' will of course have to be performed on the mvalue representing that value). A path is specified by giving a starting point and a sequence of operators to be applied, one after the other, to the starting point. These relations are used in following these paths through the mstate(s) at the point of call.

In Example 2.6.2.2, to apply the effect '<u>escape</u> <u>val</u> b' to the mstate at point (2), the mvalue indicated by '<u>val</u> b' had first to be found. This was done by starting at the mvalue mpossessed by 'b', and applying the operator 'val' to it. The action '<u>escape</u>' was then applied to the mvalue at the end of the path by making it nonindividual.

2.8.1 Passing through.

The relation 'passing through' is used to apply a sequence of the operations '<u>val</u>' and '<u>delink</u>' to an mstate to find where the sequence leads. The operations are applied starting at some mvalue, one after the other, to follow a path through the mstate. At any moment during this application, some of these operations may be 'left over', i.e., not yet applied.

A sequence of operators T, applied at an mvalue A, passes through the mvalue B with a sequence of operators S left over, iff

1. A = B and S = T, or
2. A is a reference mvalue, the first (see end of 2.5) operator of T is <u>val</u>, A mrefers to C, and the sequence of operators T' obtained from T by deleting its first operator (T = T' <u>val</u>), applied at C, passes through B with S left over, or
3. A is a reference mvalue, the first operator of T is <u>delink</u>, A mlinks to C, and the sequence of operators T' obtained from T by deleting its first operator (T = T' <u>delink</u>), applied at C, passes through B with S left over.

A track T (which has a start and a sequence of operators), starting with the identifier A passes through the mvalue B with S left over iff its sequence of operators, applied at the mvalue mpossessed by A, passes through B with S left over.

A history passes through an mvalue iff at least one of its tracks does.

2.8.2 Quickly reaches.

'Passing through' does not require that one go to the very end of the path. 'Quickly reaches' follows the path 'passed through' as far as possible within one mstate. Note that it may not be possible to follow the entire sequence of operators in one mstate, since the operators may lead to values not all represented at a single level.

A track T quickly reaches the mvalue A with S left over iff it passes through A with S left over, and either S is empty, or the first operator of S is '*val*' and A does not mrefer to any mvalue, or the first operator of S is '*delink*' and A does not mlink to any mvalue.

A history quickly reaches an mvalue iff at least one of its tracks does.

2.8.3 Ultimately reaches.

'Ultimately reaches' is meaningful only at execution time. It follows paths where 'quickly reaches' cannot -- from one mstate at one level of the stack to another mstate at another level of the stack. The stack of levels exists only at run time. The significance of 'ultimately reaches' lies in proposition 2.11.6. Since tracks are used in describing effects, it is necessary to know that they suffice for accessing mvalues representing mvalues at other levels. Proposition 2.11.6 establishes this.

A track T ultimately reaches (or, more briefly, 'reaches') the mvalue B starting at some level L iff at that level,

1. T quickly reaches B with nothing left over, or
2. T quickly reaches C with S left over (S may be empty or nonempty), the history of C quickly reaches D on the level below L, and the history H' reaches B, where H' is the history S D = { S R | R is a track in the history of D }, or
3. T quickly reaches C with Q left over, one track O I of the history of C starts with a formal parameter identifier I, and there exists some track P J in the history of the corresponding actual parameter D such that Q O P J reaches B, or
4. the start of T is not in the menvironment of L, and T

ultimately reaches B starting at the level below L.

A history (ultimately) reaches an mvalue iff at least one of its tracks does.

An mvalue reaches another mvalue iff its history does, starting at the level of the first mvalue.

2.8.4 Corollary.

For mvalues A, B, and C, if A reaches B and B reaches C, then A reaches C.

2.8.5 Escaping.

In the rest of this thesis, it will often be stated that some mvalue 'escapes'. An mvalue escapes when it is no longer clear to the analyser which things might refer to or possess the value the mvalue represents.

When an mvalue escapes, any other mvalue it mrefers to also escapes. If the original mvalue is a reference mvalue, it is made nonindividual and is made to mrefer to the 'unknown' mvalue. Each track in the original mvalue's history that is nonlocal to the smallest procedure P containing the point of the program being analysed must be in the effects directory of P with the operator 'escape'.

2.8.6 Garbage collection.

Occasionally, there may arise an mvalue in an mstate which is useless: it cannot be reached from the menvironment or from the mstack. One is tempted simply to discard such mvalues. Indeed, this can be done. If the mvalue is individual, this deletion need not even make the mvalue escape, because the value it represents must be similarly inaccessible, and so can have no further effect on execution.

Note also that it might well be efficient for a compiler to compile explicit storage freeing for such values, and not leave it to a run-time garbage collector.

2.8.7 Pop1, trues, falses, jumps, and unjumps.

The precise description of the analysis algorithm requires several functions: 'pop1', 'trues', 'falses', 'jumps', and 'unjumps'. Their meaning will become more clear where they are used in section 2.9.

Pop1 is a function from mstates to mstates. If M is a state, pop1(M) is M with the top entry removed from its mstack.

'trues', 'falses', 'jumps' and 'unjumps' are functions used by the analyser from the set of mstate sets into the set of mstate sets. Each has an mstate set as argument, and constructs a new mstate set consisting of those mstates from the argument mstate set that satisfy a particular condition.

'Trues' selects those mstates on top of whose mstacks there is an mvalue which might represent 'true', i.e., which is of type 'boolean' and does not have the attribute 'false'.

'Falses' selects those mstates on top of whose mstacks there is an mvalue which might represent 'false'; i.e., which is of type 'boolean' and does not have the attribute 'true'.

'Trues' and 'falses' are used, for example, to select those msttates which should appear at the start of the then part and of the else part of a conditional from the postfixed mstate set of the condition.

'Jumps' selects those mstates with a jump on top of their mstacks. We say that such mstates are 'jumping' mstates (see section 2.5).

'Unjumps' selects those mstates without a jump on top of their mstacks. We say that such mstates are 'unjumping' mstates.

2.8.8 Merging.

Occasionally, paths of control flow together in a program. This happens, for example,at the beginning of the body of a loop, and at the end of a conditional. When we need an mstate to place at this join, it will be constructed by 'merging' the mstates on the various confluent paths. This is the operation introduced intuitively in section 1.7. Precise specifications for merging are stated in section 2.m. Further demands are made on merging in Chapter 3. An algorithm for merging that satisfies all the specifications and demands will be presented in Chapter 3.

2.9 Execution and analysis.

In this section appear the recursive rules for the execution of a program. These rules define the actions of the abstract machine. The consistency conditions, which are the rules for analysis, are interleaved with the rules for execution. In most cases, the correctness of the consistency conditions can be readily seen by comparing them with the rules for execution. Just as execution of an expression changes the abstract machine from one state to another, the analysis of an expression constructs a postfixed mstate from a prefixed mstate. In most cases, the algorithm for analysis is obvious from the consistency conditions. Sometimes, a consistency condition is even stated in the form of an algorithm which computes a postfixed mstate from a prefixed one.

For x from 1 to 12, execution is described in sections 2.9.x.1, and analysis in sections 2.9.x.2. There are occasional extra sections providing further enlightenment.

2.9.1 Identifiers.

2.9.1.1 Execution.

Push the value paired with the identifier in the environment of the top level of the stack onto the value stack at the top level of the stack.

If the value pushed onto the stack was the 'undefined' value, then the program is incorrect.

2.9.1.2 Analysis

To construct the postfixed mstate set of an identifier from the prefixed mstate set, perform the following steps on each mstate:

Let I be the identifier.

1. If I is paired with an mvalue in the menvironment, push this mvalue onto the mvalue stack, and we are done.
2. Otherwise, if I is declared in a declaration whose actual parameter is a routine denotation, create a new mvalue of type 'routine' which consists of the effects directory of the routine denotation, and go to step 4.
3. Otherwise, create a new nonindividual mvalue of type <u>unknown</u>. Its history is the identifier I; i.e., it consists of a single track whose start is I and which has no operations.

4. Push this new mvalue onto the mvalue stack.

2.9.2 Generators.

2.9.2.1 Execution.

Push a new individual value of type ref onto the top of the value stack at the top of the level stack.

2.9.2 Analysis.

To compute the postfixed mstate set from the prefixed mstate set, perform the following operation on each mstate:

Push a new individual mvalue of type ref and empty history onto the mvalue stack.

2.9.3 <u>Conditionals</u> (<u>if</u> condition <u>then</u> then part <u>else</u> else part <u>fi</u>)

2.9.3.1 <u>Execution</u>.

1. Execute the condition of the conditional.
2. Consider the value at the top of the stack.
3. If it is a jump, execution of the conditional is complete.
4. Otherwise, pop it from the stack.
5. If it was <u>true</u>, execute the then part.
6. If it was <u>false</u>, execute the else part.
7. If it was neither a jump, <u>true</u>, nor <u>false</u>, the program is erroneous.

2.9.3.2 <u>Analysis</u>.

```
prefixed mstate set ( condition ) = prefixed mstate set (
     conditional );
prefixed mstate set ( then part ) = pop1 ( trues ( postfixed
     mstate set ( condition ) ) );
prefixed mstate set ( else clause ) = pop1 ( falses (
     postfixed mstate set ( condition ) ) );
postfixed mstate set ( conditional ) = merge (
     postfixed mstate set ( then part ),
     postfixed mstate set ( else part ),
     jumps ( postfixed mstate set ( condition ) ) ).
```

2.9.4 Iterations: while condition do body.

2.9.4.1 Execution.

1. Execute the condition.
2. If the value at the top of the stack is false or a jump, execution of the iteration is complete.
3. Otherwise, pop this top value from the stack.
4. Execute the body.
5. If the value on the stack is a jump, execution is complete.
6. Pop the top value from the stack.
7. Go back to step 1 and continue.

2.9.4.2 Analysis.

```
prefixed mstate set ( condition ) =
     merge ( pop1 ( unjumps ( postfixed mstate set ( body )
          ) ),
          prefixed mstate set ( iteration ) );
prefixed mstate set ( body ) = pop1 ( trues ( postfixed
     mstate set ( condition ) ) );
postfixed mstate set ( iteration ) =
     merge ( jumps ( postfixed mstate set ( body ) ),
          jumps ( postfixed mstate set ( condition ) ),
          falses ( postfixed mstate set ( condition ) ) ).
```

2.8.5 Go to.

2.8.5.1 Execution.

Place a new value on the stack. This value is a jump. Its label is the label from the go to statement. If the label definition is outside any procedure now being executed, the program is erroneous.

2.9.5.2 Analysis.

The postfixed mstate set is obtained from the prefixed mstate set by performing the following operations on each mstate:

Push a new mvalue onto the stack. This mvalue is a jump. Its label is the label from the go to statement.

2.9.6 Serials (blocks).

The complexity of this section is entirely because it is necessary to mimic the behaviour of the program's go to structure. This section would be much simpler if no go to statements were allowed.

2.9.6.0 Definitions.

We define several relations on the phrases of the serial.

A phrase is a 'terminal' phrase iff it is the last phrase of the serial or is followed by a completer.

If and only if a phrase is not a terminal phrase, it has a 'natural successor', the next phrase in the serial. For each unjumping mstate in the postfixed mstate set of the given phrase, we say that the next phrase 'succeeds' the given phrase 'by' that mstate.

A phrase has a second phrase as 'forced successor' iff the postfixed mstate set of the first phrase contains an mstate that is a jump to the second phrase. The second phrase is said to succeed the first phrase by that mstate. This second phrase will be the target of some go to statement in the first phrase, and the mstate will be the mstate associated with the path of control through the go to statement. Such mstates must be merged into the prefixed mstate set of the next phrase; we shall define this presently.

A phrase has a second phrase as 'possible successor' iff it has the second phrase as a natural or forced successor.

2.9.6.1 Execution.

1. For each declaration of the serial, add to the environment at the top level an identifier possessing an undefined value.
2. Consider the first phrase of the serial.
3. Execute the considered phrase.
4. If the top value on the stack is a jump, and its label is defined in this serial, then consider the phrase which follows the label definition, delete the top value from the stack, and go to step 3.
5. Otherwise, if the top value on the stack is a jump, and its label is not defined in this serial, delete the environment pairs that were added in step 1, and execution of the serial is complete.
6. Otherwise, if the top value on the stack is not a jump, and the phrase is a terminal phrase, remove from the environment the pairs that were added in step 1, and execution of the serial is complete.
7. Otherwise, if the top value on the stack is not a jump, and the phrase is not a terminal phrase, then consider its natural successor instead, pop the top value from

the stack, and go to step 3.

2.9.6.2 Analysis.

We obtain the postfixed mstate set of a serial in a number of stages.

First, to obtain the 'modified prefixed mstate set' of the serial, modify the prefixed mstate set by making the following change to each mstate:

> To the environment add pairs that pair each identifier declared in the serial with the undefined mvalue.

Next, we obtain the prefixed and postfixed mstate sets of the phrases in the serial.

```
If p is the first phrase,
    prefixed mstate set(p) = merge (
        modified prefixed mstate set (serial),
        pop1(m), for every mstate m in the postfixed mstate set
            of every phrase q of the serial, such that p
            succeeds q by m )
If p is an other phrase,
    prefixed mstate set (p) = merge (
        pop1(m), for every mstate m in the postfixed mstate set
            of every phrase q of the serial, such that p
            succeeds q by m )
```

Next, the 'draft postfixed mstate set' of the serial is obtained by merging the postfixed mstate sets of all the terminal phrases, together with every mstate of any mstate set of any phrase of the serial whose value is a jump outside the serial.

Finally, the postfixed mstate set of the serial is obtained by deleting, for each mstate of the draft postfixed mstate set, the environment entries corresponding to the declarations in the serial.

2.9.7 Routine denotations.

2.9.7.1 Execution.

Place on the stack a new value, a routine value, which consists of the routine denotation, and the top level on the level stack.

2.9.7.2 Analysis.

Place on each mstack a new mvalue of type 'routine' which has the effects directory of the procedure denotation.

2.9.8 Calls.

2.9.8.1 Execution.

1. Execute the actual parameters of the call, in order.
2. If the execution of any actual parameter leaves a routine value on the stack, the program is incorrect.
3. Execute the function of the call. The value now on top of the stack must be a routine value; otherwise, there is a program error.

We now begin the construction of a new level for the level stack.

4. The environment in the routine value is considered. To construct the new environment, we add to a copy of the considered environment new pairs identifying the formal parameters with the values of the actual parameters.
5. A new level is placed on the level stack, containing an empty value stack, and the new environment.

We have now constructed the new level. Next, the body of the routine denotation is executed.

6. Execute the body of the routine denotation.
7. Pop the value on the stack.
8. Pop the top level from the level stack.
9. Pop the value produced by execution of the function from the stack.
10. Pop the values produced by execution of the actual

parameters from the stack.

11. Place a void value onto the stack.

12. Execution of the call will now be complete.

2.9.8.2 Analysis.

If there are actual parameters,

> the prefixed mstate set of the first actual parameter is the prefixed mstate set of the call,
>
> the prefixed mstate set of each actual parameter (other than the first) is the postfixed mstate set of the previous actual parameter, and
>
> the prefixed mstate set of the function is the postfixed mstate set of the last actual parameter.

If there are no actual parameters,

> the prefixed mstate set of the function is the prefix mstate set of the call.

Analysing these actual parameters and the function, one after the other, simply leaves mvalues representing the values of the parameters and of the function on the mstack.

The postfixed mstate set of the call is the postfixed mstate set of the function after the following modifications have been made to each mstate:

Let F be the function part of the call. Let E be the effects directory of the mvalue left by F. Let P be the smallest routine denotation containing the call.

A new environment is constructed, and used temporarily. It consists of the current environment, with extra pairs associating new identifiers which do not occur elsewhere in the program with the mvalues placed on the mstack by the execution of the actual parameters. A new effects directory E' is constructed from the old effects directory E by replacing the formal parameters by these new identifiers. Thus the effects directory is translated into the notation of the new environment. The change of names is to prevent trouble with recursion, and is similar to the change of names used in the ALGOL 60 Report for defining procedure calling by the copy rule.

If a track of an effect in E' with action Q (either '<u>set</u>' or '<u>escape</u>') passes through an mvalue A with S left over (see section 2.8), then:

1. For each track R in the history of A a new effect Q S R is placed in the effects directory of P.

2. If Q is 'set' and S is empty, then A is marked with a 'set mark'.

3. If Q is 'escape' and S is empty, then A is marked with an 'escape mark'.

Each mvalue with an escape mark escapes.

Each mvalue mreferred to by an mvalue with a set mark escapes.

Each mvalue with a set mark is made to mrefer to the undefined mvalue.

2.9.8.3 <u>Correlation</u>.

When we correlate execution and analysis of procedure calls, we imagine that this postfixed mstate set becomes the current mstate set (see section 3.5.1) at the time that the new stack level is constructed. Having thrown away information according to the effects at this time, we are sure that we need not do it while the body of the routine denotation is being executed. Several theorems are necessary to prove that the effects directory provides correct information about the effects of assignments. These theorems are stated and proved later, and are used in the proof of correctness of the handling of assignments. Thus, all we need be concerned with here is that the representation of setting up the call, i.e., executing parameters, etc., is correct. That our representative continues to be correct during execution of the procedure body follows from the correct handling of each piece of code within the procedure body.

2.9.8.4 Commentary.

Note that our procedures do not return function values. This is to avoid the complexity of having yet another way for procedures to affect their calling environment. Values can be returned by assignment to nonlocal variables.

2.9.9 Assignment and link setting.

Only assignment will be spelled out here. Link setting is entirely analogous.

2.9.9.1 Execution.

1. Execute the source of the assignment.
2. If the value on the stack is a routine value, the program is incorrect.
3. Execute the destination of the assignment.
4. The value on the stack must be a nonnull reference value; otherwise, the program is incorrect.
5. The value on the top of the stack is made to refer the value second from the top of the stack.
6. The value on the top of the stack is considered.
7. The top two values are popped from the stack.
8. The considered value is placed on the stack.

2.9.9.2 Analysis.

prefixed mstate set (destination) = prefixed mstate set (assignment);

prefixed mstate set (source) = postfixed mstate set (destination);

postfixed mstate set (assignment) = postfixed mstate set (destination) with the following modifications:

Call the top and next to top mvalues on the mstack the 'mdestination' and 'msource'.

If the mdestination is individual, make it mrefer to the msource.

If the msource is individual, but the mdestination is not, the msource becomes nonindividual, and an effect 'escape' for every history of the msource must exist in the effects directory of the smallest routine denotation containing the assignment.

If the mdestination is not individual, there must be an effect indicating it has been an assignment target.

2.9.9.3 Commentary.

The hard part of proving that the analysis of assignment is correct is the proof that effects of the assignment on levels other than the top level are properly handled. The proof appears in section 2.12.

2.9.10 Dereferencing.

1. Execute the argument of the dereferencing.
2. If the value on top of the stack is not a reference value, the program is incorrect.
3. If the value on top of the stack is the null reference, the program is incorrect.
4. If the value on top of the stack is the undefined value, the program is incorrect.
5. Otherwise, replace the value on top of the stack by the value it refers to.

2.9.10.2 Analysis.

The prefixed mstate set of the argument of the dereferencing is the prefixed mstate set of the dereferencing.

To construct the postfixed mstate set of the dereferencing from the postfixed mstate set of the argument, replace the mvalue on top of the mstack of each mstate in the following manner:

If it is individual, replace it by the mvalue it refers to.

If it is nonindividual, replace it by a newly constructed nonindividual mvalue, whose history is the history of the original mvalue with the operation '<u>val</u>' added.

2.9.10.3 Correlation.

The new (replacing) mvalue represents the new value which replaces the previous value on the stack.

```
history ( new value ) = union (
     old history (new value on stack ) ,
     'val' history (old value on stack ) ) .
```

If the mvalue was individual, the mvalue newly on the stack represents the value newly on the stack just as it did before they were placed on the stack.

2.9.11 Boolean denotation.

2.9.11.1 Execution.

If it is '*true*' ('*false*'), place the value '*true*' ('*false*') on the stack.

2.9.11.2 Analysis.

To construct the postfixed mstate set, place the boolean mvalue '*true*' ('*false*') on the mstack.

2.9.12 Declarations.

2.9.12.1 Execution.

To execute a declaration, perform the following steps:

1. Execute its actual parameter.
2. Add a new pair to the environment, pairing the formal parameter of the declaration (an identifier) with the value on top of the stack.
3. Pop the top value from the value stack.

2.9.12.2 Analysis.

The prefixed mstate set of the actual parameter is the prefixed mstate set of the declaration.

To obtain the postfixed mstate set of the declaration from the postfixed mstate set of the actual parameter, perform the following steps on each mstate:

1. Add a new pair to the environment, pairing the formal parameter of the declaration (an identifier) with the mvalue on top of the stack.
2. Pop the top value from the mvalue stack.

2.10 Merging.

Let M1 and M2 be two mstates. We write the conditions for M3 to be a possible result of merging M1 and M2:

There must exist surjective (onto) mappings F1: S1->M3 and F2: S2->M3 from subsets S1 and S2 of the sets of mvalues of M1 and M2.

(These mappings can be considered as identifying which new mvalues in M3 are the 'same' as the old mvalues in M1 and M2. It will be claimed in section 2.11.1 and proved in section 2.12.1 that if an mvalue V in S1 represents some value V', then F1(V) in M3 represents the same value V'.)

These mappings must satisfy:

1) S1 and S2 include the mvalues on the environment and the stack.

2) If Fi(Vi) is individual, then Fi^{-1}(Fi(Vi)) has only one element, Vi, and it is individual (i=1,2).

3) The history of Vi is a subset of the history of Fi(Vi).

4) If Fi(V) mrefers to Fi(W), then V mrefers to W.

5) If V is j-th from the top of the mstack, then Fi(V) is j-th from the top of the mstack.

6) If V is mpossessed by an identifier, then Fi(V) is mpossessed by the same identifier.

7) If Fi(V) is boolean (true, false), then so is V.

8) If Fi(V) is of type 'routine', then so is V, and its effects directory includes that of V.

9) If Fi(V) is of type 'reference', then so is V.

10) If Fi(V) is not individual, but V is, then V has escaped.

11) If V in Mi is not in the domain of Fi, then V has escaped.

2.11 Theorem.

This section contains the statement of the main theorem of the thesis, that the consistency conditions ensure correctness of analysis.

2.11.1 Part 1.

If the specifications of section 2.10 are met, and either M1 or M2 represents the state at the top level, then so does M3.

2.11.2 Part 2.

If a variable V represented by an individual mvalue M at level L is referred to by a variable W, then W is represented by an individual mvalue N at level L or higher, and N mrefers to some mvalue M' representing V.

```
         state of abstract machine:  W ---> V
    representative at higher level:  N ---> M'
         representative at level L:         M
```

2.11.3 Part 3.

A variable represented by an individual mvalue at some level L is

(1) not represented at any lower level,

(2) not represented by any other mvalue N at the same level, and

(3) not represented by an individual mvalue at any higher level.

2.11.4 Part 4.

A variable V represented by an individual mvalue N is not referred to by a variable W

(1) represented by a nonindividual mvalue at the same level, or

(2) represented by any mvalue at any lower level, or

(3) unrepresented by any mvalue.

2.11.5 Part 5.

A variable V represented by an mvariable with a history nonlocal to some level L (i.e., one of the tracks of the history starts with an identifier that is not local to L) is not represented by an individual mvalue at that level (nor, of course, at any higher level).

2.11.6 Part 6.

If a variable V is represented by an individual mvalue I, but also by a nonindividual mvalue D (at a higher level L), then a nonlocal track in the history of D ultimately reaches I. Here 'nonlocal' means 'not local to L'; i.e. the track starts with an identifier which is global to the body of the procedure of level L.

Note: This proposition is the very heart of the correctness of the analysis algorithm. It says, in effect, that histories are adequate for exposing side effects on individual variables.

2.11.7 Part 7.

At each instant at run-time, we are nested in one or more levels. All but one of these levels will be executing a procedure call. Each of these levels will have an mstate set associated with its present point of program execution. The mstate sets correspond to the state of the abstract machine in the following sense:

1) Each mvalue represents at least one value.

2) No individual mvalue represents the same value as any other mvalue at the same level nor at any lower level.

3) No two distinct individual mvalues, at the same or different procedure call level, represent the same value.

4) If one mvalue mrefers to another mvalue, then the value represented by the first mvalue refers to the value represented by the other mvalue.

5) The value represented by any boolean (jump, routine, reference) mvalue is boolean (jump, routine, reference).

6) The value represented by any boolean true mvalue is the value <u>true</u>.

7) The value represented by any boolean false mvalue is the value <u>false</u>.

8) The mvalues on the mstack represent the values at the same position on the portion of the stack belonging to that level.

9) An individual mvalue represents only one value.

Note that these conditions include the requirements of 2.6.1.

2.12 Proof.

This section contains the proof of the results stated in section 2.11. The proof is by induction on the number of steps of execution executed so far. At the start of execution, the theorem is trivially true, since the mstates attached at the start of the program are empty.

For convenience, the various parts of the theorem are restated near their proofs.

2.12.1 Part 1.

If the specifications of section 2.10 are met, and either M1 or M2 represents the state at the top level, then so does M3.

Proof:

Without loss of generality, suppose M1 represents the state.

The only nontrivial part of the proof is to show that individuality is properly treated. Let M be an individual mvalue in M3, and let M represent V and W. $F1^{-1}(M)$ has only one element. Call it M'. M' is individual. Since M represents V and W, $F^{-1}(M)=M'$ represents V and W. Therefore V = W.

Let P represent P', and let Q represent Q' in M3.

If P≠Q and P is individual, consider $F1^{-1}(P)$ and $F1^{-1}(Q)$. Since F1 is a function, $F1^{-1}(P)$ and $F1^{-1}(Q)$ are disjoint. $F1^{-1}(P)=P'$ in M1 is individual. Thus, since P' is represented by P1 and Q' is represented by an element of $F1^{-1}(Q)$ in M1, P'≠Q'.

2.12.2 Part 2.

If a variable V represented by an individual mvalue M at level L is referred to by a variable W, then W is represented by an individual mvalue N at level L or higher, and N mrefers to some mvalue M' representing V.

```
        state of abstract machine:  W ---> V
    representative at higher level:  N ---> M'
        representative at level L:          M
```

Proof:

Consider the time, if any, at which this is first violated. Either (1,2) the condition has become true, or (3,4) the conclusion has become false.

Either:

(1) an assignment (or link setting) was executed causing V to be referred to by W, and W is individual at no level, or at a level lower than L, or

(2) an assignment is executed causing V to be referred to by W, and W is represented by an individual mvalue N, or

(3) W, which refers to V, ceases to be represented by any individual mvalue N, or

(4) the individual mvalue N, although W still refers to V, ceases to mrefer to any mvalue M' representing V.

In case 1, the destination of the assignment (W) is represented at the top level by a nonindividual mvalue. Thus the top-level mvalue representing N, the source, will escape. Since M was individual, this escape will reach M by part 6, and so M cannot be individual.

In case 2, N will be made to mrefer to the mvalue M' representing the source.

In case 3, M' will escape.

In case 4, because mvalues are lost only by merging, a merger has taken place, and so V escapes(section 2.10(11)).

2.12.3 Part 3.

A variable represented by an individual mvalue at some level L is

(1) not represented at any lower level,

(2) not represented by any other mvalue N at the same level, and

(3) not represented by an individual mvalue at any higher level.

Proof:

For the variable V to be represented by an individual mvalue N at level L, it has to have been created by a generator executed at level L.

(1) Since the mstate set at a lower level does not change during execution of the higher level L, V cannot be represented by any mvalue at this lower level: V did not exist when execution at the intermediate level started.

(2) V was not represented by another mvalue at the level L when the generator was executed, since the value of the generator was a new value. Therefore, for another mvalue M at level 1 to start representing V, a new mvalue for V must be constructed. This can happen only by dereferencing (or delinking) a value W which referred to V, when L is the top level. By part 2, N, representing V, is then mreferred to by some individual mvalue M

representng W at the top level. By part 3, no other mvalue represents W at the top level, so W is the mvalue placed on the stack by executing the argument of the dereferencing. W mrefers to V, so no new mvalue is created.

(3) This is direct. Variables represented by individual mvariables at different levels arise from execution of generators at different times. Therefore they are distinct variables.

2.12.4 Part 4.

A variable V represented by an individual mvalue N is not referred to by a variable W

(1) represented by a nonindividual mvalue at the same level, or

(2) represented by any mvalue at any lower level, or

(3) unrepresented by any mvalue.

Proof:

Suppose V is represented by a variable W. Let N represent V at level L. By part 2, W is represented by an individual mvalue at level L or higher. If at level L, then by part 3, W is not represented by any nonindividual mvalue at level L. If higher, then by part 3(1), W is not represented at level L. In either case, by part 3, W is not represented at any lower level.

2.12.5 Part 5.

A variable V represented by an mvariable with a history nonlocal to some level L (i.e., one of the tracks of the history starts with an identifier that is not local to L) is not represented by an individual mvalue at that level (nor, of course, at any higher level).

Proof:

For an mvariable representing V to acquire a history nonlocal to L, it must have been obtained either (1) by analysing a nonlocal identifier, or (2) by dereferencing (or delinking) a variable W represented by an mvariable with a history nonlocal to L.

In case (1), a new mvariable as created to represent V. This mvariable is nonindividual because the identifier is not in the menvironment. By part 3, no mvariable representing V can be individual.

In case (2), W was made nonindividual by part 5, and so V is nonindividual by part 4.

2.12.6 Part 6.

If a variable V is represented by an individual mvalue I, but also by a nonindividual mvalue D (at a higher level L), then a nonlocal track in the history of D ultimately reaches I. Here 'nonlocal' means 'not local to L'; i.e. the track starts with an identifier which is global to the body of the procedure of level L.

Note: This proposition is the very heart of the correctness of the analysis algorithm. It says, in effect, that histories are adequate for exposing side effects on individual variables.

Proof:

Consider the earliest time that the conditions are satisfied but the conclusion is not. Then (1) the conclusion becomes false or (2,3) the condition becomes true. Either

1) the nonlocal track in D's history just stopped reaching I, or

2) V just became represented by an individual mvalue I.

3) V just became represented by a nonindividual mvalue D, whose history has no track that reaches I.

1) in this case, only the mstate on the top level has

changed. This cannot affect the reach of nonlocal tracks. Therefore, this case is impossible.

2) In this case, the generator that generated V was just executed. V is represented by just one mvalue, so D did not exist.

3) D must have been just created by dereferencing (or, of course, delinking) some value W which refers to V. By 2.9.1, W is represented at some level by an individual mvalue J. Let J mrefer to C. Let K be the mvalue representing W at the top level. K mrefers to D.

$$\begin{tikzcd}
W \text{ is represented by} \arrow[d] & & J \arrow[d] & K \arrow[d] \\
V \text{ is represented by} & I & C & D
\end{tikzcd}$$

Since a nonlocal track in the history of K reaches J (2.10), a nonlocal track in the history of D reaches C. Since no nonlocal track in the history of D reaches I, no nonlocal track in the history of C reaches I (corollary 2.8.4). But C existed before the dereferencing (it was guaranteed to exist by part 2) and so we have an earlier counterexample.

2.12.7 Part 7.

At each instant at run-time, we are nested in one or more levels. All but one of these levels will be executing a procedure call. Each one of these levels will have an mstate set associated with its present point of program execution. The mstate sets correspond to the state of the abstract machine in the following sense:

1) Each mvalue represents at least one value.

2) No individual mvalue represents the same value as any other mvalue at the same level nor at any lower level.

3) No two distinct individual mvalues, at the same or different procedure call level, represent the same value.

4) If one mvalue mrefers to another mvalue, then the value represented by the first mvalue refers to the value represented by the other mvalue.

5) The value represented by any boolean (jump, routine, reference) mvalue is boolean (jump, routine, reference).

6) The value represented by any boolean true mvalue is the value true.

7) The value represented by any boolean false mvalue is the value false.

8) The mvalues on the mstack represent the values at the same position on the portion of the stack belonging to that level.

9) An individual mvalue represents only one value.

Note that these conditions include the requirements of 2.6.1.

Proof:

The proof is divided into cases, one for each syntactic type.

1) Assignment.

Here we concern ourselves with actually performing the assignment after the source and destination have been executed.

1a) If the destination is an individual mvalue, then the destination is not represented by any other mvalue Therefore, only the mvalue referred to by the destination need be adjusted. Since the destination is represented at the top level by an individual mvalue, it is not represented at any lower level. Therefore, no change is required on any lower level.

1b) If the destination D is represented by an individual mvalue I at a level other than the top level, we must consider the effects directory. By part 6, the history of the mvalue representing D must reach I, and so I will not refer to any mvalue.

1c) If the destination D is not represented by an individual mvalue at any level, the mstate set makes no claims about the value it refers to, so no change is necessary.

That the mstack is correctly adjusted is clear.

2) <u>Dereferencing</u>.

Trivial.

3) <u>Conditional</u>

This follows directly from the properties of merging.

4) <u>While loop</u>.

This follows directly from the properties of merging.

5) <u>Jump</u>

The effect of a jump is to convert a normal state into a jumping state. The modification of the mstate clearly reflects this.

<u>Note:</u> Various actions commonly considered part of the jump, such

as unstacking values and terminating blocks, are here distributed throughout the definition of the language. Therefore, the analyser performs them elsewhere, and they need not be accounted for here.

6) Serial.

We must check that the theorem continues to hold

(1) between initiating the serial and entering the first unit or declaration, and

(2) between terminating one unit or declaration and commencing another (possibly via a jump),

(3) between terminating the last unit or declaration and terminating the serial.

(1) clearly holds. It is the transition from the prefixed mstate set to the modified prefixed mstate set.

(2) is less trivial. When we terminate one unit or declaration, the state, either a normal state or a jump, is represented by some particular mstate, either a normal mstate or a jump. If the mstate is not a jump, execution of the next unit is initiated. If it is a jump, execution of the unit after the jump target is initiated. if the next unit is not labelled, the prefixed mstate set is the postfixed mstate set of the previous unit, except that the top value on each mstack

is deleted (just like the stack). If the next unit is labelled, we delete the top value from the mstack and stack. The prefixed mstate set for the new unit is the result of merging several mstates, one of which is the current one, q.e.d.

(3) The operations on the mstate set clearly are precisely the images of those on the stack.

7) Identifier.

Trivial.

8) Generator.

Trivial.

9) Denotation.

Trivial.

10) Procedure call.

After execution of the function and parameters, we have an mstate set. The pre-call processing involves only deletion of information from the mstate set, and therefore cannot affect correctness. Once the call is being performed, it is up to the other conditions in this theorem to ensure that this mstate set (which is unchanged) remains valid (see the lengthy discussion of assignment, for example). The post-call stack trimming is straightforward.

Chapter 3.

Outline of analysis algorithm.

3.1 Introduction.

Chapter 2 presented mstate sets, which can be attached to a program, and can be used to state certain facts about the program. It also presented the consistency conditions, which are sufficient conditions for those mstate sets to correctly represent the states of the abstract machine. It is the aim of this chapter to outline techniques that can be used to represent mstate sets on real machines, and to compute, in a reasonable amount of time, mstate sets that satisfy the consistency conditions presented in Chapter 2.

The basic computational technique used is iteration. Initial estimates are chosen for the mstate sets and effects directories attached to the program, and are checked against the consistency conditions. For each mismatch between the estimate and the consistency conditions, the 'postfixed' mstate set is changed so that it will meet the consistency conditions it failed to meet. These changes are continued until all consistency conditions are satisfied.

Let us look at this process informally. First consider the initial estimates.

The mstate sets we attach at all points, except the beginnings of procedure bodies and at the beginning of the entire program, are empty -- they contain no mstates. The mstate set at the start of the program has a single mstate, with an empty mstack, an empty menvironment, and no mvalues. The mstate set attached at the beginning of a procedure body is just like the one at the start of the program, except that it has an menvironment identifying the formal parameter identifiers with unknown mvalues. Each of these unknown mvalues has a history consisting of its formal parameter identifier.

The effects directory of every procedure is initialized to empty.

If M1 and M2 are mstate sets attached to the same point in the program at earlier and later times during the analysis, then M1 logically implies M2 (Remember from 2.6.1 that an mstate set could be considered a statement about the state). To force termination, then, it will be sufficient to prevent an infinite chain

M1 implies M2 implies M3 implies ...

of distinct mstate sets.

The analysis algorithm outlined here is divided into two parts. The first is an algorithm to perform the analysis of an individual routine denotation (or of the main program, which will be treated as the body of a parameterless routine), omitting any other routine denotations contained therein. The second is an algorithm to bring these diverse analyses together. The second algorithm uses the first as a subroutine.

3.2 Conditions that force convergence.

Each mstate set is a finite collection of mstates, which we imagine connected with the logical connective 'or'. An infinite ascending chain of mstate sets is possible, each one of which has just one more mstate than its predecessor. This might arise, for example, when analysing a loop for which the analyser cannot detect termination. To prevent this, an arbitrary limit is imposed on the number of mstates in a mstate set:

(1) No more than one nonjumping mstate, and

(2) No more than one mstate for each jump target.

This particular restriction was chosen because it was easy to implement. Furthermore, it permits lemma 3.2.1 to be proved. However, permitting only one mstate of each type effectively eliminates knowledge of correlations between values, such as "This pointer is nonnull iff this other value is true."

If this restriction on the number of mstates is enforced, then an infinite sequence of mstate sets can arise only from an infinite sequence of mstates (this is proved in lemma 3.2.2). We could then obtain from this infinite sequence of mstates an infinite sequence of histories (this is proved in lemma 3.2.3). Therefore, if we prevent an infinite ascending sequence of histories by an additional restriction, we will have achieved our goal. Infinite sequences of histories also might result from program loops. Since histories may be infinite sets, we need to find some finite representation for them anyway. We shall use our freedom to add extra tracks, and choose finite representations so that infinite ascending chains cannot exist. Not all histories will be accurately representable, but any history will be contained in some representable history. The process of finitely representing histories will discard information by enlarging them. This is unfortunate, but may be unavoidable. Some hope is offered in section 5.3.2.

3.2.1 Representation of histories.

A track in a history may be considered as a character string -- one character for the start of the track, and one character for each operator. A history is thus a set of strings, and can be represented by any of the common notations for languages, such as grammars or regular expressions. The representation chosen here is to represent a history by a regular expression of the form

$$O1\ S1 + O2\ S2 + \dots + On\ Sn$$

where each Si is a start, and each Oi is a regular expression describing the possible operators applied to Si. Each Oi is of the form

$$(R1 + R2 + \dots + Rk)^*\ P1\ P2\ P3 \dots Pm$$

or

$$P1\ P2\ P3 \dots Pm.$$

where each Pj and Rj is an operator. Furthermore, an arbitrary bound is placed on the number of Pj's allowed. No two Oi are allowed to have the same sequence of Pj's as well as the same Si.

If only one operator were possible, the expressions would reduce to triviality -- we could use an integer expressing the minimum number of operators to be applied, together with a bit indicating whether more operators can be added.

It is quite easy to construct algorithms to perform the necessary operations on histories represented in this manner. Bear in mind that we can always place an operator in the iterated part (inside the parentheses before the "*") if the number of Pj's becomes too large.

3.2.2 Lemma.

(This lemma is used in section 3.2.) If M[1], M[2], ... is an infinite sequence of distinct mstate sets satisfying restrictions (1) and (2) of section 3.2 and if each M[i] implies M[i+1], then there is an infinite sequence of distinct mstates m[a], m[a+1], ... among these mstate sets such that each m[i] implies m[i+1].

Proof:

If M[i] contains an mstate with a particular jump target, then so must M[i+1]. Otherwise, M[i+1] could not represent some machine state that M[i] can. Similarly, if M[i] contains an mstate with a nonjumping mstate, then so must M[i+1]. Thus we have a finite (bounded by one plus the number of labels in the program) number of sequences

m0[ao], m0[a0+1], m0[a0+2], ... ;

m1[a1], m1[a1+1], m2[a1+2], ... ;

...

mn[an], mn[an+1], mn[an+2], ...

such that:

(1) the entries in one of these sequences are either all jumps to the same jump target, or all nonjumping mstates.

(2) mj[i] is in the mstate set M[i].

(3) Every mstate in M[i] is among {m0[i], m1[i], ..., mn[i]}.

We further note that, since there is only one mstate in each mstate set with a particular (or no) jump target, and M[i] implies M[i+1], mj[i] must imply mj[i+1] whenever i>aj.

If there were no infinite ascending chain of mstates, then each of these sequences would terminate, i.e., there must exist T0, T1, ..., Tn such that mj[i] = mj[i+1] for i>Tj. Let T be the largest of the Tj and aj. Then for i>T, M[i]=M[i+1].

<u>Note</u>: If we had had more than one mstate of each type in a mstate set, we could not have said that mj[i] implies mj[i+1]. Whether this result can still be forced under a less restrictive constraint on the number of mstates in a mstate set I do not know.

3.2.3 Lemma.

(This lemma is used in section 3.2.)

If there is an infinite sequence of distinct mstates

m[1] implies m[2] implies m[3] ...,

then there is an infinite sequence of distinct histories

h[1], h[2], h[3], ...

such that h[i] is a subset of h[i+1].

Proof:

The proof of this lemma is similar to that of 3.2.1. The main difference is that we cannot use jump targets to correlate an mvalue in one mstate with an mvalue in another mstate. However, if one mstate implies a second mstate, the second is the same as the first except that some mvalues, some mreferring and mlinking information, and some other information may be missing. Since mstates are finite, at some point in the sequence no more mstates are dropped, and from this point on, sequences of mvalues analogous to the sequences of mstates in 3.2.1 can be constructed.

3.3 Analysis of an individual routine denotation.

The analysis of a procedure denotation involves the computation of an effects directory and the attachment of mstate sets to each expression within the procedure denotation, except for procedure denotations contained properly within it. This section will assume that approximations (possibly empty) are available for the effects directories of all procedure denotations in the program. These effects directories will be used to compute a new effects directory for the procedure denotation.

First, initial approximate mstate sets are attached to each expression in the procedure body. The mstate sets computed on a previous analysis of the procedure denotation can be used. Empty mstate sets can also be used, except that the mstate set attached to the beginning of the procedure body must have one mstate. This one mstate must have an empty stack, an menvironment identifying the formal parameters with the unknown mvalue, and no other mvalues.

Then an iteration is performed, using the consistency conditions in the previous chapter as iteration steps. For each point where an mstate set might be attached, except the start of the procedure body, the attached mstate set is re-computed using the consistency conditions, the existing approximate effects directories, and the existing approximate mstate sets attached at other points in the program. If the consistency conditions require new effects in the effects directory of the procedure being analysed, then these effects are inserted. This process is repeated until no change in any approximate mstate set occurs under recomputation anywhere in the procedure.

The process described above is adequate, since under the restrictions of section 3.2 it is forced to terminate. However, it is not yet completely described. The previous chapter described merging by a set of axioms (2.m), not by an algorithm. An algorithm for merging is described in the next section.

3.3.1 Merging.

3.3.1.1 Merging mstate sets.

To merge two mstate sets, first construct their set-theoretic union. Then merge any mstates that violate the restrictions in section 3.2 on the number of mstates. The next section describes merging of mstates.

3.3.1.2 <u>Merging mstates</u>.

Let M1 and M2 be the mstates to be merged. Let M be the disjoint union of the sets of mvalues of M1 and M2. Define the following equivalence relation R on M:

For mvalues V in M1 and W in M2, V R W iff either

1) V and W are mpossessed by the same identifier, or
2) V and W are both i-th from the top of the mstack, for some integer i.

Extend R to the smallest equivalence relation S on M that satisfies

1) V R W implies V S W.
2) If V mrefers (mlinks) to W, and V' mrefers (mlinks) to W', and V S V', then W S W'.

The mvalues for a new mstate M' are the equivalence classes of M under the relation S. The mappings Fi of 2.10 are the mappings induced by the equivalence relation. A new mvalue mrefers to another iff each of its elements mrefers to an element of the other. A new mvalue is in a position on the mstack iff one of its members was in that position on an old mstack. An identifier mpossesses a new mvalue iff it (in the old mstates) mpossesses at least one of its elements.

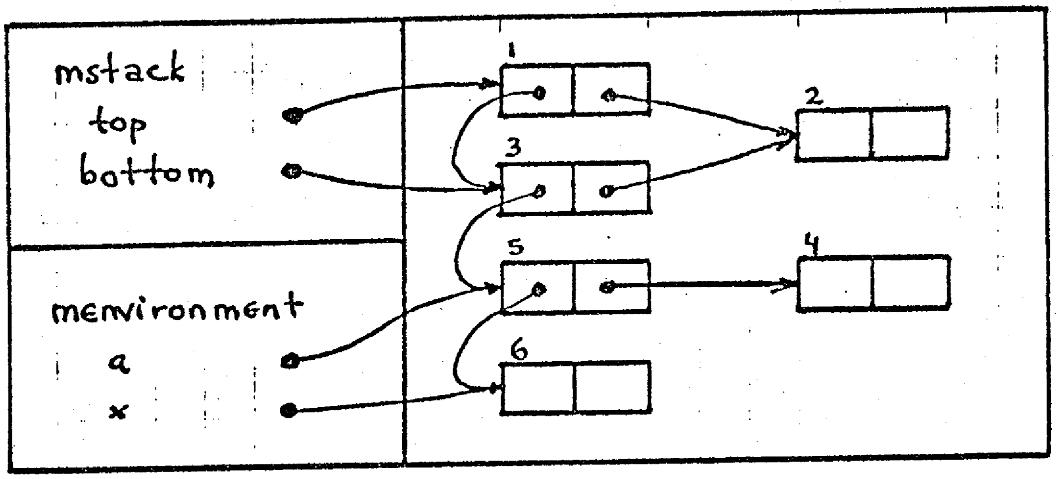
A new mvalue is individual iff it has just one member from each old mstate, and each of its members is individual.

Effects with the action 'escape' are placed in the effects directory of the routine denotation being analysed as necessary to satisfy conditions 10 and 11 in Section 2.10 (on merging).

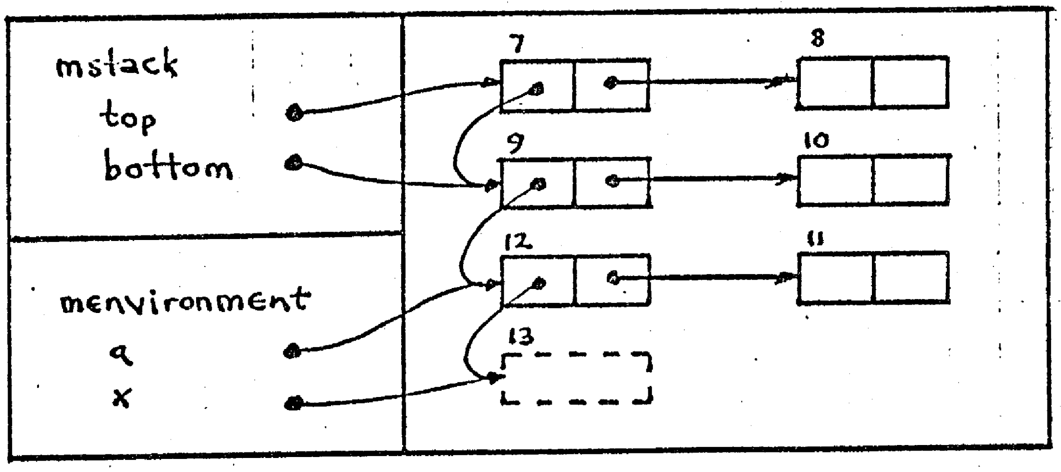
The history of each new mvalue is obtained by merging the old histories of the elements of M.

3.3.1.3 Example.

Suppose we must merge the mstate



with the mstate



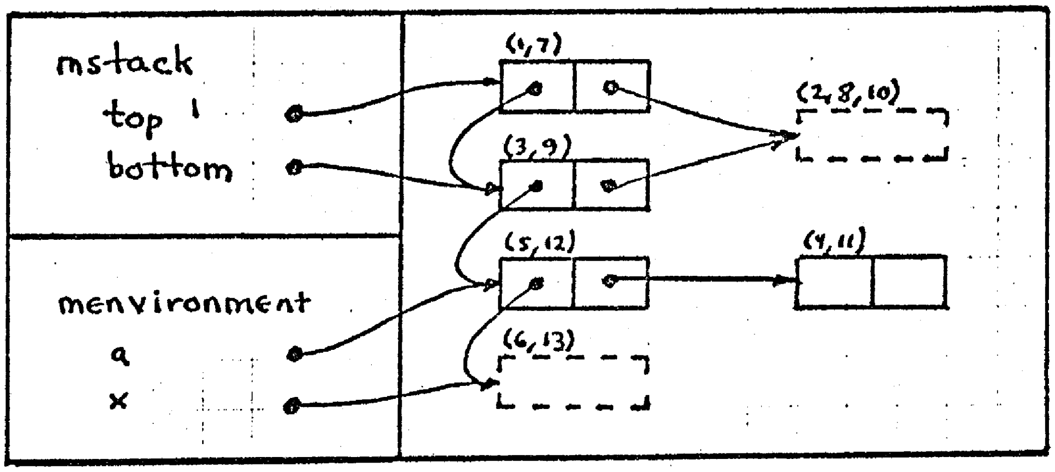
The equivalence relation R has equivalence classes

(1,7), (3,9), (5,12), (6,13), 2, 4, 8, 10, 11

S has equivalence classes

(1,7), (3,9), (5,12), (6,13), (2,8,10), (4,11)

Thus the new mstate is



Escapes will be logged for the histories of mvalues

2, 6, 8, 10.

Note that if the second mstate was an accurate representation of the state of the abstract machine, mvalues 8 and 10 must have represented different mvalues. Therefore, the new mvalue (2,8,10) represents two values, not just one.

3.4 Analysis of the entire program.

Here is a method of integrating the analyses of the various procedure denotations in the program.

Initialize the effects directory of each procedure denotation to the empty set. Then analyse procedure denotations until further analysis of procedure denotations has no further effect on any effects directory. Finally, analyse the main program as if it were the body of a parameterless procedure.

The above algorithm is guaranteed to terminate because of the techniques of section 3.2. It remains, though, to organize the choice of the next procedure to be analysed in an efficient way.

The effects directory of a procedure depends on the effects directories of the other procedures it calls. If there were no recursion, we could simply start with the procedures that call no other procedures, and thereafter analyse a procedure only when all the other procedures it calls have been analysed. Then each procedure need be analysed only once, and accurate information is still obtained.

In the presence of recursion things are more complicated. A heuristic is wanted. Instead of choosing the procedure that calls <u>no</u> unanalysed procedures (since there may not exist one), it makes sense to analyse the one that calls (directly or indirectly) the fewest unanalysed procedures. Then we iterate to find the complete effects directories of this procedure and the ones it calls, directly or indirectly. After the iteration terminates, we have the complete analysis of several more procedures, and can go back to choose another one, if any remain, using the heuristic.

3.5 Summary.

This chapter outlines an algorithm which computes mstate sets that satisfy the consistency conditions. The results of the previous chapter then guarantee that the mstate sets are correct representatives of the run-time machine state at each instant during program execution. The next chapter will describe a program that implements the algorithm. This will provide some insight into the programming effort and computer time necessary to implement and use this analysis technique.

Chapter 4. Implementation.

This chapter contains a description of a program that implements a variant of the analysis algorithm. Most of the chapter is highly technical, but sections 4.1, 4.5, and 4.6 contain information that may be of more general interest.

4.1 Purpose and limitations of the program.

The algorithm outlined in chapters 2 and 3 was implemented while it was being developed, in order to fix the ideas and thus aid development by focusing attention on important details. In order to complete the job of programming an algorithm that, at the time, was itself under development, it was decided to reduce the goals. A number of restrictions were adopted.

Structures were not provided. As a result, since there are no field-selectors, the only operator in histories becomes _val_, the dereferencing operator. There is no '_delink_' operator. The processing of histories becomes very simple. However, it was originally planned to provide structures. The routines that handle histories form a separate module, so that it can be replaced independently of the rest of the program. The program will handle procedure-valued expressions, but not variables referring to procedures, nor procedures as parameters to other procedures.

The program text appears as an appendix. It is hoped that the redundancy it provides may be of some help to those who find some of the text of this thesis obscure. It may also be of some help to future implementors of optimizing analysers to see the blunders and successes in this one.

The program was written in ALGOL W, with a few language extensions defined in ML/1 [Brown 1].

4.2 External data representation.

The ALGOL W program described in this chapter processes programs written in the simple language. It parses them and builds an internal parse tree. It prints input verbatim while it is parsing. It analyses each program, computing state models and attaching them to points in the parse tree. Finally, it prints the program (possibly with redundant parentheses inserted or removed), together with selected state models.

4.2.1 Program representation.

Programs written according to the grammar in 2.2.2 and punched in the first 72 character positions of each line are accepted. Identifiers, reserved words, and other symbols must be separated from each other by blanks or end-of-line. Programs are separated from each other by lines which start with two periods.

Three extra kinds of <primary> are provided to enable selective display of state models and other information. These primaries are executed by placing the void value on the stack. '*display*' causes its initial state model to be printed with the results of the analysis. '*trace*' and '*notrace*' were used to debug the analyser and turn analysis-time tracing on and off.

4.2.2 Representation of state models.

State models are displayed in an indented manner (see Appendix B for examples). The state models are printed as a sequence of mstates.

Each mstate is printed with the title 'mstate', followed by its menvironment and mstack.

An menvironment is printed as a sequence of pairs <identifier> = <mvalue>.

The mstack is printed as a sequence of mvalues.

Mvalues are printed in such a way as to indicate their types and sometimes other information. Each reference mvalue is equipped with an arbitrary code (e.g. "RCCL08.292") that uniquely identifies it. Reference mvalues are described as individual or nonindividual. If an individual mvalue mrefers to another mvalue, this is stated. Routine mvalues are printed with their effects directories.

A history is printed by specifying the possible number(s) of dereferencing from each relevant identifier. As mentioned elsewhere, '10' means '10 or more'. Other integers stand for themselves.

Following the analysed program, each routine is printed together with its effects directory.

4.3 Internal data representation.

4.3.1 Program representation.

Both the program and the state models are represented using ALGOL W list-processing facilities. One of the serious restrictions in ALGOL W is the limit of 14 programmer-defined record classes. This requires careful conservation of record classes, leading to an inelegant multiple use of some record classes. Sometimes this has led to fields that are of no use in one interpretation of the record, but which are used in another.

The records used for representing programs are TRIO, DECL, IDENTIFIER, and CONSTANT.

CONSTANT records are used to represent constants. TYPE is a code representing the type of the constant. TYPE may be "BOOL", indicating boolean, or "NULL", indicating that the constant is the null reference. If TYPE is "BOOL", then the value, true or false, is in VALUEB, and VALUEKNOWN is true. All other fields are irrelevant for constants.

DECL records are used to represent declarations. FORMAL is a reference to the formal parameter, which is an identifier. ACTUAL refers to the actual parameter. DECLLINK is used by the program input routine to chain all declarations together into a symbol table.

IDENTIFIER records are used for identifiers. IDNAME is the name of the identifier. There are no other fields. Identifiers are represented by records, instead of by strings, so that all expressions can be handled in a uniform way as values of type REFERENCE(TRIO, IDENTIFIER, CONSTANT, DECL). ALGOL W does not permit a reference to a string, only a reference to a record containing a string.

TRIO is the most important record for programs. It can be considered as representing an operator with operands. It has fields for a single operator and up to three operands. The following are the permissible operators:

niladic	"GE"	a generator
	"TR"	*trace*
	"NT"	*notrace*
monadic	" "	a one-element list
	"VL"	*val* first
	"GO"	*go* first
dyadic	", "	a parameter list
	"; "	a list of statements in a serial
	": "	a label in a serial. MATTACH is used to attach a state model.
	":="	first := second
	"PR"	a procedure denotation. FIRST is the list of formal parameters, represented by declarations.

SECOND is the procedure body.

"CA" a call. first(second)

"WD" while first do second od. MATTACH is used to attach the initial state model of the condition.

triadic "IF" if first then second else third fi

4.3.2 Representation of state models.

4.3.2.1 State models.

A state model, which is a set of mstates, is represented by a doubly linked list of MSTATES records. Each MSTATES record refers to an MSTATE record and to its successor and predecessor on the list. It was necessary to have a separate level of records for the linked list, and not merely place the links in the MSTATE records themselves, so that one MSTATE could be on several independent linked lists.

4.3.2.2 Mstates.

An mstate is represented by an MSTATE record. If the mstate is a jumping mstate, then the field LABEL is not blank, and contains the jump target. Otherwise, LABEL is blank. SYMBOLTABLE points to a linked list of SYMBOL records which represents the environment. MSTACK is a linked list of TRIOs representing the mstack. The first TRIO is at the top of the mstack. The operator of each TRIO is " ". FIRST of each TRIO points to a CONSTANT, which represents an mvalue. SECOND points to the next TRIO on the mstack.

4.3.2.3 <u>Mvalues</u>.

Mvalues are represented by CONSTANTs. Somewhat more types are available than for CONSTANTS in program text.

HISTORY points to the history of the mvalue.

The TYPE of a CONSTANT can be "BOOL", meaning boolean, "REF", meaning reference, "PROC", meaning procedure, "NULL", meaning the null reference, or "   ", meaning that the type is unknown.

If the type is "BOOL", then the mvalue may be <u>true</u>, with VALUEKNOWN = VALUEB = <u>true</u>, or <u>false</u>, with VALUEKNOWN = <u>true</u> and VALUEB = <u>false</u>, or unknown, with VALUEKNOWN = <u>false</u>.

If the type is "REF ", then the mvalue is a nonnull reference mvalue. VALUEKNOWN=<u>true</u> iff this mvalue mrefers to some other mvalue. VALUER points to this other mvalue. ISINDIVIDUAL=<u>true</u> iff the mvalue is individual. If ISINDIVIDUAL=<u>false</u>, then VALUEKNOWN = <u>false</u>.

If the type is "PROC", then the mvalue is a procedure mvalue. Its effects directory is referred to by CEFFECTS.

If the type is "NULL", then the mvalue represents the null reference.

If the type is "   ", then the mvalue is the unknown mvalue.

4.3.2.4 <u>Effects directories</u>.

An effects directory is represented by an EFFECTS record. Its fields SET and ESCAPE point to histories which specify which tracks are set and which escape.

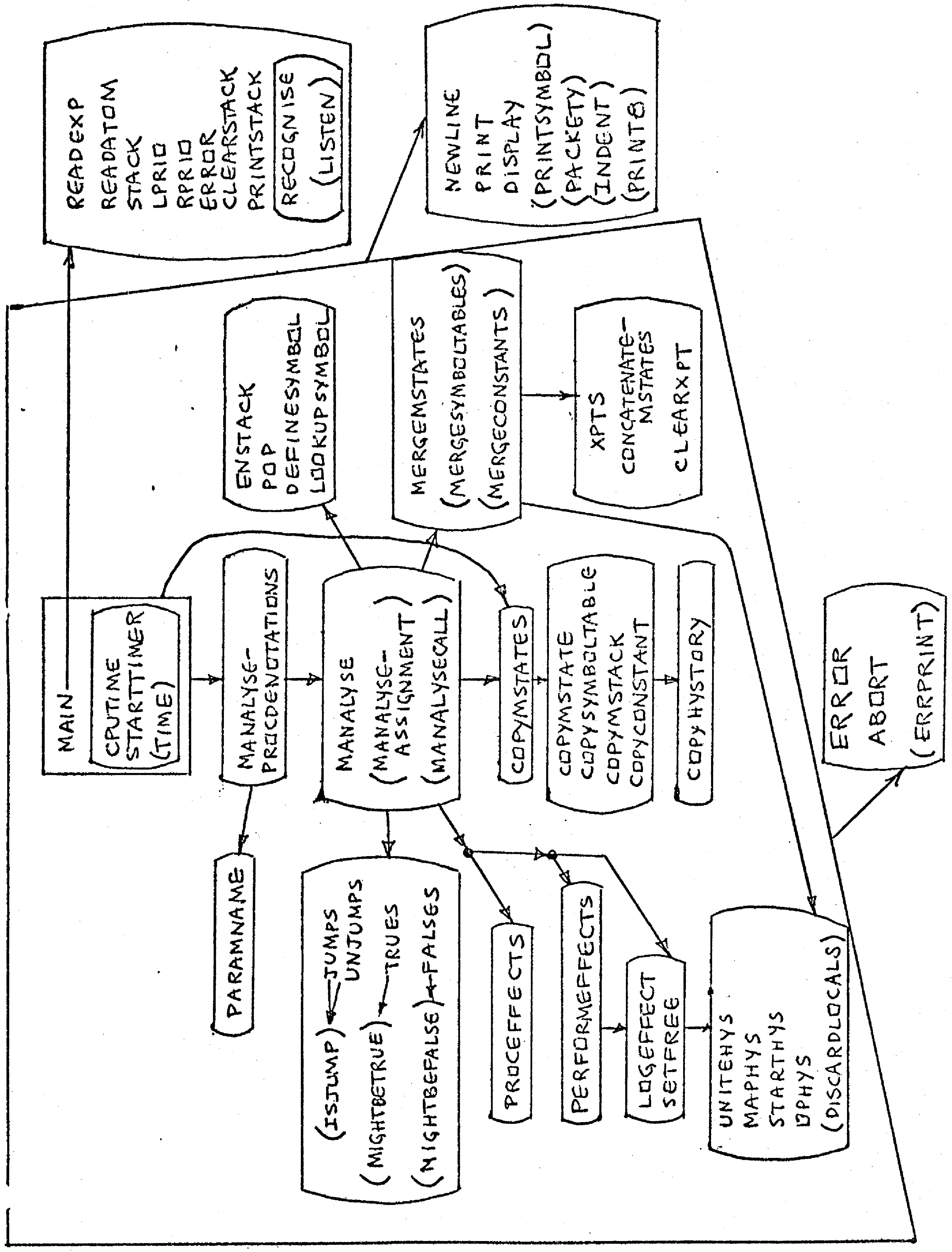
4.3.2.5 <u>Histories</u>.

Histories are represented as a linked list of HYSTORY [sic] records, linked by the NEXTHYS field. The start of the history is pointed to by the STARTH field. Since only one operator, <u>val</u>, can appear, only the possible numbers of operators need be kept for each start. A particular HYSTORY record specifies at least LOWH and at most HIGHH operators. 10 is used as a special code - it means 10 or more. We count 0,1,2,3,4,5,6,7,8,9,many.

4.4 The program.

4.4.1 Map.

The following page contains a map of all procedures in the program. The original attempt was to construct a chart indicating precisely which procedures called which other procedures, but the large number of procedures made this unwieldy. It was necessary to group the procedures into groups, and to indicate which groups called which other groups. Each group is indicated by a box in the diagram. Each box contains procedures which it defines for use elsewhere, and perhaps also some local procedures which are used only inside the box. Such local procedures are either parenthesized or enclosed in a smaller box with curved sides. An arrow from one box to another indicates that the procedures in the first box may call procedures defined by the second box. Within some of the smaller boxes, all arrows are omitted, since they are not very interesting, and would confuse the diagram by enlarging it. For example, MANALYSE may call JUMPS or NEWLINE, but not ISJUMP or PRINTSYMBOL.


READEXP
READATOM
STACK
LPRIO
RPRIO
ERROR
CLEARSTACK
PRINTSTACK
RECOGNISE
(LISTEN)
NEWLINE
PRINT
DISPLAY
(PRINTSYMBOL)
(PACKETY)
(INDENT)
(PRINT8)
ENSTACK
POP
DEFINESYMBOL
LOOKUPSYMBOL
MERGEMSTATES
(MERGESYMBOLTABLES)
(MERGECONSTANTS)
XPTS
CONCATENATE-MSTATES
CLEARXPT
MAIN
CPUTIME
STARTTIMER
(TIME)
MANALYSE-PROCDENOTATIONS
MANALYSE
(MANALYSE-ASSIGNMENT)
(MANALYSECALL)
COPYMSTATES
COPYMSTATE
COPYSYMBOLTABLE
COPYMSTACK
COPYCONSTANT
COPYHYSTORY
ERROR
ABORT
(ERRPRINT)
PARAMNAME
(ISJUMP)
JUMPS
UNJUMPS
(MIGHTBETRUE)
TRUES
(MIGHTBEFALSE)
FALSES
PROCEFFECTS
PERFORMEFFECTS
LOGEFFECT
SETFREE
UNITEHYS
MAPHYS
STARTHYS
DPHYS
(DISCARDLOCALS)

4.4.2 Macro extensions to ALGOL W.

The program was written in an extended ALGOL W. The extensions were implemented using P. J. Brown's macro-processor ML/1[Brown 1]. Some of the extensions are new language constructions to facilitate list processing; others are merely abbreviations to make coding less tedious.

4.4.2.1 Comment macro.

Comments may be inserted anywhere, enclosed by cent signs ("¢"). A comment begun with a cent sign is terminated by the next cent sign or end-of-line, whichever occurs sooner. The macroprocessor deletes these comments, and does not forward them to ALGOL W. This type of comment can be used almost anywhere in the program without disrupting the physical layout of the text.

4.4.2.2 Abbreviating macros.

"EXP" is short for "REFERENCE (TRIO, IDENTIFIER, CONSTANT, DECL)".

"REF" is short for "REFERENCE",

"PROC" is short for "PROCEDURE".

4.4.2.3 ABORT and ERROR macros.

```
ABORT "message"
ERROR "message"
```

The message may not contain any quotation marks ('"').

These macros generate calls to the procedure ERRPRINT. ABORT prints the message and goes to the label EXIT. ERROR just prints the message. The messages themselves are saved up and inserted into the expansion of the macro MESSAGES in the procedure ERRPRINT near the end of the program. ERROR and ABORT cannot be used textually after the procedure declaration for ERRPRINT.

4.4.2.4 Linked list looping macro.

```
<simple statement> ::= for <declarer option> <identifier> in list
          <expression> link <field selector> <while option> do
          <body>
<declarer option> ::= <empty>
     | <type>
<while option> ::= <empty>
     | while <boolean expression>
<body> ::= <simple statement>
     | open <block body> close
```

```
               for t:i in list l link s while w do d
```

expands to

```
     begin t i;
          i := l;
     LOOPSTARTc: if i=null then go to LOOPEXITc;
          if ¬( w ) then go to LOOPEXITc;
          d;
          i:=s(i);
          go to LOOPSTARTc;
     LOOPEXITc: end
```

If "t :" is omitted, then the declaration " t i ; " is omitted. If " while w " is omitted, then if 34(w) then go to LOOPEXITc; is omitted. "open" and "close" are used only for grouping and are not copied to output text. "c" is some integer, used to ensure unique labels.

Stupid restriction: The <simple statement> may not properly contain any begin-end statements. If the <simple statement> is a block, then the first begin must be on the same line as the do.

4.4.2.5 Macro for deletion from a list.

```
<simple statement> ::= delete <expression> from list <identifier>
          ll <selector 1> rl <selector 2>
```

<expression>, an element of a doubly linked list with left link <selector 1> and right link <selector 2> starting with <identifier>, is deleted from the list. The macro expansion for

```
              delete e from list l ll left rl right
```

is

```
     begin if e=l then l:=right(l);
          if left(e) =null then right(left(e)):=right(e);
          if right(e) =null then left(right(e)):=left(e)
     end
```

4.4.2.6 <u>Concatenation macro</u>.

```
<primary> ::=
     concatenate mstates ( <expression list> )
     | concatenate hystories ( <expression list> )
```

These macros produced a nested set of calls to the procedures CONCATENATEMSTATES and CONCATENATEHISTORIES respectively, to concatenate the values of the expressions, which are lists, in the order in which they are coded.

4.4.2.7 <u>Macro for creation of CONSTANT records</u>.

```
CONSTANT ( TYPE = <expression>,
     VALUEKNOWN = <expression>,
          .
          .
          .
          )
```

Any of the parameters may be omitted, provided that at least one is specified.

This macro expands to a record class designator to create a CONSTANT record, each of whose fields is initialized, if specified, to the specified value, and otherwise, to a neutral default value.

4.4.3 Output routines.

The routines used for output are NEWLINE, PRINT, PRINTSYMBOL, and DISPLAY. They use a global variable INDENTATION, which is initialized to zero in the global initialization code. INDENTATION specifies how much the current line is to be indented.

NEWLINE causes a new line to be started, and then writes three spaces for each column of indentation.

PRINT accepts a pointer to any of the record classes used in the program, and prints the tree starting at that record in a sensible format. If it receives a SYMBOL record, it prints the entire symbol table from that record on.

PRINTSYMBOL accepts a pointer to a SYMBOL record and prints just that record, not the entire symbol table.

DISPLAY is used to print programs in a readable, indented format. It will cause the state models attached to the statement 'DISPLAY' to be printed as well. DISPLAY uses PACKETY, which decides whether to parenthesize a phrase, and INDENT, which causes a phrase to be indented. PRINT should really use INDENT too, but it was written before INDENT, and was not subsequently changed.

PRINT8 prints an identifier with some trailing blanks suppressed.

4.4.4 Input.

The input routine is READEXP, which contains a number of internal procedures, READATOM, RECOGNISE, CLEARSTACK, PRINTSTACK, STACK, LPRIO, RPRIO, and ERROR (another one, not the macro). It uses double operator priority techniques to parse a program, and stores it in list structure form.

It does very little error checking, and does not produce clear error messages. It accepts programs written according to the grammar in 2.2.2, with an extension. It has a very cheap lexical scanner. Identifiers, reserved words, and other symbols must be separated from each other by blanks or end-of-line. The parser is not of great interest.

The extension is:

<primary> ::= display | trace | notrace .

Display, trace, and notrace each evaluate to the void value. 'display' indicates to MANALYSE and DISPLAY that its initial state model is to be saved and printed. 'Trace' and 'notrace' turn tracing of the analyser on and off.

ERROR is called to handle some errors. It prints "ERROR" followed by an error number, and then terminates analysis of the current program by GOing TO EXIT.

4.4.5 Merging.

Merging is coded as several procedures. The main one, MERGEMSTATES, will merge two state models. It calls CONCATENATEMSTATES, MERGESYMBOLTABLES, MERGECONSTANTS, XPTS, and SETFREE.

The basic technique is that of the growing equivalence relation described in Chapter 3. The equivalence relation is represented in the way invented by M. J. Fischer and B. A. Galler [Knuth 1, volume 1, 2.3.3, pp 353-355]. The XPT field of CONSTANT records is used to link together equivalent mvalues in an inverse tree that leads to a representative of the equivalence class. This last mvalue, the representative, is the merge of all the mvalues in its class, and bears, in the field XCT, the number of mvalues it represents (including itself). A merged mvalue can be individual only if the original mvalues were individual and XCT is 2.

XPTS simply chains down a list of mvalues using the XPT link to find the last one. It is used to find the canonical representative of an equivalence class.

SETFREE records that its argument, C, has escaped. If C mrefers to another mvalue, then, recursively, this other mvalue is also set free.

4.4.6 Copymstates.

Given a reference to an MSTATES list, representing a state model, COPYMSTATES will construct an entirely new copy of that state model, which shares no storage with the original one, except for identifiers and generators which are the starts of histories. COPYMSTATES uses the XPT fields to point from the old mvalues to the new, in order to copy circular lists. The XPT fields are subsequently cleared. The XPT and XCT fields are not copied. They are set to the neutral values null and 1 instead.

COPYMSTATES calls COPYSYMBOLTABLE, COPYCONSTANT, COPYHYSTORY, COPYMSTACK, and CLEARXPT to do its job.

4.4.7 Mstate selectors.

JUMPS, UNJUMPS, TRUES, and FALSES select from a state model the set of its mstates that satisfy certain conditions. These mstates are then linked in a new chain of new MSTATES records.

JUMPS selects the jumping mstates

UNJUMPS selects the nonjumping mstates.

TRUES selects those mstates which have a top element on their mstacks which might represent *true*; i. e., it is of unknown type, or is boolean with unknown value, or it is boolean and *true*.

FALSES selects those mstates which have a top element on their mstacks which might represent *false*.

TRUES and FALSES call MIGHTBETRUE and MIGHTBEFALSE. JUMPS calls ISJUMP.

4.4.8 Analysis.

This was meant to be a single procedure, MANALYSE, but the ALGOL W restriction on the size of procedures forced it to be broken into several fragments. MANALYSE is a straightforward coding of Chapter 2. Its fragments are MANALYSE, MANALYSEPROCDENOTATIONS, MANALYSEASSIGNMENT, and MANALYSECALL. MANALYSECALL calls PERFORMEFFECTS.

Each of these takes an expression to be analysed and an initial state model as parameters. Each returns a final state model as value.

4.4.8.1 PARAMNAME.

According to 2.9.8.2, a new environment is constructed which contains new identifiers. PARAMNAME constructs these new identifiers. The new identifiers are used everywhere in histories and effects directories for formal parameters.

4.4.9 Effects handling procedures.

PROCEFFECTS looks in the arrays PROCTABLE and EFFECTSTABLE for the effects directory of the procedure denotation passed to it as parameter.

LOGEFFECT adds an effect to the effects directory of the procedure denotation now being analysed. This effects directory is identified by the variable CURRENTEFFECTS.

4.4.10 History handling procedures.

The procedures (except for copying and printing) that handle histories have been collected together in one place to make it easy to modify them, for example, to add operators other than val).

OPHYS tacks an extra operator onto each track of a history.

CONCHYS has two arguments, both histories. CONCHYS tacks the operators of the first history onto the second history. This is to implement the concatenation of tracks in step 1 of the analysis of procedure calls (2.8.8.2).

STARTHYS converts an identifier or generator into a one-track history that starts with that identifier or generator and has no operators.

UNITEHYS constructs a new history which is a (possibly improper) superset of the union of its arguments, which are both histories. CHANGE is set to _true_ if the new history is different from the first parameter.

MAPHYS is a functional. It has three arguments, a procedure FN, a HYSTORY record HYS, and an mvalue START. MAPHYS implements passing through. For each track T in the part of the history described by HYS, for each mvalue V that the operators of T pass through starting from START, the procedure FN is called with V as parameter.

4.4.11 Miscellaneous procedures and variables.

ENSTACK places a CONSTANT on top of the mstack of an MSTATE.

POP removes the top I elements from the stack of an MSTATE, or from the stack of each MSTATE in an MSTATES list.

COPYBASE copies an identifier, generator, or CONSTANT. COPYBASE is used by the history handling procedures. The double angle brackets around the word <<CONSTANT>> on the tenth line serve to prevent it from being recognized as a call to the CONSTANT macro.

DEFINESYMBOL inserts a new identifier-mvalue pair into an environment.

LOOKUPSYMBOL finds the mvalue mpossessed by an identifier, given the identifier and the environment.

ERRPRINT is called from the ABORT and ERROR macro expansions. Its constituent macro MESSAGES expands to all the messages used as parameters to ABORT and ERROR before this point in the program.

SUPERINIT is the initial state model used at the beginning of each procedure body (except that it has no dummy formal parameters) and at the beginning of the entire analysed program. SUPERINIT is always copied, and never used directly.

CHANGEEFFECTS records whether any effects directory has been changed. It is set to _false_ by MANALYSEPROCDENOTATIONS, and to _true_ by UNITEHYS, when called with CHANGEEFFECTS as parameter.

4.5 Empirical results.

The analyser was used to analyse a number of small programs. The output from the analyser can be seen in Appendix B. The programs analysed were small because it is difficult to write convincing large programs in the simple programming language. Further, a somewhat illegitimate trick was used. The analyser does not distinguish between an undefined value and an unknown value. An undefined value is one which the program fails to provide during execution, e.g., the value referred to by a variable for which no assignments have yet been executed. An unknown value is one which the analyser does not know. In the test cases, to obtain expressions that were too complex for the analyser to evaluate, uninitialized variables were dereferenced.

4.5.1 Discussion of examples.

Example 1.

This is a variation on the first example in Chapter 1. A note about the output format is in order. An mstate is printed with an menvironment and an mstack. The environment contains two identifiers, XX and YY. These identifiers mpossess the individual mvariables which are identified by arbitrary codes. They, in turn, both mrefer to the same individual reference, which mrefers to true.

Example 2.

This example illustrates a simple loop that scans a linked list.

Example 3.

This example contains a simple loop in which an mvalue is passed from one variable to another (from val B to val A). Although the variable referred to by the variable possessed by B is not the same after one cycle through the loop, it is still represented by the same mvalue, because it has the same properties.

<u>Example 4</u>.

Loops can also be built from <u>go to</u> statements.

<u>Example 5</u>.

A procedure with side effects. The effects directory of the procedure is printed at the end, divided into two parts, 'set' and 'escape'. In each part we find the tracks that are set or set free. Note that '10' means '10 or more' if it occurs in this context.

<u>Example 6</u>.

Another procedure. This one, however, does not make B escape. The assignment to A merely provides a local handle whereby P can manipulate B if it should so choose.

<u>Example 7</u>

The local handle B is not so local. It is assigned to C, which is outside the block. Therefore, B and A escape.

<u>Example 8</u>.

A recursive procedure which passes its arguments like a shift register shifting left. Note that B and C both escape, although the analyser must sift through several layers of recursion to discover this.

<u>Example 9</u>.

This loop has ten variables, which it forms into a shift register. One might expect that this type of program would make the analyser take longest, especially with complex mvalues. The single false from A propagates down the sequence, rendering one more variable unknown each time the analyser processes the loop.

4.5.2 Execution time and storage space.

The analyser was made to display the CPU time used for reading and parsing submitted programs, for performing the analysis, and for printing the results. In most cases the analysis time was of the same order of magnitude as the reading and printing time. This seems quite encouraging. The longest program tested (example 9) took significantly more analysis time than read time, but it was designed specifically to do so. It is unlikely that such programs will be common in real life. It would have been nice to concoct a convincingly realistic large program, but the limitations of the simple programming language make it difficult for a large program to appear realistic. No time or space bounds have been derived theoretically for the analyser. Because of the semi-interpretive nature of the analyser, it is likely to be nearly as difficult to derive realistic time and space bounds for this analyser as to derive them for execution of arbitrary programs.

The program is 1950 lines long, including macro definitions. This produces 1960 lines of ALGOL W code, and 62308 bytes of IBM/360 object code (including run-time checking). Program runs happily in the 160K bytes of data storage space.

4.6 Improvements and comments.

4.6.1 Data representation.

At many times during the writing of the program it was necessary, or would have been convenient, to use a loop that iterated through all mvalues of a given mstate. If all the mvalues of a single mstate had been represented as an array, this would have been very simple. With the list structures used, a garbage-collector-like scan was needed for this, marking those mvalues already processed. This led to complications when the function of the loop was to modify the list structure. I was often reluctant to use such a loop and found another algorithm if possible. The routines for merging and copying, in particular, would have been simpler, more comprehensible, and perhaps more efficient, if arrays had been used. An array should also be used for the environment, and perhaps also the stack. Subscripts within the arrays should be used as pointers.

4.6.2 Program representation.

In this version of the analyser, programs were represented as tree structures rather like their parse trees. This was to preserve the nesting structure of loops, *if* - *then* - *else* - *fi* statements, and so forth. It is reasonable to suppose that knowledge of nested control structure can improve efficiency of the analysis. (Indeed, serious work has been done [Allen 2] on discovering the nested control structure of programs written in languages which do not indicate it syntactically.) However, assignments, generators, calls, dereferencing, and the like have very little important nesting structure. It might be easier to work with postfix code for such parts of a program. A program could then be compactly stored as an array of instructions, with a few special tricks for remembering nested control structure. Whether such a representation would truly be an improvement is hard to say.

4.6.3 Programming language in which the analyser is written.

ALGOL W was chosen for implementing the analyser for a number of reasons. It was available in an efficient, reliable implementation. It provides *if* - *then* - *else* conditionals and *while* - *do* loops, which facilitate clear coding. It possesses list-processing facilities with garbage collection. This permits one to ignore the spectre of storage misallocation, thereby making debugging simpler. Most important for debugging, it provides pointer security -- it is impossible to use a pointer pointing to a record of one class as if it pointed to another. This greatly reduces the chances of obtaining an incomprehensibly scrambled data structure.

The major drawback with ALGOL W was one that could not be patched by a macro extension. It was impossible to have pointers pointing to arrays or to have arrays in garbage-collected storage. It was this that prevented using the data representations of the previous sections.

It is interesting to note that ALGOL 68 possesses the above advantages, but none of the drawbacks, except for the overriding (and I hope temporary) one of the nonavailability of an implementation.

Chapter 5.

Extensions and further research.

5.1 Summary.

This thesis has presented a technique for analysing the source text of a program to determine information about the data structures it creates and maintains at run time. The algorithm identifies a special class of variables, the individual variables, which can be treated by conventional optimizing techniques, despite the presence of pointer variables in the program being analysed. Variations on the algorithm presented here might be useful for detecting some errors in programs that might otherwise be left to be detected at run time. The algorithm maintains the theoretical distinction between values, representatives of values, and the names used for values in the source program.

This chapter suggests some lines for further development of these techniques.

5.2 Handling further language features.

5.2.1 Procedure variables and procedures as parameters.

ALGOL 68 permits a variable to refer to a routine, and also permits an actual parameter in a procedure call to be a routine. The simple language used in this thesis does not. The difficulty lies in finding and applying the effects of the routine when it is subsequently called.

At first sight, it seems that we need only find all the procedures that might be involved, and take the union of their effects. If the identifier V in the call V(...) mpossesses an individual mvariable, this may be easy: its value may be known. If its value is not known, it is more difficult.

One might try to classify routines as individual and nonindividual, depending on whether it is known which variables refer to them. Then, when an unknown routine is called, one can be sure it is one of the nonindividual routines, and use the union of all the effects directories of all the nonindividual routines. Routines will be individual and nonindividual in much the same way as variables.

Next, once one has an effects directory created from this union, one must apply it. In the cases discussed in Chapters 2 and 3, the global identifiers of a procedure being called were always known at the point of call. Further, the environment of the routine was a subset of the environment at the point of call -- the identifiers they had in common possessed the very same objects. Once procedures can be passed around as parameters, this ceases to work. Consider the following example:

```
int ii, jj;
proc p = (proc void q, ref int j) void: ( --; q; --);
begin int jj, k;
    proc qq = void: ( --; j:=0; k:=0; --);
    p(qq,jj)
end
```

When q (i. e. qq) is called from p, the identifiers affected by q are not known at the point where p is declared. It is therefore not possible to properly describe the effects of calling q, or p.

It might be possible to define 'scopeless' effects, which are not deleted when their apparent range expires (2.9.6.3), but stick around to affect all identifiers of the same name, no matter how many layers of recursion or parameterisation happen. This solution is crude, but it may be effective.

5.2.2 Label variables and labels as parameters.

Label variables can be treated using the mechanism for procedure variables. Indeed, ALGOL 68 forces the programmer to use procedure variables instead of label variables.

But they do not need all of this mechanism, since they are simpler entities. A jump to a label cannot have the side effects on the state that a procedure call can. It does not change the values of any variables. The possible targets of a go to can be found in much the same way as the procedures that might be called by a call.

For each local target, the mechanisms already explained suffice, and for each nonlocal target, an effect can be logged. It still requires further thought to determine that this technique does indeed work.

5.3 New techniques.

5.3.1 Merging and universal algebra.

Here some insight can come from universal algebra. [Gratzer 1]. (This model was, in fact, used in developing the merging algorithm). An mstate can be considered as a partial algebra. The identifiers in the environment are nullary operators delivering values as function values. The positions in the stack (top, second, third, etc.) are also nullary operators. The mappings 'mrefer' and 'mlink' become a unary partial operators. The problem is simply to construct another algebra, of the same type as the original two, that is a homomorphic image of a subalgebra of each original mstate.

The new algebra can be obtained by the following steps:

1. Consider the disjoint union of the original mstate sets.

2. Construct an equivalence relation on this union.

3. Extend this equivalence relation to a congruence relation, deleting occasional mvalues, and logging effects.

4. Take the modified union modulo the congruence relation as the new mstate.

5. Decide which of the new mvalues are to be individual.

5.3.2 Optimum representation of histories?

If there is no recursion in a program, it is possible to find an optimum representation for histories. These histories are optimum in the sense that they are the smallest histories permitted by the consistency conditions. The histories will, in fact, be regular sets.

Suppose we have completed the analysis of all procedures called by the particular procedure which we are now going to analyse. First, we perform the analysis of our procedure as usual, except that we do not bother to construct any histories. This is possible because histories are needed only to construct the effects directory of the procedure. Since the procedure is not recursive, the effects directory is not yet needed.

The histories, which will be regular sets, will next be determined. We construct a grammar for these regular sets: Consider each mvalue in each mstate in each mstate set to be a nonterminal. If the history of mvalue A is required, by the consistency conditions, to contain that of B with an operator O added, place a production

$$A \rightarrow O\ B$$

in the grammar. If an mvalue A is mpossessed by the identifier I insert the production

$$A \rightarrow I.$$

The language generated by these productions using an mvalue as start symbol will be the history of that mvalue. There are well-known methods for handling regular languages: they will not be discussed here.

This technique looks attractive. However, it has several drawbacks. First, the computations required to produce an effects directory in some convenient form may well be complex. Second, the technique does not work correctly in the presence of recursion. It might be suspected that we simply have to generalize to context-free grammars. The complex manner in which the effects directory of a procedure being called causes mvalues (i.e. nonterminals) to be deleted makes this hope seem forlorn.

5.3.3 Bit matrix techniques.

Mstates can be partially represented by bit matrices. Columns correspond to mvalues, and rows to mvalues, identifiers, and stack positions. The matrix then represents the relation 'mpossesses'. This seems inefficient, however, since there will be at most one bit in each row. However, if we always provide the 'unknown' mvalue as both a row and a column, we might be able to handle the division of the data structure into parts as suggested in section 5.2.2. First, we treat mpossession in a nonstandard way. Instead of placing a bit in the matrix if one mvalue/identifier/stack position is known to refer to/possess/contain the other, we place a bit if it might. Merging mstates becomes taking the inclusive or of two bit matrices. An identifier mpossesses an mvalue other than 'unknown' only if it has only one one bit in its row. An mvalue is individual only if there is only one bit in its column, with that bit not in the 'unknown' row.

Histories still have to be handled separately.

An analyser using this representation will not have as many individual variables as the analyser of Chapter 4, since only one mvalue is permitted to refer to each individual mvalue. However, it has available quite a lot of information not available at all to the analyser of this thesis. Proof is still required that this added information can be maintained correctly.

Frances Allen [Allen 1] describes a technique similar to this, except that she uses only a single bit matrix. This is as if a single mstate were used to describe all run time states instead of just those associated with a single point in the program. Each row or column of the rather large matrix corresponds to a variable, expression, procedure, on-condition, or other language construction appearing in the program. A bit in the matrix indicates whether an action on the construction associated with its row can affect the construction associated with its column. The transitive closure of this matrix is used in performing optimizations. This technique is clear and simple, but lacks sharpness of discrimination concerning temporary associations between program constructions.

5.4 Applications.

5.4.1 List structures and capability.

Data structures that programs maintain at run time may well be divided into independent disjoint parts. Alteration of one such part should not affect other parts. For example, if one tree in the data structure is altered, other disjoint trees are not affected. It might be possible to localize an effect on a nonindividual variable to one variable or group of variables, and know that others cannot be affected. This kind of classification might be used to permit common subexpression elimination even for nonindividual variables -- unaffected variables have constant values.

Such divisions into parts might be feasible using the idea of 'capability'. An identifier or value is a capability that permits access to another value if there is a path through the list structure from that identifier or value that can be followed by standard program operations (e.g. dereferencing, subscripting, field-selection). The sets of values that can be accessed from identifiers may be the above-mentioned parts of the program data structure. It remains to identify disjointness. This can be done by finding assignments from one part to another. If there are any such assignments, then the parts are not disjoint. Nondisjointness is then extended to an equivalence relation. Effects directories will then have to record which parts procedures render nondisjoint.

These ideas of 'capability' are reminiscent of the protection schemes proposed by Lampson[Lampson 1, Spooner 1].

5.4.2 <u>Scopes</u>.

ALGOL 68 has a rigorous scheme of scope checking, which has not yet been discussed in this thesis. If scope information is attached to mvalues, it may become possible to do much of the run-time scope checking at compile time, thereby saving much execution time at no cost to security. It may also be possible to use scopes in the analysis itself, since they provide a simple and far-reaching restriction. If V permits access to W, then the scope of V is no larger than the scope of W.

5.4.3 Storage allocation optimization.

The additional structure provided by scopes in ALGOL 68 can also be a nuisance. If a programmer wishes to use complex data structures, such as data structures that contain procedures, then he must fight the scope restrictions every step of the way. Still, it seemed impossible to remove the scope restrictions without abandoning either security or the efficient stack method of storage allocation. Oregano [Berry 1], an ALGOL 68 variant, has indeed abandoned the stack, and uses garbage collected storage for everything.

Some of this waste in Oregano is unnecessary. Each variable that never becomes nonindividual can be allocated on the stack. Any assignment of the variable it represents to a global variable would cause it to become nonindividual.

References.

[Allen 1] Frances E. Allen, A basis for program optimization, IBM Thomas J. Watson Research Center technical report RC3138.

[Allen 2] Frances E. Allen, Control flow analysis, Proceedings of a symposium on compiler optimization, Sigplan notices, 1970 July, pp 1-19.

[Berry 1] Daniel Berry, Introduction to oregano, in Proceedins of a symposium on data structures in programming languages, Sigplan notices, 1971 v 6 n 2 February, pp 171-190.

[Brown 1] P. J. Brown, The ML/1 macro processor, CACM 1967 v 10 n 10 pp 618-623.

[Cocke 1] John Cocke and J. T. Schwartz, Programming languages and their compilers, Courant Institute of Mathematical Sciences, April 1970.

[Dijkstra 1] E. W. Dijkstra, Go to considered harmful (letter) CACM 1968 v11 n3 March pp 147-148.

[Dijkstra 2] E. W. Dijkstra, Cooperating sequential processes, in Programming languages, Genuys (editor), NATO Advanced Study Institute, Academic press 1968.

[Gratzer 1] George Gratzer, Universal algebra.

[Knuth 1] D. E. Knuth, The art of computer programming.

[Lampson 1] Butler W. Lampson, Protection, in Proceedings of the fifth Princeton conference on information sciences and systems, pp 437-443.

[Landin 1] The next 700 programming languages, CACM 1966 v9 n3 March pp157-166.

[Lucas 1] P. Lucas, P. Lauer, H. Stigleitner, Method and notation for the formal definition of programming languages, IBM@ laboratory vienna technical report TR 25.087, 28 June 1968.

[Sites 1] Richard L. Sites, ALGOL W reference manual, Stanford University computer science department.

[Spooner 1] C. R. Spooner, A software architecture for the seventies part I - the general approach, Software practice and experience, v1 n1, pp 5-38.

[Van Wijngaarden 1] Van Wijngaarden, Mailloux, Peck, and Koster, Report on the algorithmic language ALGOL 68.

[Wirth 1] Niklaus Wirth and C. A. R. Hoare, A contribution to the development of Algol, CACM 1966 v 9 n 6 June pp 413-431.

7. Appendix A. Program listing.

```
MCSKIP MT,< WITH < > WITH >
MCINS << & . >>
MCSKIP = WITH = NL
MCSKIP ¢ OPT ¢ OR NL ALL
MCSKIP T, < WITH '  ' WITH >
MCDEF<<()>>AS<<<<(>>&B1.<<)>>>>
MCDEF<<EXP>> AS<<REFERENCE(TRIO,IDENTIFIER,CONSTANT,DECL)>>
MCDEF<<REF>>AS<<REFERENCE>>
MCDEF <<PROC>> AS <<PROCEDURE>>
MCSKIP DT,<<"">>
MCDEF 5 VARS <<FOR OPT : N1 OR N1 IN WITHS LIST LINK
         OPT WHILE N2 OR N2 DO
            OPT END NO
             OR ; NO
             OR ELSE NO
             OR CLOSE NO
             ALL
         OR DO WITHS OPEN CLOSE
         OR DO WITHS BEGIN END
         ALL
      OR : WITH = ALL>>
AS<<MCGO L5 IF &WD1.=<<:=>>
MCSET T4=2
MCGO L1 IF &WD1.=<<:>>
MCSET T4=1
&L1.MCSET T5 = &&T4.+4.
MCGO L2 IF &WD&T4.+2.=<<WHILE>>
MCSET T5=&&T4.+3.
&L2.BEGIN MCGO L3 IF &T4.=1
&A1. &A2.;&L3.
&A&T4..:=&A&T4+1..;
LOOPSTART&T2.:IF &A&T4..=NULL THEN GO TO LOOPEXIT&T2.;
MCGO L4 UNLESS &WD&T4.+2.=<<WHILE>>
IF ¬(&A&T4.+3.) THEN GO TO LOOPEXIT&T2.;
&L4.MCGO L6 IF &WDT5.=<<END>>
&BT5.MCGO L7
&L6.BEGIN&BT5.END&L7.;
&A&T4..:=&A&T4.+2.(&A&T4..);
GO TO LOOPSTART&T2.;
LOOPEXIT&T2.:
END MCGO L0
&L5.<<FOR>>&B1.<<:=>>>>
MCDEF<<IF THEN OPT ELSE N1 OR N1 ; NO OR END NO OR ) NO
                     OR CLOSE NO ALL>>
AS<<<<IF>>&B1.<<THEN>>&B2.==
MCGO L0 UNLESS &WD2.=<<ELSE>>
<<ELSE>>&B3.>>
MCDEF <<WHILE DO OPT ELSE NO OR ; NO OR END NO OR CLOSE NO ALL>>
AS<<<<WHILE>>&B1.<<DO>>&B2.==
>>
MCDEF<<DELETE FROM WITHS LIST LL RL
  OPT ; NO OR END NO OR OD NO OR ELSE NO ALL >> AS ==
```

7. Appendix A. Program listing.

```
<<BEGIN IF &A1.=&A2. THEN &A2.:=&A3.(&A1.);
   IF &A3.(&A1.)¬=NULL THEN &A4.(&A3.(&A1.)):=&A4.(&A1.);
   IF &A4.(&A1.)¬=NULL THEN &A3.(&A4.(&A1.)):=&A3.(&A1.)
   END>>
MCDEF <<MESSAGE>> NL
 AS<<MCSET P2 = &P2.+1
MCDEFG ERMESS&P2. AS <'<<'>&B1.<'>>'>
&P2.>>
MCDEF<<ERROR WITHS " ">>
SSAS<<ERRPRINT(FALSE,MESSAGE&WB1.
)>>
MCDEF<<ABORT WITHS " " >>
SSAS <<ERRPRINT(TRUE,MESSAGE&WB1.
)>>
MCDEF 4 VARS <<MESSAGES>>
AS<<MCSET T4=1
MCGO L2
&L1.;
MCSET T4 = &T4.+1
&L2.MCGO L0 IF &T4.GR&P2.
MCDEF <<XXX>> AS <<ERMESS>>&T4.
WRITE(<'"'>XXX<'"'>) MCGO L1
>>
MCDEF<<STATDELS>> AS <<<<OPT ELSE NO OR ; NO OR END NO OR ) NO
         OR CLOSE NO ALL>>>>
MCDEF << DEFCONC NL >> AS <<==DE
MCDEFG 4 VARS<<CONCATENATE WITHS >>&A1.<< WITHS ( N2 OPT,N2 OR )
                                                  ALL>>
AS<<MCSET T4=1
&L1.CONCATENATE>>&A1.<<<<(>>&AT4.,MCSET T4=&T4.+1
MCGO L1 IF &WDT4.=<<,>>
&AT4.MCSET T4=1
&L2.<<)>>MCSET T4=&T4.+1
MCGO L2 IF &WDT4.=<<,>>
>>
>>
DEFCONC <<MSTATES>>
DEFCONC <<HYSTORIES>>
MCSKIP<< ( WITHS DUMMYFIELDS WITHS ) >>
MCSKIP <<SKIP>>
MCSET P1=0
MCSET P2=0
MCDEF 4 VARS <<CONSTANT WITH ( N1 OPT = N2 OR N2 , N1 OR ) ALL >>
AS<<MCSET T4=0
MCSKIP NL WITH SPACES
MCDEF F WITH ! WITH TYPE AS <<"    ">>
MCDEF F WITH ! WITH VALUEKNOWN AS FALSE
MCDEF F WITH ! WITH VALUEB AS FALSE
MCDEF F WITH ! WITH VALUER AS NULL
MCDEF F WITH ! WITH CEFFECTS AS NULL
MCDEF F WITH ! WITH ISINDIVIDUAL AS FALSE
MCDEF F WITH ! WITH IDENTITY AS 0
```

7. Appendix A. Program listing.

```
MCDEF F WITH ! WITH HISTORY AS NULL
MCDEF F WITH ! WITH SETMARK AS FALSE
MCDEF F WITH ! WITH USEMARK AS FALSE
MCDEF F WITH ! WITH ESCMARK AS FALSE
MCDEF F WITH ! WITH XMARK AS "         "
MCDEF F WITH ! WITH XPT AS NULL
MCDEF F WITH ! WITH XCT AS 0
MCDEF F WITH ! WITH SCANMARK AS FALSE
&L1.MCGO L4 IF &WDT4.=<<)>>
MCSET T4=&T4.+1
MCGO L2 IF &WDT4.=<<=>>
MCGO L1 IF &AT4.=<<NONINDIV>>
MCGO L3 UNLESS &AT4.=<<INDIV>>
MCDEF << F WITH ! WITH ISINDIVIDUAL>> AS TRUE
MCGO L1
&L2.MCDEF F WITH ! WITH &AT4. AS <'<<'>&WA&T4.+1.<'>>'>
MCSET T4 = T4 + 1
MCGO L1
&L3.MCNOTE<<INVALID FIELD>>
MCGO L1
&L4.<<CONSTANT>>(F!TYPE,F!VALUEKNOWN,F!VALUEB,F!VALUER,
                    F!CEFFECTS,F!ISINDIVIDUAL,
                    F!IDENTITY,
                    F!HISTORY,
                    F!SETMARK,F!USEMARK,F!ESCMARK,
                    F!XMARK,F!XCT,F!XPT,F!SCANMARK)>>
```

DATA STRUCTURES

```
BEGIN
  RECORD TRIO(STRING(2) OP;
        EXP FIRST, SECOND, THIRD;
        REF(MSTATES) MATTACH);
        ¢ TRIO is used for operators with zero to three
                                            arguments,
        ¢ which are EXPressions.
        ¢
        ¢ NULLARY "GE" generator
        ¢ UNARY:  "  " a list of one element.
        ¢          "VL" val
        ¢          "GO" A JUMP. FIRST IS THE JUMP TARGET
        ¢BINARY:  ", " parameter list, formal or actual.
        ¢          "; " statements in a serial.
        ¢          ": " place a label.
        ¢          ":=" assignment
        ¢          "PR" procedure denotation. FIRST is the list
        ¢               of formal parameters, represented as a
        ¢               "  " or ", "-list of DECLarations.
        ¢               SECOND is the body of the procedure.
        ¢          "CA" a call. FIRST is the function;
        ¢                       SECOND is the actual parameter
                                            list.
        ¢          "WD" while first do second od
        ¢ TERNARY "IF" if FIRST then SECOND else THIRD fi

  RECORD IDENTIFIER(
        STRING(8) IDNAME ¢ the string which represents the
                                            identifier
                  ¢ in the source text. ¢ );
  RECORD DECL(REFERENCE(IDENTIFIER)FORMAL;
              EXP ACTUAL;
              REFERENCE(DECL)DECLLINK ¢ the link used by
                  ¢ FIXIDENTIFIERS to link together its symbol
                                            table¢);
  RECORD MSTATE(STRING(8) LABEL; ¢ the jump target ¢
        REF(SYMBOL) SYMBOLTABLE; ¢ the current environment ¢
        EXP MSTACK ¢ The model of the run-time stack,
                                            represented
                  ¢ as a "  "-linked list of CONSTANTs ¢);

        ¢ An mstate is represented by a linked list of MSTATE
        ¢ records. Each one represents the state of the run-time
        ¢ machine, usually upon completion of elaboration of
                                            some
        ¢ expression. ¢

  RECORD MSTATES(REF(MSTATE) THISMSTATE;
        REF(MSTATES) NEXTMSTATE,LASTMSTATE);

  RECORD SYMBOL(STRING(8) SNAME;
        REF(CONSTANT, TRIO) POSSESSION;
```

DATA STRUCTURES

```
        REF(SYMBOL)NEXTSYMBOL);

   RECORD <<CONSTANT>>(STRING(4) TYPE; ¢ "BOOL" or "REF" or
                                          "proc"
                                     ¢ or "    "(unknown) or "NULL
                                          " ¢
        ¢ Next,the value of the constant, in one of two fields,
        ¢ depending on the type. ¢
        LOGICAL VALUEKNOWN;
         LOGICAL VALUEB; ¢ if bool,true or false ¢
        REF(CONSTANT) VALUER; ¢ if ref, value referred to ¢
        REF(EFFECTS) CEFFECTS ¢ if a procedure ¢ ;

        LOGICAL ISINDIVIDUAL;
        INTEGER IDENTITY; ¢ This field is not used ¢
        REF(HYSTORY) HISTORY ¢ how we got this value ¢ ;
        LOGICAL SETMARK,USEMARK,ESCMARK;

        STRING(8) XMARK;
        INTEGER XCT;
        REF(CONSTANT) XPT;
        LOGICAL SCANMARK ¢ normally FALSE ¢ );

   RECORD HYSTORY(INTEGER LOWH,HIGHH;
        REF(TRIO,IDENTIFIER) STARTH;
        REF(HYSTORY)NEXTHYS);

   RECORD EFFECTS(REF(HYSTORY) ESET,EUSE,ESCAPE);

        REF(TRIO ¢ proc denotation ¢) ARRAY PROCTABLE(1::100);
        REF(EFFECTS)ARRAY EFFECTSTABLE(1::100);
        REF(SYMBOL) PROCSYMBOLTABLE;
        REF(EFFECTS) CURRENTEFFECTS;

   INTEGER LARGESTIDENTIFIERCODE, NIDENTS32, NPROCS, NSTK;
   EXP T,F;

   LOGICAL BG; ¢ boolean garbage pail ¢
   REF(MSTATE,MSTATES) MG; ¢ mstate garbage pail ¢
   INTEGER INDENTATION; ¢ for the print file ¢
```

<u>TIMING</u>

```
PROCEDURE TIME(INTEGER KEY, PR;
               INTEGER ARRAY RES(*));
   FORTRAN "TIME";

INTEGER ARRAY TIMEARRAY(1::5);

PROCEDURE STARTTIMER; TIME(0,0,TIMEARRAY);

REAL PROCEDURE CPUTIME;
   COMMENT TIME IN SECONDS;
   BEGIN
   TIME(9,0,TIMEARRAY);
   ( TIMEARRAY(1)  + TIMEARRAY(2) )  / (256*300)
   END;
```

OUTPUT

```
PROC NEWLINE; ¢ begin a new line with the proper indentation ¢
   BEGIN
      IOCONTROL(2);
      WRITEON(INDENTATION);
      FOR J:=1 UNTIL INDENTATION DO WRITEON("   ")
      END;

PROC PRINT(REF(TRIO,IDENTIFIER,CONSTANT,DECL,
      MSTATE,MSTATES,SYMBOL,HYSTORY,EFFECTS) VALUE X);
  ¢ Print X. For EXPressions, use DISPLAY instead ¢
   IF X = NULL THEN SKIP
   ELSE IF X IS TRIO THEN BEGIN
      WRITEON("TRIO(",OP(X),",");
      PRINT(FIRST(X));WRITEON(",");
      PRINT(SECOND(X));WRITEON(",");
      PRINT(THIRD(X));WRITEON(")") END
   ELSE IF X IS IDENTIFIER THEN WRITEON(IDNAME(X))
   ELSE IF X IS DECL THEN BEGIN
      WRITEON("LET ",IDNAME(FORMAL(X)),"=");
      PRINT(ACTUAL(X)) END
   ELSE IF  X IS CONSTANT THEN BEGIN
      IF TYPE(X)="BOOL" THEN BEGIN
         IF VALUEKNOWN(X) THEN BEGIN
            IF VALUEB(X) THEN WRITEON("TRUE ")
               ELSE WRITEON("FALSE ")
            END
            ELSE WRITEON("UNKNOWN BOOLEAN ")
         END

      ELSE IF TYPE(X)="REF " THEN BEGIN
         IF ISINDIVIDUAL(X)
            THEN WRITEON("INDIVIDUAL ")
            ELSE WRITEON("NONINDIVIDUAL ");
         WRITEON("REFERENCE ");
         WRITEON(X);
         IF SCANMARK(X)=TRUE THEN WRITEON("ALREADY MENTIONED
                                             ")
            ELSE BEGIN
            PRINT(HISTORY(X));
            IF ISINDIVIDUAL(X) AND VALUEKNOWN(X) THEN BEGIN
               NEWLINE;
               SCANMARK(X):=TRUE;
               WRITEON("THIS REFERENCE REFERS TO ");
               PRINT(VALUER(X));
               SCANMARK(X):=FALSE;
               END
            END
         END

      ELSE IF TYPE(X)="NULL" THEN WRITE("NULL")
```

<u>OUTPUT</u>

```
         ELSE IF TYPE(X)="PROC" THEN BEGIN
            WRITEON("PROC ");
            PRINT(CEFFECTS(X))
            END
         ELSE IF TYPE(X)="     " THEN BEGIN
            WRITEON("UNKNOWN ");
            PRINT(HISTORY(X))
            END
         ELSE WRITEON(" INVALID TYPE -- """,TYPE(X),""" ")
         END

      ELSE IF X IS MSTATE THEN
         BEGIN NEWLINE;
         WRITEON("MSTATE ");
         INDENTATION:=INDENTATION+1;
         IF LABEL(X)  ¬= " " THEN BEGIN
            WRITEON("JUMPING TO ", LABEL(X))
            END;
         NEWLINE;
         PRINT(SYMBOLTABLE(X));
         NEWLINE;
         WRITEON("STACK : ");
         INDENTATION:=INDENTATION+1;
         FOR REF(TRIO):I IN LIST MSTACK(X) LINK SECOND DO OPEN
            NEWLINE;
            PRINT(FIRST(I))
            CLOSE;
         INDENTATION:=INDENTATION-2 END

      ELSE IF X IS MSTATES THEN BEGIN
         NEWLINE;
         WRITEON("STATE MODEL "); INDENTATION:=INDENTATION+1;
         FOR REF(MSTATES): I IN LIST X LINK NEXTMSTATE DO
            PRINT(THISMSTATE(I));
         INDENTATION:=INDENTATION-1;
         END

      ELSE IF X IS SYMBOL THEN ¢ whole symbol table ¢ BEGIN
         NEWLINE;
         WRITEON("ENVIRONMENT ");
         INDENT(
         FOR REF(SYMBOL): T IN LIST X LINK NEXTSYMBOL DO OPEN
            NEWLINE;
            PRINTSYMBOL(T)
            CLOSE )
         END

      ELSE IF X IS HYSTORY THEN
         BEGIN NEWLINE; WRITEON("HISTORY ");
         INDENTATION:=INDENTATION+1;
         INTFIELDSIZE := 3;
         FOR REF(HYSTORY): I IN LIST X LINK NEXTHYS DO BEGIN
```

OUTPUT

```
          NEWLINE;
          WRITEON("FROM ",LOWH(I),"TO",HIGHH(I),
                   " VALS, STARTING AT ");
          PRINT(STARTH(I));
          END;
       INTFIELDSIZE := 14;
       INDENTATION:=INDENTATION-1
       END

    ELSE IF X IS EFFECTS THEN BEGIN
       NEWLINE;
       WRITEON("EFFECTS -- ");
       INDENTATION:=INDENTATION+1;
       NEWLINE;
       WRITEON("SET -- ");
       PRINT(ESET(X));
       NEWLINE;
       WRITEON("ESCAPE -- ");
       PRINT(ESCAPE(X));
       INDENTATION:=INDENTATION-1
       END;

 PROC PRINTSYMBOL(REF(SYMBOL) VALUE S);
    BEGIN PRINT8(SNAME(S));
       WRITEON("= ");
       INDENT(PRINT(POSSESSION(S)))
       END;

 PROC PACKETY(INTEGER VALUE OUTPRIO, INPRIO;
              INTEGER RESULT NEWPRIO;
              PROC X);
   ¢ Enclose X in parentheses, if necessary ¢
    IF OUTPRIO > INPRIO THEN BEGIN
          WRITEON(" ( ");
          NEWPRIO:=0;
          INDENT(X);
          WRITEON(") ")
          END
       ELSE BEGIN
          NEWPRIO := INPRIO;
          X
          END;

 PROC INDENT(PROC X);
   ¢ indent X ¢
    BEGIN
       INDENTATION := INDENTATION + 1;
       X;
       INDENTATION := INDENTATION - 1
    END;

 PROC DISPLAY(INTEGER VALUE PRIO;
```

OUTPUT

```
                EXP VALUE X);
      ¢ print X in readable form, as a priority
      ¢ PRIO expression ¢
       BEGIN INTEGER NEWP;
          REFERENCE (TRIO) I;
          IF X IS TRIO THEN BEGIN
             IF OP(X) = "GE" THEN WRITEON("GEN ")
             ELSE IF (OP(X) = "VL") OR (OP(X) = "GO") THEN
                PACKETY(PRIO, 40, NEWP, BEGIN
                   IF OP(X) = "VL" THEN WRITEON("VAL ")
                      ELSE WRITEON("GO ");
                   INDENT(DISPLAY(NEWP, FIRST(X)))
                   END)
             ELSE IF OP(X) = "," THEN PACKETY(PRIO, 25, NEWP,
                                       BEGIN
                   FOR I IN LIST X LINK SECOND DO OPEN
                      DISPLAY(25,FIRST(I));
                      IF SECOND(I) ¬= NULL THEN WRITEON(", ")
                      CLOSE
                   END)
             ELSE IF (OP(X) = "; ") OR (OP(X) = ": ") THEN
                PACKETY(PRIO, 20, NEWP, BEGIN
                   FOR I IN LIST X LINK SECOND DO OPEN
                      DISPLAY(20, FIRST(I));
                      IF (SECOND(I) ¬= NULL) OR (OP(I) = ": ")
                                       THEN BEGIN
                         WRITEON(OP(I));
                         IF OP(I) ¬= ": " THEN NEWLINE
                         END
                      CLOSE
                END)
             ELSE IF (OP(X) = ":=") THEN PACKETY(PRIO, 30, NEWP,
                                       BEGIN
                   DISPLAY(NEWP+1, FIRST(X));
                   WRITEON(":= ");
                   DISPLAY(NEWP, SECOND(X))
                   END)
             ELSE IF OP(X) = "PR" THEN PACKETY(PRIO,20,NEWP, BEGIN
                WRITEON("( PROC ( ");
                FOR I IN LIST FIRST(X) LINK SECOND DO OPEN
                   DISPLAY(25,FORMAL(FIRST(I)));
                   IF SECOND(I) ¬= NULL THEN WRITEON(", ")
                   CLOSE;
                WRITEON(") : ");
                INDENT(DISPLAY(20, SECOND(X)));
                WRITEON(") ")
                END)
             ELSE IF OP(X) = "CA" THEN PACKETY(PRIO, 100, NEWP,
                                       BEGIN
                DISPLAY(NEWP + 1, FIRST(X));
                WRITEON("( ");
                FOR I IN LIST SECOND(X) LINK SECOND DO OPEN
```

<u>OUTPUT</u>

```
                    DISPLAY(25, FIRST(I));
                    IF SECOND(I) ¬= NULL THEN WRITEON(", ")
                    CLOSE;
                 WRITEON(") ")
                 END)
             ELSE IF OP(X) = "WD" THEN BEGIN
                 WRITEON("WHILE ");
                 INDENT(DISPLAY(0, FIRST(X)));
                 NEWLINE;
                 WRITEON("DO ");
                 INDENT(DISPLAY(0, SECOND(X)));
                 NEWLINE;
                 WRITEON("OD ")
                 END
             ELSE IF OP(X) = "IF" THEN BEGIN
                 WRITEON("IF ");
                 INDENT( DISPLAY(0,FIRST(X)));
                 NEWLINE;
                 WRITEON("THEN ");
                 INDENT(DISPLAY(0,SECOND(X)));
                 NEWLINE;
                 WRITEON("ELSE ");
                 INDENT(DISPLAY(0, THIRD(X)));
                 NEWLINE;
                 WRITEON("FI ")
                 END
             ELSE IF OP(X) = "DS" THEN BEGIN
                 WRITEON("DISPLAY ");
                 NEWLINE;
                 PRINT(MATTACH(X));
                 NEWLINE
                 END
             ELSE IF OP(X) = "TR" THEN WRITEON("TRACE ")
             ELSE IF OP(X) = "NT" THEN WRITEON("NOTRACE ")
             ELSE PRINT(X)
             END
         ELSE IF X IS IDENTIFIER THEN PRINT8(IDNAME(X))
         ELSE IF X IS CONSTANT THEN PRINT(X)
         ELSE ¢ X IS DECL ¢ PACKETY(PRIO, 20, NEWP, BEGIN
             WRITEON("LET ");
             DISPLAY(NEWP, FORMAL(X));
             WRITEON("= ");
             DISPLAY(NEWP, ACTUAL(X))
             END)
     END;

   PROC PRINT8(STRING(8) VALUE S);
     ¢print identifiers with few trailing blanks ¢
      IF S(2|6) = " " THEN WRITEON(S(0|3))
      ELSE IF S(4|4)=" " THEN WRITEON(S(0|5))
      ELSE IF S(6|2)=" " THEN WRITEON(S(0|7))
      ELSE WRITEON(S," ");
```

INPUT

```
EXP
PROCEDURE READEXP;
   COMMENT Read an expression ;


   BEGIN

   COMMENT The parsing stack ;

   INTEGER ARRAY SRPRIO, SLPRIO(0::NSTK);
   EXP ARRAY SEXP(0::NSTK);
   STRING(8) ARRAY SSTR(0::NSTK);
   INTEGER TOP;

   COMMENT miscellaneous variables;

   STRING(8) NEWSTRING;
   INTEGER NEWLPRIO, NEWRPRIO;
   STRING(80) INPUT; INTEGER INPUTP;
   LOGICAL ERROROCCURRED;

   PROCEDURE READATOM;

      COMMENT  READATOM reads the next atom from the input
                                  file.
                Atoms are delimited by blanks and newlines.
               internal global variables:
                INPUT is a one card buffer.
                INPUTP points to the point in INPUT next
                                    to be processed.
               external variables:
                NEWSTRING - the atom read, an 8-character
                                  string,
                   padded with blanks on the right.
                NEWLPRIO,NEWRPRIO - th left and right
                                  priorities
                   of the atom read;

      BEGIN INTEGER I,J; STRING(16) STUFF; I:=INPUTP;
         IF I>71 THEN BEGIN READCARD(INPUT); I:=0; WRITE
                                 (INPUT) END ;
         WHILE INPUT(I|1)=" " DO BEGIN
            IF I>=71 THEN BEGIN READCARD(INPUT); I:=-1;
                      WRITE(INPUT) END;
            I:=I+1
            END;
         J:=I;
         WHILE(INPUT(J|1)¬=" ") AND(J<=71) DO J:=J+1;
         INPUTP:=J;
         IF J-I>8 THEN BEGIN WRITE(" TOO LONG"); J:=I+8 END;
         STUFF(0|8):=INPUT(I|8);
```

```
            STUFF(J-I|8):="        ";
            NEWSTRING:=STUFF(0|8);
            NEWLPRIO:=LPRIO(NEWSTRING);
            NEWRPRIO:=RPRIO(NEWSTRING)
         END;

      PROCEDURE RECOGNISE(INTEGER VALUE I);

         COMMENT RECOGNISE does the semantic processing when a
                     significant expression has been placed on the
                     stack. It also expells the entries for the
                                                 parts
                     of the expression from the stack, replacing
                     them by a single entry fornthe entire
                                                 expression.
                   Input -
                     SLPRIO,SRPRIO,SEXP,SSTR - the stack.
                     I,TOP - The expression is found in entries
                        I through TOP on the stack.
                   Output -
                     SLPRIO,SRPRIO,SEXP,SSTR,TOP;

         BEGIN

         EXP PROCEDURE LISTEN(EXP VALUE EXPR);
            IF ¬(EXPR IS TRIO) THEN TRIO("  ",EXPR,NULL,NULL
                                                ,NULL)
            ELSE IF(OP(EXPR)=", ") OR (OP(EXPR)="; ") OR (OP
                                                (EXPR)=": ")
                   OR (OP(EXPR)="  ") THEN EXPR
            ELSE TRIO("  ",EXPR,NULL,NULL,NULL);

         EXP EXPR;
         EXPR:=NULL;
            IF (IF TOP-I+1<3 THEN FALSE
                       ELSE(SSTR(I+1)=",       ")
                         OR(SSTR(I+1)=".       ")
                         OR(SSTR(I+1)=":       ")
                         OR(SSTR(I+1)=";       ")      )
            THEN BEGIN EXP PEXP;
               PEXP:=TRIO("  ",SEXP(TOP),NULL,NULL,NULL);
               FOR J:=TOP-2 STEP -2 UNTIL I DO
                  PEXP:=TRIO(SSTR(J+1)(0|2),SEXP(J),PEXP,NULL
                                                ,NULL);
               EXPR:=PEXP
               END
            ELSE
          CASE TOP-I+1 OF BEGIN
            IF SSTR(I)="TRUE    " THEN EXPR:=CONSTANT(TYPE="BOOL"
                                   ,VALUEKNOWN=TRUE,VALUEB=TRUE)
            ELSE IF SSTR(I)="FALSE   "
               THEN EXPR:=CONSTANT(TYPE="BOOL",VALUEKNOWN=TRUE
                                  ,VALUEB=FALSE)
   ELSE IF  SSTR(I)="GEN     "
      THEN EXPR:=TRIO("GE",NULL,NULL,NULL,NULL)
   ELSE IF SSTR(I) = "TRACE   " THEN
      EXPR := TRIO("TR", NULL, NULL, NULL, NULL)
   ELSE IF SSTR(I) = "NOTRACE " THEN
      EXPR := TRIO("NT", NULL, NULL, NULL, NULL)
   ELSE IF SSTR(I) = "DISPLAY " THEN
      EXPR := TRIO("DS", NULL, NULL, NULL, NULL)
   ELSE EXPR:=IDENTIFIER(SSTR(I));

COMMENT two symbols ;
   IF SSTR(I)="VAL     "
      THEN EXPR:=TRIO("VL",SEXP(TOP),NULL,NULL,NULL)
   ELSE IF (SSTR(I)="GO      ") THEN
      EXPR:=TRIO("GO",SEXP(TOP),NULL,NULL,NULL)
   ELSE IF (SSTR(I)="........") AND (SSTR(TOP)=".......
                                             .")
      THEN EXPR:=TRIO("CA",SEXP(I),LISTEN(SEXP(I+1))
                                         ,NULL,NULL)
   ELSE ERROR(2);

COMMENT 3 symbols ;
   IF ( SSTR(I)="(       ") AND ( SSTR(I+2) = ")
                                             " )
      THEN EXPR:=SEXP(I+1)
   ELSE IF(SSTR(I+1)=":=      ")OR(SSTR(I+1)=":       ")
          OR (SSTR(I+1)=";       ") OR (SSTR(I+1)=",
                                              ")
      THEN EXPR:=TRIO(SSTR(I+1)(0|2),SEXP(I),SEXP(TOP),
                           NULL,NULL)
   ELSE IF (SSTR(I)="GO      ") AND (SSTR(I+1)="TO
                                             ")
      THEN EXPR:=TRIO("GO",SEXP(TOP),NULL,NULL,NULL)
   ELSE IF (SSTR(I+1)="(       ")
       AND (SSTR(I+2)=")       ")
      THEN EXPR:=TRIO("CA",SEXP(I),NULL,NULL,NULL)
   ELSE ERROR(3);

COMMENT 4 symbols ;

   IF (SSTR(I)="PROC    ") AND (SSTR(I+2)=":       ")
      THEN BEGIN
         EXP IN,OUT;
         IN:=LISTEN(SEXP(I+1));
         OUT:=IN;
         WHILE IN¬=NULL DO BEGIN
            FIRST(IN):=DECL(FIRST(IN),NULL,NULL);
            IN:=SECOND(IN)
            END;
         EXPR:=TRIO("PR",OUT,SEXP(I+3),NULL,NULL);
         NPROCS:=NPROCS+1;
```

INPUT

```
                PROCTABLE(NPROCS):=EXPR;
                EFFECTSTABLE(NPROCS):=EFFECTS(NULL,NULL,NULL)
                END
            ELSE IF  (SSTR(I)="LET      ") AND (SSTR(I+2)="=
                                                            ")
              THEN BEGIN EXPR:=DECL(SEXP(I+1),SEXP(I+3),NULL);
                IF SEXP(I+3) IS TRIO THEN
                  IF OP(SEXP(I+3))="PR" THEN
                    PROCSYMBOLTABLE:=SYMBOL(IDNAME(SEXP(I+1)
                                                  ),
                                                 SEXP(I+3),PROCSYMBOLTABLE)
                END
            ELSE IF  (SSTR(I+1)="(        ")
                AND  (SSTR(I+3)=")        ")
              THEN  EXPR:=TRIO("CA",SEXP(I),LISTEN(SEXP(I+2))
                                                  ,NULL,NULL)
            ELSE ERROR(4);

          COMMENT 5 symbols ;
            IF  (SSTR(I)="WHILE    ")  AND  (SSTR(I+2)="DO       ")
                  AND  (SSTR(I+4)="OD       ")
              THEN  EXPR:=TRIO("WD",SEXP(I+1),SEXP(I+3),NULL
                                                  ,NULL);

          COMMENT 6 SYMBOLS ;
            ERROR(6);

          COMMENT 7 SYMBOLS ;
            IF  (SSTR(I)="IF       ")  AND  (SSTR(I+2)="THEN     ")
                                                  AND
              (SSTR(I+4)="ELSE     ")  AND  (SSTR(I+6)="FI       ")
              THEN  EXPR:=TRIO("IF",SEXP(I+1),SEXP(I+3),SEXP(I+5
                                                  ),NULL)

          END;

          IF EXPR = NULL THEN ERROR(8);
          CLEARSTACK(I+1,TOP);
          SLPRIO(I):=SRPRIO(I):=
            (IF SRPRIO(I-1)>NEWLPRIO THEN SRPRIO(I-1) ELSE
                                           NEWLPRIO);
          SEXP(I):=EXPR;
          SSTR(I):=".........";
          TOP:=I;
          END;

      PROCEDURE CLEARSTACK(INTEGER VALUE I, J);
        COMMENT clear all stack entries from I to J inclusive ;
          FOR K:=I UNTIL J DO
          BEGIN
            SLPRIO(K):=-99; SRPRIO(K):=-99;
            SEXP(K):=NULL;
```

<u>INPUT</u>

```
            SSTR(K):="--------"
         END;

   PROCEDURE PRINTSTACK(INTEGER VALUE I,J);
      COMMENT print part of the stack on a single line.
              only SSTR( I through J ) will be printed;
      BEGIN WRITE  ("STACK",I);
         FOR K:=I UNTIL J DO WRITEON(" ",SSTR(K));
      END;


   PROCEDURE STACK;
      COMMENT place the new atom on the stack;
      BEGIN
         TOP:=TOP+1;
         IF TOP>NSTK THEN ERROR(10);
         SRPRIO(TOP):=NEWRPRIO;
         SLPRIO(TOP):=NEWLPRIO;
         SSTR(TOP):=NEWSTRING;
         SEXP(TOP):=NULL;
      END;

   INTEGER PROCEDURE LPRIO(STRING(8) VALUE S);
      COMMENT The left priority of the atom S ;
      IF S="WHILE   " THEN 100 ELSE
      IF S="DO      " THEN 0 ELSE
      IF S="OD      " THEN 0 ELSE
      IF S="IF      " THEN 100 ELSE
      IF S="THEN    " THEN 0 ELSE
      IF S="ELSE    " THEN 0 ELSE
      IF S="FI      " THEN 0 ELSE
      IF S="(       " THEN 100 ELSE
      IF S=")       " THEN 0 ELSE
      IF S=":=      " THEN 31 ELSE
      IF S=";       " THEN 20 ELSE
      IF S=":       " THEN 20 ELSE
      IF S=".       " THEN 20 ELSE
      IF S="GO      " THEN 41 ELSE
      IF S="VAL     " THEN 41 ELSE
      IF S="..      " THEN -100 ELSE
      IF S="PROC    " THEN 22 ELSE
      IF S=",       " THEN 25 ELSE
      IF S="LET     " THEN 22 ELSE
      IF S="=       " THEN 22 ELSE
      110;

   INTEGER PROCEDURE RPRIO(STRING(8) VALUE S);
      COMMENT The right priority of the atom S ;
      IF S="WHILE   " THEN 0 ELSE
      IF S="DO      " THEN 0 ELSE
      IF S="OD      " THEN 100 ELSE
      IF S="IF      " THEN 0 ELSE
```

INPUT .

```
   IF S="THEN     " THEN 0 ELSE
   IF S="ELSE     " THEN 0 ELSE
   IF S="FI       " THEN 101 ELSE
   IF S="(        " THEN 0 ELSE
   IF S=")        " THEN 101 ELSE
   IF S=":=       " THEN 30 ELSE
   IF S=";        " THEN 20 ELSE
   IF S=":        " THEN 20 ELSE
   IF S=".        " THEN 20 ELSE
   IF S="GO       " THEN 40 ELSE
   IF S="VAL      " THEN 40 ELSE
   IF S="..       " THEN -100 ELSE
   IF S="PROC     " THEN 20 ELSE
   IF S=",        " THEN 25 ELSE
   IF S="LET      " THEN 22 ELSE
   IF S="=        " THEN 22 ELSE
   110;

PROCEDURE ERROR(INTEGER VALUE I);
   BEGIN STRING(80)MARK;
   WRITE(" SYNTAX ERROR ");
   WRITE(" NEW ATOM: ",NEWSTRING,NEWLPRIO,NEWRPRIO);
   WRITE(" STACK ");
   FOR I := TOP STEP -1 UNTIL 0 DO
      WRITE(I,SSTR(I),SLPRIO(I),SRPRIO(I),SEXP(I));
      WRITE(INPUT);
      MARK:="
                       ";
      IF (INPUTP<80) AND (INPUTP >= 0)
         THEN MARK(INPUTP|1):="*";
      WRITE(MARK,INPUTP);
      WRITE(" FLUSH INPUT");
      WHILE INPUT(0|2)¬=".." DO BEGIN
         READCARD(INPUT); WRITE(INPUT) END;
      ERROROCCURRED:=TRUE;
      GO TO EXIT
   END;

COMMENT stack and variable initialisation.

   -99 is the undefined value for integers,
   "--------" is the undefined string;

TOP:=0;
SLPRIO(0):=-100;
SRPRIO(0):=-100;

SEXP(0):=NULL;
SSTR(0):="..      ";
CLEARSTACK(1,NSTK);
NEWSTRING:="--------";
NEWLPRIO:=NEWRPRIO:=-99;
```

<u>INPUT</u>

```
    INPUTP:=80;
    ERROROCCURRED:=FALSE;

    WHILE SSTR(2)¬="..      " DO BEGIN
       READATOM;
       IF SRPRIO(0)>=SLPRIO(1) THEN ERROR(9);
       WHILE SRPRIO(TOP) > NEWLPRIO DO BEGIN INTEGER I;
          I:=TOP;
          WHILE SRPRIO(I-1)=SLPRIO(I) DO I:=I-1;
          IF SRPRIO(I-1)>SLPRIO(I)
              THEN ERROR(1);
          RECOGNISE(I);
       END;
       STACK
       END;
    IF ERROROCCURRED THEN GO TO EXIT;
    SEXP(1)

 END;
```

MERGING

```
    REF(MSTATES) PROCEDURE CONCATENATEMSTATES(REF(MSTATES) VALUE P
                                                ,Q);
       BEGIN REF(MSTATES)T,V;
          IF P=NULL THEN V:=Q
          ELSE IF Q=NULL THEN V:=P
          ELSE BEGIN T:=P;
             WHILE NEXTMSTATE(T) ¬= NULL  DO T:=NEXTMSTATE(T);
             NEXTMSTATE(T):=Q;
             LASTMSTATE(Q):=T;
             V:=P
             END;
          V
          END;

    REF(MSTATES) PROCEDURE MERGEMSTATES(REF(MSTATES) VALUE M1, M2;
                                              LOGICAL VALUE RESULT
                                                CHANGE);
          ¢ This procedure produces an mstate which is true iff
                                                either
          ¢ argument mstate is true. It pairs off the two MSTATE
                                                lists,
          ¢ jump by jump, and merges the ones that pair off. The
          ¢ unpaired MSTATES are simply chained on. M1 and M2 are
          ¢ destroyed in the process. ¢
          ¢ CHANGE is set to TRUE if M1 is different from the
                                                returned
          ¢ value ¢

       BEGIN REF(MSTATE)I,J; I:=J:=NULL;
          M1:=COPYMSTATES(M1);
          M2:=COPYMSTATES(M2);
          FOR REF(MSTATES): K IN LIST M1 LINK NEXTMSTATE DO OPEN
              I:=THISMSTATE(K);
             FOR REF(MSTATES): L IN LIST M2 LINK NEXTMSTATE DO
                                                OPEN
                 J:=THISMSTATE(L);
                IF (LABEL(I) = LABEL(J))
          THEN BEGIN
                      EXP P,Q,S;
                      DELETE L FROM LIST M2 LL LASTMSTATE RL
                                                NEXTMSTATE;
                      ¢ Merge I and J. This requires recursively
                      ¢ scanning their data structure ¢
                      ¢ Merge the stacks ¢
                      P:=MSTACK(I);Q:=MSTACK(J);
                      WHILE(P¬=NULL) AND (Q¬=NULL) DO BEGIN
                         MERGECONSTANTS(FIRST(P),FIRST(Q),CHANGE);
                         P:=SECOND(P);
                         Q:=SECOND(Q);
                         END;
                      IF (P¬=NULL) OR (Q¬=NULL)
```

```
                              THEN ERROR "NON-MATCHING STACK LENGTHS";
                           ¢ stack is now merged ¢
                           MERGESYMBOLTABLES(SYMBOLTABLE(I),
                                             SYMBOLTABLE(J),CHANGE);
                           END
                     CLOSE CLOSE;
               IF (M2¬=NULL) THEN CHANGE:=TRUE;
               M1:=CONCATENATE MSTATES( M1,M2);
              ¢ Edit M1 to remove duplicate mvalues ¢
               FOR REF(MSTATES): K IN LIST M1 LINK NEXTMSTATE DO OPEN
                  FOR REF(SYMBOL):S IN LIST SYMBOLTABLE(THISMSTATE(K))
                           LINK NEXTSYMBOL DO
                     FOR REF(CONSTANT):C IN LIST POSSESSION(S) LINK
                                                 VALUER DO
                        VALUER(C):=XPTS(VALUER(C));
                  FOR REF(TRIO):S IN LIST MSTACK(THISMSTATE(K)) LINK
                                                 SECOND DO
                     FOR REF(CONSTANT):C IN LIST FIRST(S) LINK VALUER
                                                 DO
                        VALUER(C):=XPTS(VALUER(C))
                  CLOSE;
               M1
               END;

         REF(CONSTANT) PROC XPTS(REF(CONSTANT) VALUE C);
            BEGIN REF(CONSTANT) I;
               FOR I IN LIST C LINK XPT WHILE XPT(I)¬=NULL DO SKIP;
               I
               END;

         PROCEDURE MERGESYMBOLTABLES(REF(SYMBOL)VALUE A,B;
                                             LOGICAL VALUE RESULT
                                               CHANGE);
            ¢ Merge two symbol tables ¢
            BEGIN
               FOR REF(SYMBOL): I IN LIST A LINK NEXTSYMBOL DO
                  FOR REF(SYMBOL): J IN LIST B LINK NEXTSYMBOL DO OPEN
                     IF SNAME(I)=SNAME(J) THEN
                        MERGECONSTANTS(POSSESSION(I),
                                             POSSESSION(J),CHANGE)
                     CLOSE;
               END;

         PROC MERGECONSTANTS(REF(CONSTANT) VALUE C,D;
                              LOGICAL VALUE RESULT CHANGE);
            BEGIN
               C:=XPTS(C);
               IF (C=NULL) OR (D=NULL) THEN ERROR " NULL CONSTANT "
               ELSE IF XPTS(D)=C THEN SKIP
               ELSE IF XPT(D)¬=NULL ¢ D has previously been matched to
                          ¢ something other than C ¢ THEN BEGIN
                  CHANGE:=TRUE;
```

```
          MERGECONSTANTS(C,XPTS(D),CHANGE)
          END
       ELSE BEGIN HISTORY(C):=UNITEHYS(HISTORY(C),
                    HISTORY(D), CHANGE);
          XCT(C):=XCT(C)+XCT(D);
          IF TYPE(C)¬=TYPE(D) THEN BEGIN
             CHANGE:=TRUE;
             IF TYPE(C)="REF " THEN SETFREE(C);
             IF TYPE(D)="REF " THEN SETFREE(D);
             TYPE(C):="    ";
             VALUER(C):=NULL ¢ TO PERMIT STORAGE TO BE FREED ¢
             END
          ELSE IF TYPE(C)="    " THEN SKIP
          ELSE IF TYPE(C)="NULL" THEN SKIP
          ELSE IF TYPE(C)="BOOL" THEN BEGIN
             IF (VALUEKNOWN(C) = VALUEKNOWN(D))
                AND (VALUEB(C)=VALUEB(D)) THEN SKIP
             ELSE IF ¬VALUEKNOWN(C) THEN SKIP
             ELSE IF VALUEKNOWN(D) AND (VALUEB(C)=VALUEB(D))
                                    THEN SKIP
             ELSE BEGIN CHANGE:=TRUE;
                        VALUEB(C):=FALSE;
                        VALUEKNOWN(C):=FALSE
                  END
             END
          ELSE IF TYPE(C)="PROC" THEN BEGIN REF(EFFECTS) CE,DE;
             CE:=CEFFECTS(C); DE:=CEFFECTS(D);
             ESET(CE):=UNITEHYS(ESET(CE),ESET(DE),CHANGE);
             EUSE(CE):=UNITEHYS(EUSE(CE),EUSE(DE),CHANGE);
             ESCAPE(CE):=UNITEHYS(ESCAPE(CE),ESCAPE(DE),CHANGE)
             END
          ELSE IF TYPE(C)="REF " THEN BEGIN
             IF ISINDIVIDUAL(C) THEN BEGIN
                IF ISINDIVIDUAL(D) THEN BEGIN
                   IF XCT(C) > 2 THEN BEGIN
                      SETFREE(C);
                      CHANGE := TRUE
                      END
                   ELSE IF VALUEKNOWN(C) AND VALUEKNOWN(D)
                      THEN MERGECONSTANTS(VALUER(C),VALUER(D)
                                          ,CHANGE)
                   ELSE IF VALUEKNOWN(C) THEN BEGIN
                      LOGEFFECT(3,HISTORY(VALUER(C)));
                      CHANGE:=TRUE;
                      VALUEKNOWN(C):=FALSE;
                      VALUER(C):=NULL
                      END
                   ELSE IF VALUEKNOWN(D)
                      THEN LOGEFFECT(3,HISTORY(VALUER(D)))
                   ELSE SKIP
                   END
                ELSE BEGIN
```

```
                     LOGEFFECT(3,HISTORY(C));
                     ISINDIVIDUAL(C):=FALSE;
                     VALUEKNOWN(C):=FALSE;
                     VALUER(C):=NULL
                     END
                  END
               ELSE ¢ ¬ ISINDIVIDUAL(C) ¢ BEGIN
                    IF ISINDIVIDUAL(D) THEN LOGEFFECT(3,HISTORY
                                                  (D))
                      ELSE SKIP END
               END
            ELSE ERROR "INVALID TYPE"
         END
         END;

   PROC SETFREE(REF(CONSTANT) VALUE C);
         ¢ RECORD THAT C HAS ESCAPED. THIS HAS CONSEQUENCES FOR
         ¢ VALUER(C) IF C IS A REFERENCE ¢
   IF C¬=NULL
      THEN BEGIN LOGEFFECT(3,HISTORY(C));
         IF TYPE(C)="REF "
         THEN IF ISINDIVIDUAL(C)
            THEN BEGIN
               REF(CONSTANT) CR;
               CR:=VALUER(C);
               ISINDIVIDUAL(C):=FALSE;
               VALUER (C):=NULL;
               VALUEKNOWN(C):=FALSE;
               SETFREE(CR)
               END;
         END;
```

COPY MSTATES

```
   REF(MSTATES) PROC COPYMSTATES(REF(MSTATES)VALUE M1);
      BEGIN REF(MSTATES) VV,OVV,V;
         REF(MSTATES) I;
         OVV:=VV:=NULL; I:=NULL;
         FOR I IN LIST M1 LINK NEXTMSTATE DO OPEN
            V:=MSTATES(COPYMSTATE(THISMSTATE(I)),NULL,OVV);
            IF OVV¬=NULL THEN NEXTMSTATE(OVV):=V;
            IF VV = NULL THEN VV:=V;
            OVV:=V;
            CLOSE;
         VV
         END;

REF(MSTATE) PROC COPYMSTATE(REF(MSTATE) VALUE SM);
   BEGIN
      REF(MSTATE)M; REF(SYMBOL)SYM; REF(TRIO) MST;
      M:=MSTATE(LABEL(SM),
             COPYSYMBOLTABLE(SYMBOLTABLE(SM)),
             COPYMSTACK(MSTACK(SM)) );
      FOR SYM IN LIST SYMBOLTABLE(SM) LINK NEXTSYMBOL DO
         CLEARXPT(POSSESSION(SYM));
      FOR MST IN LIST MSTACK(SM) LINK SECOND DO
         CLEARXPT(FIRST(MST));
      M
   END;

   REF(SYMBOL) PROC COPYSYMBOLTABLE(REF(SYMBOL) VALUE S);
      IF S = NULL THEN NULL ELSE
      SYMBOL(SNAME(S),COPYCONSTANT(POSSESSION(S)),
         COPYSYMBOLTABLE(NEXTSYMBOL(S)) );

   REF(CONSTANT) PROC COPYCONSTANT(REF(CONSTANT) VALUE C);
      BEGIN REF(CONSTANT) V;
         IF C = NULL THEN V:=NULL ELSE
         IF XPT(C)¬=NULL THEN V:=XPT(C)
            ELSE BEGIN V:=XPT(C):=<<CONSTANT>>(
                     TYPE(C),
                     VALUEKNOWN(C),
                     VALUEB(C),
                     NULL ¢ valuer ¢ ,
                     CEFFECTS(C),
                     ISINDIVIDUAL(C),
                     IDENTITY(C),
                     NULL ¢ history ¢,
                     SETMARK(C),
                     USEMARK(C),
                     ESCMARK(C),
                     XMARK(C),
                     1 ¢ XCT ¢,
                     NULL,
                     SCANMARK(C));
```

<u>COPY MSTATES</u>

```
                VALUER(V):=COPYCONSTANT(VALUER(C));
                HISTORY(V):=COPYHYSTORY(HISTORY(C))
                END;
          V
          END;

   REF(HYSTORY) PROC COPYHYSTORY(REF(HYSTORY) VALUE H);
      IF H = NULL THEN NULL ELSE
      HYSTORY(LOWH(H),HIGHH(H),STARTH(H),COPYHYSTORY(NEXTHYS(H))
                                          );

   REF(TRIO) PROC COPYMSTACK(REF(TRIO) VALUE S);
      IF S = NULL THEN NULL ELSE
         TRIO(OP(S),COPYCONSTANT(FIRST(S)),COPYMSTACK(SECOND(S)),
                                         NULL,NULL);

   PROC CLEARXPT(REF(CONSTANT) VALUE C);
      IF C=NULL THEN SKIP ELSE
      IF XPT(C)=NULL THEN SKIP
         ELSE BEGIN XPT(C):=NULL;
                    CLEARXPT(VALUER(C))
             END;
```

MSTATE SELECTORS

```
   REF(MSTATES) PROC JUMPS(REF(MSTATES) VALUE P);
      BEGIN REF(MSTATES) V,T; V:=NULL;
         FOR T IN LIST P LINK NEXTMSTATE DO OPEN
            IF ISJUMP(THISMSTATE(T))
               THEN BEGIN REF(MSTATES) Q;
                  Q:=MSTATES(THISMSTATE(T),NULL,V);
                  IF V ¬= NULL THEN LASTMSTATE(V):=Q;
                  V:=Q
                  END
            CLOSE;
         V
         END;

   LOGICAL PROC ISJUMP(REF(MSTATE) VALUE P);
      LABEL(P) ¬= "         ";

   REF(MSTATES) PROC UNJUMPS(REF(MSTATES) VALUE P);
      BEGIN REF(MSTATES) V,T; V:=NULL;
         FOR T IN LIST P LINK NEXTMSTATE DO OPEN
            IF ¬ISJUMP(THISMSTATE(T))
               THEN BEGIN REF(MSTATES) Q;
                  Q:=MSTATES(THISMSTATE(T),NULL,V);
                  IF V ¬= NULL THEN LASTMSTATE(V):=Q;
                  V:=Q
                  END
            CLOSE;
         V
         END;

   REF(MSTATES) PROC TRUES(REF(MSTATES) VALUE P);
      BEGIN REF(MSTATES) V,T; V:=NULL;
         FOR T IN LIST P LINK NEXTMSTATE DO OPEN
            IF MIGHTBETRUE(THISMSTATE(T))
               THEN BEGIN REF(MSTATES) Q;
                  Q:=MSTATES(THISMSTATE(T),NULL,V);
                  IF V ¬= NULL THEN LASTMSTATE(V):=Q;
                  V:=Q
                  END
            CLOSE;
         V
         END;

   REF(MSTATES) PROC FALSES(REF(MSTATES) VALUE P);
      BEGIN REF(MSTATES) V,T; V:=NULL;
         FOR T IN LIST P LINK NEXTMSTATE DO OPEN
            IF MIGHTBEFALSE(THISMSTATE(T))
               THEN BEGIN REF(MSTATES) Q;
                  Q:=MSTATES(THISMSTATE(T),NULL,V);
                  IF V ¬= NULL THEN LASTMSTATE(V):=Q;
                  V:=Q
                  END
            CLOSE;
         V
         END;

   LOGICAL PROC MIGHTBEFALSE(REF(MSTATE) VALUE S);
      IF ISJUMP(S) THEN FALSE
         ELSE IF MSTACK(S)=NULL THEN FALSE
         ELSE IF ¬ ( FIRST(MSTACK(S)) IS CONSTANT ) THEN FALSE
         ELSE IF TYPE(FIRST(MSTACK(S)))="     " THEN TRUE
         ELSE IF TYPE(FIRST(MSTACK(S)))¬="BOOL" THEN FALSE
         ELSE (¬VALUEKNOWN(FIRST(MSTACK(S))))
              OR ¬VALUEB(FIRST(MSTACK(S)));

   LOGICAL PROC MIGHTBETRUE(REF(MSTATE) VALUE S);
      IF ISJUMP(S) THEN FALSE
         ELSE IF MSTACK(S)=NULL THEN FALSE
         ELSE IF ¬ ( FIRST(MSTACK(S)) IS CONSTANT ) THEN FALSE
         ELSE IF TYPE(FIRST(MSTACK(S)))="     " THEN TRUE
         ELSE IF TYPE(FIRST(MSTACK(S)))¬="BOOL" THEN FALSE
         ELSE (¬VALUEKNOWN(FIRST(MSTACK(S))))
              OR VALUEB(FIRST(MSTACK(S)));
```

ANALYSIS

```
PROCEDURE MANALYSEPROCDENOTATIONS;
   BEGIN
      CHANGEEFFECTS:=TRUE;
      WHILE CHANGEEFFECTS DO BEGIN
         CHANGEEFFECTS:=FALSE;
         FOR I := 1 UNTIL NPROCS DO BEGIN
            REF(MSTATES) INIT;
            EXP P;
            REF(TRIO) J;
            REF(SYMBOL) ARGS;
            INTEGER K;
            ARGS:=NULL;
            P:=PROCTABLE(I);
            CURRENTEFFECTS:=EFFECTSTABLE(I);
            INIT:=COPYMSTATES(SUPERINIT);
            K:=1;
            FOR J IN LIST FIRST(P) ¢ formal parameters ¢ LINK
                                          SECOND
               DO OPEN
               DEFINESYMBOL(FORMAL(FIRST(J)),
                  CONSTANT(TYPE="     ", HISTORY=STARTHYS(
                     IDENTIFIER(PARAMNAME(K)))),ARGS);
               K := K+1
               CLOSE;
            SYMBOLTABLE(THISMSTATE(INIT)):=ARGS;
            MG:=MANALYSE(INIT,SECOND(P))
            END;
         CURRENTEFFECTS:=EFFECTS(NULL,NULL,NULL)
         END
      END;

STRING(8) PROCEDURE PARAMNAME(INTEGER VALUE I);
   BEGIN STRING(8) N; STRING(12) S;
   N:="P       ";
   S:=INTBASE10(I);
   N(2|6):=S(6|6);
   N
   END;

REF(MSTATES) PROC MANALYSE(REF(MSTATES) VALUE INIT; EXP VALUE
                                          EXPR);
   BEGIN REF(MSTATES) EXITMSTATE; EXITMSTATE:=NULL;
   IF TRACING THEN BEGIN
      WRITE("MANALYSE -- ");
      DISPLAY(100,EXPR);
      PRINT(INIT)
      END;
   IF EXPR = NULL THEN ERROR "MANALYSE NULL EXPR."
   ELSE IF EXPR IS TRIO THEN BEGIN
      IF OP(EXPR) = "GE" THEN BEGIN
         FOR REF(MSTATES):I IN LIST INIT LINK NEXTMSTATE DO
```

<u>ANALYSIS</u>

```
            ENSTACK(THISMSTATE(I),CONSTANT(TYPE="REF ",INDIV)
                                            );
         EXITMSTATE:=INIT
         END
      ELSE IF (OP(EXPR) = "DS") OR (OP(EXPR) = "TR")
            OR (OP(EXPR) = "NT") THEN BEGIN
         IF OP(EXPR) = "DS"
            THEN MATTACH(EXPR) := COPYMSTATES(INIT)
         ELSE IF OP(EXPR) = "TR" THEN TRACING := TRUE
         ELSE IF OP(EXPR) = "NT" THEN TRACING := FALSE;
         FOR REF(MSTATES): I IN LIST INIT LINK NEXTMSTATE DO
                                              OPEN
            ENSTACK(THISMSTATE(I), VOIDVALUE)
            CLOSE;
         EXITMSTATE := INIT
         END
      ELSE IF (OP(EXPR) = "  ") OR (OP(EXPR) = ", ") THEN
                                              BEGIN
         ¢ actual parameter list ¢
         REFERENCE(MSTATES) M;
         INTEGER I; I:=0; EXITMSTATE:=NULL;
         FOR REF(TRIO): T IN LIST EXPR LINK SECOND DO OPEN
            I:=I+1;
            INIT := MANALYSE(INIT, FIRST(T));
            M := JUMPS(INIT);
            POP(I-1,M);
            EXITMSTATE:=MERGEMSTATES(EXITMSTATE, M, BG);
            INIT:=UNJUMPS(INIT)
            CLOSE;
         EXITMSTATE := MERGEMSTATES(EXITMSTATE,INIT, BG)
         END
      ELSE IF OP(EXPR) = "VL" THEN BEGIN
         INIT:=MANALYSE(INIT,FIRST(EXPR));
         FOR REF(MSTATES):I IN LIST INIT LINK NEXTMSTATE DO
                                              BEGIN
            REF(MSTATE) STATE;
            REF(CONSTANT) REFER,RVALUE;
            EXP STACK;
            STATE:=THISMSTATE(I);
            STACK:=MSTACK(STATE);
            IF STACK = NULL THEN ERROR "VAL WITH NULL STACK";
            REFER:=FIRST(STACK);
            IF REFER = NULL THEN ERROR"DEREFERENCE NULL";
            IF (TYPE(REFER)¬="REF ")AND (TYPE(REFER)¬="    ")
                                              THEN
               ABORT "INVALID ARGUMENT FOR VAL";
            IF ISINDIVIDUAL(REFER) AND VALUEKNOWN(REFER)
               THEN RVALUE:=VALUER(REFER)
               ELSE RVALUE:=CONSTANT(TYPE="    ",NONINDIV,
                      HISTORY=OPHYS("VL",HISTORY(REFER)));
            POP(1,STATE);
            ENSTACK(STATE,RVALUE);
            END;
         EXITMSTATE:=INIT
         END
      ELSE IF OP(EXPR)="WD" THEN BEGIN
         REF(MSTATES) CONDINIT,
                      LOOPJUMPS,
                      CONDJUMPS,
                     BODYJUMPS,
                      BODYINIT,
                      CONDEXIT,
                      BODYUNJUMPS,
                      BODYEXIT;
         LOGICAL CHANGE,GARBAGE;
         CONDINIT:=MERGEMSTATES(INIT,MATTACH(EXPR), CHANGE);
         LOOPJUMPS:=BODYINIT:=CONDEXIT:=BODYEXIT:=NULL;
         CHANGE:=TRUE;
         WHILE CHANGE DO BEGIN
            CHANGE:=FALSE;
            CONDEXIT:=MANALYSE(COPYMSTATES(CONDINIT),FIRST
                                         (EXPR));
            CONDJUMPS:=JUMPS(CONDEXIT);
            LOOPJUMPS:=MERGEMSTATES(LOOPJUMPS,CONDJUMPS
                                         ,GARBAGE);
            BODYINIT:=UNJUMPS(CONDEXIT);
            CONDEXIT:=COPYMSTATES(FALSES(BODYINIT));
            BODYINIT:=COPYMSTATES(TRUES(BODYINIT));
            POP(1,BODYINIT);
            BODYEXIT:=MANALYSE(BODYINIT,SECOND(EXPR));
            BODYJUMPS:=COPYMSTATES(JUMPS(BODYEXIT));
            LOOPJUMPS:=MERGEMSTATES(LOOPJUMPS,BODYJUMPS
                                         ,GARBAGE);
            BODYUNJUMPS:=UNJUMPS(BODYEXIT);
            POP(1,BODYUNJUMPS);
            CONDINIT:=MERGEMSTATES(CONDINIT,BODYUNJUMPS
                                         ,CHANGE)
            END;
         MATTACH(EXPR):=CONDINIT;
         EXITMSTATE:=MERGEMSTATES(LOOPJUMPS,CONDEXIT,GARBAGE)
         END
      ELSE IF OP(EXPR)="IF" THEN BEGIN
         REF(MSTATES) IFINIT, IFEXIT,THENINIT,THENEXIT,
                      ELSEINIT,ELSEEXIT,IFJUMPS;
         LOGICAL GARBAGE;
         REF(HYSTORY)HYS;
         REF(SYMBOL) CALLSYM;
         IFINIT:=INIT;
         IFEXIT:=THENEXIT:=THENINIT:=THENEXIT:=ELSEINIT:
                                         =ELSEEXIT:=
                                        IFJUMPS:=NULL;
         IFEXIT:=MANALYSE(IFINIT,FIRST(EXPR));
         IFJUMPS:=JUMPS(IFEXIT);
         THENINIT:=COPYMSTATES(TRUES(IFEXIT));
```

ANALYSIS

```
         POP(1,THENINIT);
         ELSEINIT:=COPYMSTATES(FALSES(IFEXIT));
         POP(1,ELSEINIT);
         THENEXIT:=MANALYSE(THENINIT,SECOND(EXPR));
         ELSEEXIT:=MANALYSE(ELSEINIT,THIRD(EXPR));
         EXITMSTATE:=
            MERGEMSTATES(IFJUMPS,
               MERGEMSTATES(THENEXIT,ELSEEXIT,GARBAGE)
                                           ,GARBAGE);
         END
      ELSE IF OP(EXPR)="PR" ¢ procedure denotation ¢ THEN
                                             BEGIN
         FOR REF(MSTATES):M IN LIST INIT LINK NEXTMSTATE DO
                                             OPEN
            ENSTACK(THISMSTATE(M),CONSTANT(TYPE="PROC"
                  ,CEFFECTS=PROCEFFECTS(EXPR)))
            CLOSE;
         EXITMSTATE:=INIT
         END
      ELSE IF OP(EXPR) = "CA" ¢ a call ¢ THEN
         MANALYSECALL(INIT,EXPR,EXITMSTATE)
      ELSE IF OP(EXPR) = "GO" THEN BEGIN
         FOR REF(MSTATES): I IN LIST INIT LINK NEXTMSTATE DO
                                             BEGIN
            LABEL(THISMSTATE(I)):=IDNAME(FIRST(EXPR));
            ENSTACK(THISMSTATE(I), VOIDVALUE)
            END;
         EXITMSTATE:=INIT
         END
      ELSE IF OP(EXPR)=":="
         THEN MANALYSEASSIGNMENT(INIT,EXPR,EXITMSTATE)
      ELSE IF (OP(EXPR) = "; ")
            OR (OP(EXPR) = ": ")
      THEN BEGIN
         REF(MSTATES) M;
         REF(TRIO) P, R;
         REF(MSTATES) Q, TMSTATE;
         LOGICAL CHANGE;
         REF(MSTATE) JMP;
         REF(SYMBOL) SYM;
         REF(TRIO) SAVELOCALS;
         SAVELOCALS:=LOCALS;
         FOR M IN LIST INIT LINK NEXTMSTATE DO OPEN
            SYM:=SYMBOLTABLE(THISMSTATE(M));
            FOR P IN LIST EXPR LINK SECOND DO OPEN
               IF FIRST(P) IS DECL THEN BEGIN
                  SYM:=SYMBOL(IDNAME(FORMAL(FIRST(P))),NULL
                                            ,SYM);
                  LOCALS:=TRIO("  ",FORMAL(FIRST(P)),
                               LOCALS,NULL,NULL)
                  END;
               CLOSE;
```

ANALYSIS

```
            SYMBOLTABLE(THISMSTATE(M)):=SYM
            CLOSE;
         CHANGE:=TRUE;
         EXITMSTATE := NULL;
         WHILE CHANGE DO BEGIN
            TMSTATE:=INIT;
            CHANGE:=FALSE;
            FOR P IN LIST EXPR LINK SECOND DO OPEN
              IF OP(P)=": " THEN BEGIN
                MATTACH(P):=MERGEMSTATES(MATTACH(P),
                   TMSTATE,CHANGE);
                TMSTATE:=COPYMSTATES(MATTACH(P)) END
              ELSE BEGIN
               TMSTATE:=MANALYSE(TMSTATE,FIRST(P));
               FOR Q IN LIST JUMPS(TMSTATE) LINK NEXTMSTATE
                                       DO OPEN
                  FOR R IN LIST EXPR LINK SECOND DO OPEN
                     IF OP(R)=": " THEN
                        IF LABEL(THISMSTATE(Q))=IDNAME(FIRST
                                             (R))
                           THEN BEGIN
                              JMP:=COPYMSTATE(THISMSTATE(Q));
                              POP(1,JMP);
                              LABEL(JMP):="         ";
                              MATTACH(R):=
                                 MERGEMSTATES(MATTACH(R),
                                  MSTATES(JMP,NULL,
                                   NULL),CHANGE);
                              GO TO FOUND
                              END
                     CLOSE;
                  EXITMSTATE:=MERGEMSTATES(EXITMSTATE,
                     MSTATES(THISMSTATE(Q),NULL,NULL),BG);
                 FOUND: CLOSE;
               TMSTATE:=UNJUMPS(TMSTATE);
               IF SECOND(P)¬=NULL THEN POP(1,TMSTATE)
               END
               CLOSE
            END;
        ¢ restore LOCALS ¢
         EXITMSTATE:=MERGEMSTATES(EXITMSTATE,TMSTATE,BG);
         LOCALS:=SAVELOCALS
        ¢ finished ! ¢
         END
      ELSE ERROR " INVALID OP IN TRIO -- MANALYSE "
      END
   ELSE IF EXPR IS CONSTANT ¢ boolean -- true or false ¢
      THEN BEGIN FOR REF(MSTATES): I IN LIST INIT LINK
                                    NEXTMSTATE DO
         ENSTACK(THISMSTATE(I),COPYBASE(EXPR));
         EXITMSTATE:=INIT
         END
      ELSE IF EXPR IS IDENTIFIER
         THEN BEGIN
            REF(CONSTANT, TRIO) C;
            REF(HYSTORY) H;
            REF(TRIO) LOC;
            FOR REF(MSTATES):I IN LIST INIT LINK NEXTMSTATE
               DO OPEN
                C:=LOOKUPSYMBOL(EXPR,SYMBOLTABLE(THISMSTATE(I)));
                IF C=NULL THEN BEGIN
                   C:=LOOKUPSYMBOL(EXPR,PROCSYMBOLTABLE);
                   IF C¬=NULL THEN C:=CONSTANT(TYPE="PROC"
                      ,CEFFECTS=PROCEFFECTS(C))
                   END;
                H:=STARTHYS(EXPR);
               ¢check if no history is to be recorded ¢
                LOC:=LOCALS;
                WHILE LOC¬=NULL DO BEGIN
                   IF IDNAME(FIRST(LOC)) = IDNAME(EXPR)
                      THEN H:=NULL;
                   LOC:=SECOND(LOC)
                   END;
                IF C=NULL THEN C:=CONSTANT(TYPE="     ",
                     HISTORY=H);
                ENSTACK(THISMSTATE(I),C)
                CLOSE;
            EXITMSTATE:=INIT
            END
      ELSE IF EXPR IS DECL THEN BEGIN
         REF(MSTATES) AACTUAL;
         AACTUAL:=MANALYSE(INIT,ACTUAL(EXPR));
         FOR REF(MSTATES):I IN LIST AACTUAL LINK NEXTMSTATE DO
            DEFINESYMBOL(FORMAL(EXPR),FIRST(MSTACK(THISMSTATE(I)
                                             )),
                   SYMBOLTABLE(THISMSTATE(I)));
         EXITMSTATE:=AACTUAL
         END
      ELSE ERROR "INVALID EXP -- MANALYSE";
      IF TRACING THEN BEGIN
         WRITE("EXIT MANALYSE -- ");
         DISPLAY(100,EXPR);
         PRINT(EXITMSTATE)
         END;
      EXITMSTATE
      END;

   PROC MANALYSEASSIGNMENT(
                   REF(MSTATES) VALUE INIT;
                   EXP VALUE EXPR;
                   REF(MSTATES) RESULT EXITMSTATE);
      BEGIN
         REF(MSTATES) SOURCEINIT,
                      SOURCEEXIT,
                    DESTINIT,
                    DESTEXIT,
                    ASSJUMPS,
                    EXIT;
        LOGICAL GARBAGE;
        REF(MSTATES) I;
        SOURCEINIT:= INIT;
        SOURCEEXIT:=MANALYSE(SOURCEINIT,SECOND(EXPR));
        ASSJUMPS:=JUMPS(SOURCEEXIT);
        DESTINIT:=UNJUMPS(SOURCEEXIT);
        DESTEXIT:=MANALYSE(DESTINIT,FIRST(EXPR));
        ASSJUMPS:=MERGEMSTATES(ASSJUMPS,JUMPS(DESTEXIT),GARBAGE
                                   );
        DESTEXIT:=UNJUMPS(DESTEXIT);
        FOR I IN LIST DESTEXIT LINK NEXTMSTATE DO OPEN BEGIN
           REF(MSTATE) ST;
           EXP SO,DE;
           ST:=THISMSTATE(I);
           DE:=FIRST(MSTACK(ST));
           LOGEFFECT(1,HISTORY(DE));
           SO:=FIRST(SECOND(MSTACK(ST)));
           IF ¬ISINDIVIDUAL(DE) THEN SETFREE(SO)
              ELSE BEGIN VALUER(DE):=SO;
                 VALUEKNOWN(DE):=TRUE
                 END;
           POP(2,ST);
           ENSTACK(ST,DE);
           END CLOSE;
        EXITMSTATE:=DESTINIT ¢ as modified above ¢
        END;

    PROCEDURE MANALYSECALL(
                    REF(MSTATES) VALUE INIT;
                    REF(TRIO) VALUE EXPR;
                    REF(MSTATES) RESULT EXITMSTATE);
       BEGIN

          PROCEDURE EFFCHOOSE(REF(CONSTANT) VALUE V;
                              REF(HYSTORY) VALUE H);
                    ¢ global integer effsel ¢
             IF H ¬= NULL THEN LOGEFFECT(EFFSEL,
                                  CONCHYS(H,HISTORY(V)))
                ELSE BEGIN CASE EFFSEL OF BEGIN
                      SETMARK(V):=TRUE;
                      USEMARK(V):=TRUE;
                      ESCMARK(V):=TRUE END;
                   LOGEFFECT(EFFSEL, HISTORY(V))
                   END;
          ¢ end effchoose ¢

          REF(MSTATES) EXITPARAMS,
                       EXITJUMPS,
```

<u>ANALYSIS</u>

```
                 EXITPROC;
 EXP I;
 REF(EFFECTS) EFF;
 INTEGER NPARAMS;
 REF(TRIO) STACKINDEX;
 REF(SYMBOL) CALLSYMBOL,CS,FPARMS;
 REF(HYSTORY) HY;
 INTEGER EFFSEL;
¢ count the parameters ¢
 NPARAMS:=0;
 FOR I IN LIST SECOND(EXPR) LINK SECOND
        DO NPARAMS:=NPARAMS+1;
 EXITPARAMS:=MANALYSE(INIT,SECOND(EXPR));
 EXITMSTATE := JUMPS(EXITPARAMS);
 EXITPARAMS := UNJUMPS(EXITPARAMS);
 EXITPROC:=MANALYSE(EXITPARAMS,FIRST(EXPR));
 EXITJUMPS := JUMPS(EXITPROC);
 POP(1,EXITJUMPS);
 EXITMSTATE := MERGEMSTATES(EXITMSTATE, EXITJUMPS, BG);
 EXITPROC := UNJUMPS(EXITPROC);
 FOR REF(MSTATES):I IN LIST EXITPROC LINK NEXTMSTATE DO
                                        OPEN
    EFF:=CEFFECTS(FIRST(MSTACK(THISMSTATE(I))));
    IF EFF ¬= NULL THEN BEGIN
      ¢ Attach entries to SYMBOLTABLE identifying actual
      ¢ parameters with formal parameters. The resuting
      ¢ table will be called CALLSYMBOL. SYMBOLTABLE is
      ¢ not changed. ¢
       CALLSYMBOL:=SYMBOLTABLE(THISMSTATE(I));
       STACKINDEX:=SECOND(MSTACK(THISMSTATE(I)));
       FOR I:=NPARAMS STEP -1 UNTIL 1 DO BEGIN
          CALLSYMBOL:=SYMBOL(PARAMNAME(I),
             FIRST(STACKINDEX), CALLSYMBOL);
          STACKINDEX:=SECOND(STACKINDEX);
          END;
      ¢ callsymboltable has now been created ¢
      ¢ apply effects ¢
       FOR EFFSEL2:=1,2,3 ¢ CHOOSES SET,USE,ESCAPE ¢ DO
                                        BEGIN
          EFFSEL:=EFFSEL2;
          FOR HY IN LIST CASE EFFSEL OF
                (ESET(EFF),EUSE(EFF),ESCAPE(EFF))
             LINK NEXTHYS DO
             MAPHYS(EFFCHOOSE,HY,LOOKUPSYMBOL(STARTH(HY),
                                            CALLSYMBOL))
          END;
       FOR REF(TRIO):J IN LIST MSTACK(THISMSTATE(I))
              LINK SECOND DO
          PERFORMEFFECTS(FIRST(J));
       FOR REF(SYMBOL):J IN LIST CALLSYMBOL
              LINK NEXTSYMBOL DO
          PERFORMEFFECTS(POSSESSION(J))
          END;
       CLOSE;
    POP(NPARAMS,EXITPROC);
    EXITMSTATE:=MERGEMSTATES(EXITPROC, EXITMSTATE, BG)
    END;
```

EFFECTS HANDLING PROCEDURES

```
PROC PERFORMEFFECTS(REF(CONSTANT) VALUE C);
  ¢ Recursively scan through the constants to perform the
                                       actions
  ¢ indicated by the ESCMARKs and the SETMARKs. PERFORMEFFECTS
  ¢ uses SCANMARK to trap cycles ¢
   IF C = NULL THEN SKIP
   ELSE IF SCANMARK(C) THEN SKIP
   ELSE IF ESCMARK(C) THEN SETFREE(C)
   ELSE IF SETMARK(C) THEN BEGIN
      SETFREE(VALUER(C));
      VALUER(C):=NULL;
      VALUEKNOWN(C):=FALSE END
   ELSE BEGIN
      SCANMARK(C):=TRUE;
      PERFORMEFFECTS(VALUER(C));
      SCANMARK(C):=FALSE
      END;

REF(EFFECTS) PROC PROCEFFECTS(REF(TRIO ¢ PROCDENOTATION ¢
                                      )VALUE P);
   BEGIN INTEGER J;
      FOR I:=1 UNTIL NPROCS DO BEGIN J:=I;
         IF PROCTABLE(I)=P THEN GO TO FOUND END;
      ABORT "MISSING PROCEDURE";
     FOUND: EFFECTSTABLE(J)
      END;

PROC LOGEFFECT(INTEGER VALUE EFFSEL;
               REF(HYSTORY) VALUE H);
   CASE EFFSEL OF BEGIN
      ESET(CURRENTEFFECTS):=UNITEHYS(
         ESET(CURRENTEFFECTS),DISCARDLOCALS(H),CHANGEEFFECTS);
      EUSE(CURRENTEFFECTS):=UNITEHYS(
         EUSE(CURRENTEFFECTS),DISCARDLOCALS(H),CHANGEEFFECTS);
      ESCAPE(CURRENTEFFECTS):=UNITEHYS(
         ESCAPE(CURRENTEFFECTS),DISCARDLOCALS(H)
                                      ,CHANGEEFFECTS)
      END;
```

HISTORY HANDLING PROCEDURES

```
REF(HYSTORY) PROC DISCARDLOCALS(REF(HYSTORY) VALUE H);
   BEGIN REF(HYSTORY)I,J; REF(TRIO) L;
      I:=NULL;
      FOR J IN LIST H LINK NEXTHYS DO OPEN
         IF STARTH(J) IS IDENTIFIER THEN BEGIN
            FOR L IN LIST LOCALS LINK SECOND DO OPEN
               IF IDNAME(STARTH(J)) = IDNAME(FIRST(L))
                  THEN GO TO EX
               CLOSE;
            I := HYSTORY(LOWH(J), HIGHH(J), STARTH(J), I);
        EX: END
      CLOSE;
      I
      END;

REF(HYSTORY) PROC OPHYS(STRING(2) VALUE OPH;
                         REF(HYSTORY) VALUE HYS);
   ¢ tack the extra operation on to the hystory HYS
   BEGIN REF(HYSTORY)I,V; V:=NULL;
      IF OPH¬="VL" THEN ABORT "INVALID OPHERATOR FOR OPHHYS";
      FOR I IN LIST HYS LINK NEXTHYS DO
         V:=HYSTORY(
            (IF LOWH(I)>=10 THEN 10 ELSE LOWH(I)+1),
            (IF HIGHH(I)>=10 THEN 10 ELSE HIGHH(I)+1),
            STARTH(I),
            V);
      V
      END;

REF(HYSTORY) PROC CONCHYS(REF(HYSTORY) VALUE HYS1,HYS2);
   ¢ apply the operations of HYS1, in order, to HYS2 ¢
   BEGIN REF(HYSTORY) I,J,V; INTEGER MIN,MAX; V:=NULL;
      MIN:=100;
      MAX:=-100;
      V:=NULL;
      IF HYS1¬=NULL THEN BEGIN
         FOR I IN LIST HYS1 LINK NEXTHYS DO BEGIN
                  IF MIN>LOWH(I) THEN MIN:=LOWH(I);
                  IF MAX<HIGHH(I) THEN MAX:=HIGHH(I) END;
         FOR J IN LIST HYS2 LINK NEXTHYS DO
            V:=HYSTORY(
               (IF LOWH(J)+MIN>=10 THEN 10 ELSE LOWH(J)+MIN),
               (IF HIGHH(J)+MAX>=10 THEN 10 ELSE HIGHH(J)+MAX
                                      ),
               STARTH(J),
               V);
         END;
      V
      END;

REF(HYSTORY) PROC STARTHYS(EXP VALUE EXPR);
```

HISTORY HANDLING PROCEDURES

```
      ¢ construct a HYSTORY for EXPR. EXPR must be an
      ¢ IDENTIFIER or GENERATOR ¢
      BEGIN
      IF ¬(IF EXPR IS TRIO THEN OP(EXPR)="GE"
                          ELSE EXPR IS IDENTIFIER)
         THEN ERROR "INVALID EXPR FOR STARTHYS";
      HYSTORY(0,0,EXPR,NULL)
      END;

   REF(HYSTORY) PROC UNITEHYS(REF(HYSTORY) VALUE HYS1,HYS2;
                              LOGICAL VALUE RESULT CHANGE);
      ¢ construct an upper bound for the union of HYS1 and HYS2 ¢
      BEGIN
         REF(HYSTORY) I,J,IONLY,IJ;
         IONLY:=NULL;
         IJ:=NULL;
         FOR I IN LIST HYS1 LINK NEXTHYS DO OPEN
            FOR J IN LIST HYS2 LINK NEXTHYS DO OPEN
               IF (STARTH(I)=STARTH(J))
                  OR ( (STARTH(I) IS IDENTIFIER)
                     AND (STARTH(J) IS IDENTIFIER)
                     AND (IDNAME(STARTH(I))=IDNAME(STARTH(J))) )
                  THEN BEGIN IJ:=HYSTORY(
                     (IF LOWH(I)<LOWH(J) THEN LOWH(I) ELSE LOWH
                                                  (J)),
                     (IF HIGHH(I)>HIGHH(J) THEN HIGHH(I) ELSE
                                                  HIGHH(J)),
                     STARTH(I),
                     IJ);
                     IF (LOWH(I) ¬= LOWH(IJ)) OR (HIGHH(I)¬
                                                 =HIGHH(IJ))
                        THEN CHANGE:=TRUE;
                     GO TO IEQJ
                     END
            CLOSE;
            ¢ I ¬= any J ¢
            IONLY:=HYSTORY(LOWH(I),HIGHH(I),STARTH(I),IONLY);
            IEQJ:
            CLOSE;
         FOR J IN LIST HYS2 LINK NEXTHYS DO BEGIN
            FOR I IN LIST IJ LINK NEXTHYS DO
               IF (STARTH(I)=STARTH(J))
                  OR ( (STARTH(I) IS IDENTIFIER)
                     AND (STARTH(J) IS IDENTIFIER)
                     AND (IDNAME(STARTH(I))=IDNAME(STARTH(J))) )
                  THEN GO TO IEQJ;
            CHANGE:=TRUE;
            IONLY := HYSTORY(LOWH(J),HIGHH(J),STARTH(J),IONLY);
            IEQJ: END;
         FOR I IN LIST IONLY LINK NEXTHYS WHILE NEXTHYS(I)¬=NULL
                  DO SKIP;
         IF IONLY¬=NULL THEN NEXTHYS(I):=IJ
```

<u>HISTORY HANDLING PROCEDURES</u>

```
                              ELSE IONLY:=IJ;
         IONLY
         END;

   PROC MAPHYS(PROC FN; REF(HYSTORY) VALUE HYS;
                        REF(CONSTANT) VALUE START);
         ¢ apply the procedure FN at every mvalue that might lie
                                                  on
         ¢ a terminus of some path indicated by the operators in
                                                  HYS
         ¢ starting from START.
         ¢ if we come to some mvalue to which we cannot applt the
         ¢ next operator from the history then we give FN a non
                                                 -null
         ¢ second argument -- the rest of the operations in the
                                                  history.
         ¢ HYS is only a single history record, not a list of
                                                  them.
      BEGIN INTEGER I; REF(CONSTANT) VAL, SCANMARKED;
         ¢ remember that 10 means 10 or more
         I:=0; VAL:=START; SCANMARKED:=NULL;
         FOR VAL IN LIST START LINK VALUER
               WHILE (I<=HIGHH(HYS)) AND (¬SCANMARK(VAL)) DO OPEN
            IF I >= LOWH(HYS) THEN FN(VAL,NULL);
            IF ¬ISINDIVIDUAL(VAL) THEN BEGIN
               IF (I>=LOWH(HYS)) AND (I<=HIGHH(HYS))
                  THEN FN(VAL,HYSTORY(
                      LOWH(HYS)-I,
                      (IF HIGHH(HYS)<10 THEN HIGHH(HYS)-I ELSE
                                                   10),
                      NULL,NULL));
                  GO TO OUT
               END;
            IF I>=10 THEN BEGIN SCANMARK(VAL):=TRUE;
               IF SCANMARKED=NULL THEN SCANMARKED:=VAL
               END;
            IF I>=10 THEN I:=10 ELSE I:=I+1;
            CLOSE;
OUT:
         FOR VAL IN LIST SCANMARKED LINK VALUER WHILE SCANMARK
                                                 (VAL)
            DO SCANMARK(VAL):=FALSE
         END;
```

MISCELLANEOUS

```
PROC ENSTACK(REF(MSTATE) VALUE M; REF(CONSTANT) VALUE VAL);
      MSTACK(M):=TRIO("  ",VAL,MSTACK(M),NULL,NULL);

PROC POP(INTEGER VALUE I; REF(MSTATE,MSTATES) VALUE M);
   IF M = NULL THEN SKIP
   ELSE IF M IS MSTATES THEN
      FOR REF(MSTATES): J IN LIST M LINK NEXTMSTATE DO
         POP(I,THISMSTATE(J))
   ELSE IF M IS MSTATE THEN BEGIN
      FOR J:=1 UNTIL I DO
         IF MSTACK(M)=NULL THEN ERROR"NULL MSTACK -- POP"
            ELSE MSTACK(M):=SECOND(MSTACK(M))
      END;

REF(IDENTIFIER,TRIO,CONSTANT) PROC COPYBASE(
                  REF(IDENTIFIER,TRIO,CONSTANT) VALUE B);
   IF B = NULL THEN NULL
   ELSE IF B IS IDENTIFIER THEN IDENTIFIER(
                IDNAME(B))
   ELSE IF B IS TRIO THEN
      ( IF OP(B) = "GE"
        THEN TRIO("GE",NULL,NULL,NULL,NULL)
        ELSE NULL)
   ELSE IF B IS CONSTANT THEN <<CONSTANT>>(TYPE(B),VALUEKNOWN
                                         (B),
                VALUEB(B),VALUER(B),CEFFECTS(B),ISINDIVIDUAL
                                         (B),
                IDENTITY(B),HISTORY(B),SETMARK(B),USEMARK(B),
                ESCMARK(B),XMARK(B),1,XPT(B),SCANMARK(B))
   ELSE NULL;

PROC DEFINESYMBOL(REF(IDENTIFIER) VALUE I;
                  REF(CONSTANT) VALUE V;
                  REF(SYMBOL) VALUE RESULT E);
   BEGIN REF(SYMBOL) S;
      FOR S IN LIST E LINK NEXTSYMBOL DO
         IF SNAME(S)=IDNAME(I) THEN GO TO FOUND;
      ¢ not found ¢ E:=S:=SYMBOL(IDNAME(I),V,E);
      FOUND: POSSESSION(S):=V
      END;

REF(CONSTANT,TRIO) PROC LOOKUPSYMBOL(REF(IDENTIFIER) VALUE I;
                                     REF(SYMBOL) VALUE E);
   BEGIN REF(SYMBOL) S;
      FOR S IN LIST E LINK NEXTSYMBOL DO
         IF SNAME(S)=IDNAME(I) THEN GO TO FOUND;
      ¢ not found ¢
      S:=SYMBOL(IDNAME(I),NULL,E);
      FOUND: POSSESSION(S)
      END;
```

MISCELLANEOUS

```
PROCEDURE ERRPRINT(LOGICAL VALUE ABORT; INTEGER VALUE
                                        MESSAGECODE);
   BEGIN
      CASE MESSAGECODE OF BEGIN MESSAGES END;
      IF ABORT THEN GO TO EXIT
   END;

REF(MSTATES) SUPERINIT;
LOGICAL CHANGEEFFECTS;
REF(CONSTANT) VOIDVALUE;
REF(TRIO) LOCALS;
LOGICAL TRACING;
```

MAIN DRIVING ROUTINE

```
   TRACING := FALSE;

   EXIT:

   WHILE TRUE DO BEGIN
      EXP X;
      IOCONTROL(3);
      WRITE("<><><><>");
      WRITE("<><><><>");
      WRITE("<><><><>");

      ¢ initialisation that should have been with the
                                                 declarations
      ¢ but couldn't be ¢
      NPROCS:=0;
      PROCSYMBOLTABLE:=NULL;
      CURRENTEFFECTS:=EFFECTS(NULL,NULL,NULL);
            NSTK:=100;
      BG:=FALSE; ¢ boolean garbage pail ¢
      MG := NULL; ¢ mstate garbage pail ¢
      INDENTATION:=0;

      SUPERINIT:=MSTATES(MSTATE("        ",NULL,NULL),NULL,NULL);
      VOIDVALUE:=CONSTANT(TYPE="    ");
      LOCALS:=NULL;

      STARTTIMER;
      X:=READEXP;
      WRITE(CPUTIME, " SECONDS TO READ INPUT");
      STARTTIMER;
      MANALYSEPROCDENOTATIONS;
      MG:=MANALYSE(COPYMSTATES(SUPERINIT), X);
      WRITE(CPUTIME, " SECONDS FOR ANALYSIS ");
      STARTTIMER;
      NEWLINE;
      DISPLAY(100,X);
      FOR I := 1 UNTIL NPROCS DO BEGIN
         WRITE("PROCEDURE ");
         INDENT( BEGIN NEWLINE; DISPLAY(100,PROCTABLE(I)) END);
         WRITE("HAS EFFECTS ");
         INDENT(BEGIN NEWLINE; PRINT(EFFECTSTABLE(I)) END)
         END;
      WRITE(CPUTIME," SECONDS TO PRINT RESULTS")
      END

END.
```

7. Appendix B. Sample output.

Example 1.

```
( LET XX = GEN ; LET YY = GEN ;
   XX := GEN ; YY := GEN ;
   VAL XX := TRUE ; VAL YY := FALSE ;
   YY := VAL XX ; VAL VAL YY ; DISPLAY )
..
     0.2251953   SECONDS TO READ INPUT
     0.1289843   SECONDS FOR ANALYSIS
( LET XX = GEN ;
   LET YY = GEN ;
   XX := GEN ;
   YY := GEN ;
   VAL XX := TRUE ;
   VAL YY := FALSE ;
   YY := VAL XX ;
   VAL VAL YY ;
   DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            YY = INDIVIDUAL REFERENCE      RCCL08.10
               THIS REFERENCE REFERS TO INDIVIDUAL REFERENCE
                                          RCCL08.11
               THIS REFERENCE REFERS TO TRUE
            XX = INDIVIDUAL REFERENCE      RCCL08.13
               THIS REFERENCE REFERS TO INDIVIDUAL REFERENCE
                                          RCCL08.11
               THIS REFERENCE REFERS TO TRUE
         STACK :
   )
    0.07510412   SECONDS TO PRINT RESULTS
```

Example 2.

```
LET A = GEN ;
LET C = GEN ;
DISPLAY ;
WHILE VAL C DO
   A := VAL VAL A
OD ;
DISPLAY
..
    0.08574218   SECONDS TO READ INPUT
    0.05733073   SECONDS FOR ANALYSIS
( LET A  = GEN ;
   LET C  = GEN ;
   DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            C  = INDIVIDUAL REFERENCE      RCCL08.22
            A  = INDIVIDUAL REFERENCE      RCCL08.23
         STACK :
   ;
   WHILE VAL C
   DO A  := VAL VAL A
   OD ;
   DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            C  = INDIVIDUAL REFERENCE      RCCL08.45
            A  = INDIVIDUAL REFERENCE      RCCL08.46
         STACK :
   )
    0.08035153   SECONDS TO PRINT RESULTS
```

Example 3.

```
LET A = GEN ;
LET B = GEN ;
LET C = GEN ;
A := GEN ;
B := GEN ;
VAL B := TRUE ;
DISPLAY ;
WHILE
   DISPLAY ;
   VAL C
DO
   A := VAL B ;
   B := ( GEN := TRUE ) ;
   DISPLAY
OD ;
DISPLAY
..
     0.1755338   SECONDS TO READ INPUT
     0.2884114   SECONDS FOR ANALYSIS
( LET A  = GEN ;
   LET B  = GEN ;
   LET C  = GEN ;
   A  := GEN ;
   B  := GEN ;
   VAL B  := TRUE ;
   DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            C  = INDIVIDUAL REFERENCE      RCCL08.59
            B  = INDIVIDUAL REFERENCE      RCCL08.60
               THIS REFERENCE REFERS TO INDIVIDUAL REFERENCE
                                         RCCL08.61
               THIS REFERENCE REFERS TO TRUE
            A  = INDIVIDUAL REFERENCE      RCCL08.63
               THIS REFERENCE REFERS TO INDIVIDUAL REFERENCE
                                         RCCL08.64
         STACK :
   ;
   WHILE DISPLAY

      STATE MODEL
         MSTATE

            ENVIRONMENT
               C  = INDIVIDUAL REFERENCE      RCCL08.77
               B  = INDIVIDUAL REFERENCE      RCCL08.78
                  THIS REFERENCE REFERS TO INDIVIDUAL REFERENCE
                                            RCCL08.79
                  THIS REFERENCE REFERS TO TRUE
```

Example 3.

```
                A  = INDIVIDUAL REFERENCE      RCCL08.81
                   THIS REFERENCE REFERS TO INDIVIDUAL REFERENCE
                                              RCCL08.82
            STACK :

      ;
      VAL C
   DO A  := VAL B  ;
      B  := GEN := TRUE ;
      DISPLAY

      STATE MODEL
         MSTATE

            ENVIRONMENT
               C  = INDIVIDUAL REFERENCE     RCCL08.107
               B  = INDIVIDUAL REFERENCE     RCCL08.108
                  THIS REFERENCE REFERS TO INDIVIDUAL REFERENCE
                                             RCCL08.109
                  THIS REFERENCE REFERS TO TRUE
               A  = INDIVIDUAL REFERENCE     RCCL08.111
                  THIS REFERENCE REFERS TO INDIVIDUAL REFERENCE
                                             RCCL08.112
                  THIS REFERENCE REFERS TO TRUE
            STACK :

   OD ;
   DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            C  = INDIVIDUAL REFERENCE     RCCL08.142
            B  = INDIVIDUAL REFERENCE     RCCL08.143
               THIS REFERENCE REFERS TO INDIVIDUAL REFERENCE
                                          RCCL08.144
               THIS REFERENCE REFERS TO TRUE
            A  = INDIVIDUAL REFERENCE     RCCL08.146
               THIS REFERENCE REFERS TO INDIVIDUAL REFERENCE
                                          RCCL08.147
         STACK :
   )
     0.2216015   SECONDS TO PRINT RESULTS
```

Example 4.

```
LET XX = GEN ;
LET YY = GEN ;
A : ( IF VAL YY THEN GO B ELSE XX FI ; DISPLAY ) ;
   X := ( GEN := VAL XX ) ;
   GO A ;
B : DISPLAY
..
     0.1025260   SECONDS TO READ INPUT
     0.3019010   SECONDS FOR ANALYSIS
( LET XX = GEN ;
   LET YY = GEN ;
   A  : IF VAL YY
   THEN GO B
   ELSE XX
   FI ;
   DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            YY = INDIVIDUAL REFERENCE     RCCL08.239
            XX = INDIVIDUAL REFERENCE     RCCL08.240
         STACK :
   ;
   X  := GEN := VAL XX ;
   GO A  ;
   B  : DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            YY = INDIVIDUAL REFERENCE     RCCL08.268
            XX = INDIVIDUAL REFERENCE     RCCL08.269
         STACK :
   )
    0.09334630   SECONDS TO PRINT RESULTS
```

Example 5.

```
( LET P = ( PROC ( X ) : ( A := B ) ) ;
  LET A = GEN ;
  LET B = GEN ;
  B := TRUE ;
  A := B ;
  P ( A ) ;
  DISPLAY ;
VAL VAL B )
..
     0.1210417   SECONDS TO READ INPUT
    0.07639319   SECONDS FOR ANALYSIS
( LET P  = ( PROC ( X  ) : A  := B  ) ;
   LET A  = GEN ;
   LET B  = GEN ;
   B  := TRUE ;
   A  := B  ;
   P  ( A  ) ;
   DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            B  = NONINDIVIDUAL REFERENCE     RCCL08.292
            A  = INDIVIDUAL REFERENCE     RCCL08.293
            P  = PROC
               EFFECTS --
                  SET --
                  HISTORY
                     FROM   0  TO  0   VALS, STARTING AT A
                  ESCAPE --
                  HISTORY
                     FROM   0  TO  0   VALS, STARTING AT B
         STACK :
   ;
   VAL VAL B  )
PROCEDURE
   ( ( PROC ( X  ) : A  := B  ) )
HAS EFFECTS

   EFFECTS --
      SET --
      HISTORY
         FROM   0  TO  0   VALS, STARTING AT A
      ESCAPE --
      HISTORY
         FROM   0  TO  0   VALS, STARTING AT B
     0.1221094   SECONDS TO PRINT RESULTS
```

Example 6.

```
( LET P = ( PROC ( X ) : ( LET A = GEN ; A := B ) ) ;
  LET B = GEN ;
  LET C = GEN ;
  P ( D ) ;
  DISPLAY ;
  B )
..
     0.1033984   SECONDS TO READ INPUT
     0.1005599   SECONDS FOR ANALYSIS
( LET P  = ( PROC ( X  ) : LET A  = GEN ;
      A  := B  ) ;
   LET B  = GEN ;
   LET C  = GEN ;
   P  ( D  ) ;
   DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            C  = INDIVIDUAL REFERENCE     RCCL08.320
            B  = INDIVIDUAL REFERENCE     RCCL08.321
            P  = PROC
               EFFECTS --
                  SET --
                  ESCAPE --
         STACK :
   ;
   B  )
PROCEDURE
   ( ( PROC  ( X  ) : LET A  = GEN ;
         A  := B  ) )
HAS EFFECTS

   EFFECTS --
      SET --
      ESCAPE --
    0.09570313   SECONDS TO PRINT RESULTS
```

Example 7.

```
( LET P = ( PROC ( X ) : ( LET A = GEN ; A := B ; C := A ) ) ;
  LET B = GEN ;
  LET C = GEN ;
  P ( D ) ;
  DISPLAY ;
  B )
..
     0.1346354   SECONDS TO READ INPUT
    0.09166664   SECONDS FOR ANALYSIS
( LET P  = ( PROC ( X  ) : LET A  = GEN ;
       A  := B  ;
       C  := A  ) ;
   LET B  = GEN ;
   LET C  = GEN ;
   P  ( D  ) ;
   DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            C  = INDIVIDUAL REFERENCE     RCCL08.353
            B  = NONINDIVIDUAL REFERENCE     RCCL08.354
            P  = PROC
               EFFECTS --
                  SET --
                  HISTORY
                     FROM   0  TO  0   VALS, STARTING AT C
                  ESCAPE --
                  HISTORY
                     FROM   0  TO  0   VALS, STARTING AT B
         STACK :

   ;
   B  )
 PROCEDURE
   ( ( PROC ( X  ) : LET A  = GEN ;
         A  := B  ;
         C  := A  ) )
 HAS EFFECTS

   EFFECTS --
      SET --
      HISTORY
         FROM   0  TO  0   VALS, STARTING AT C
      ESCAPE --
      HISTORY
         FROM   0  TO  0   VALS, STARTING AT B
     0.1327344   SECONDS TO PRINT RESULTS
```

Example 8.

```
LET P = ( PROC ( A , B ) : ( A := B ; P ( B , C ) ) ) ;
LET C = GEN ;
LET X = GEN ; LET Y = GEN ; P ( X , Y )
..
    0.09654945   SECONDS TO READ INPUT
     0.2541406   SECONDS FOR ANALYSIS
( LET P   = ( PROC ( A   , B   ) : A   := B   ;
      P   ( B   , C   ) ) ;
   LET C   = GEN ;
   LET X   = GEN ;
   LET Y   = GEN ;
   P   ( X   , Y   ) )
PROCEDURE
   ( ( PROC ( A   , B   ) : A   := B   ;
         P   ( B   , C   ) ) )
HAS EFFECTS

   EFFECTS --
      SET --
      HISTORY
         FROM   0  TO  0   VALS, STARTING AT P 000001
         FROM   0  TO  0   VALS, STARTING AT C
         FROM   0  TO  0   VALS, STARTING AT P 000002
      ESCAPE --
      HISTORY
         FROM   0  TO  0   VALS, STARTING AT P 000002
         FROM   0  TO  0   VALS, STARTING AT C
    0.07938796   SECONDS TO PRINT RESULTS
```

Example 9.

```
LET A = GEN ;
LET B = GEN ;
LET C = GEN ;
LET D = GEN ;
LET E = GEN ;
LET F = GEN ;
LET G = GEN ;
LET H = GEN ;
LET I = GEN ;
LET J = GEN ;
A := FALSE ;
B := TRUE ;
C := TRUE ;
D := TRUE ;
E := TRUE ;
F := TRUE ;
G := TRUE ;
H := TRUE ;
I := TRUE ;
J := TRUE ;
DISPLAY ;
WHILE
   DISPLAY ;
   VAL J
DO DISPLAY ;
   J := VAL I ;
   I := VAL H ;
   H := VAL G ;
   G := VAL F ;
   F := VAL E ;
   E := VAL D ;
   D := VAL C ;
   C := VAL B ;
   B := VAL A ;
   DISPLAY
OD ;
DISPLAY
..
     0.4299349   SECONDS TO READ INPUT
      4.562864   SECONDS FOR ANALYSIS
( LET A  = GEN ;
   LET B  = GEN ;
   LET C  = GEN ;
   LET D  = GEN ;
   LET E  = GEN ;
   LET F  = GEN ;
   LET G  = GEN ;
   LET H  = GEN ;
   LET I  = GEN ;
   LET J  = GEN ;
   A  := FALSE ;
   B  := TRUE ;
```

Example 9.

```
  C   := TRUE ;
  D   := TRUE ;
  E   := TRUE ;
  F   := TRUE ;
  G   := TRUE ;
  H   := TRUE ;
  I   := TRUE ;
  J   := TRUE ;
  DISPLAY

  STATE MODEL
     MSTATE

        ENVIRONMENT
           J  = INDIVIDUAL REFERENCE     RCCL08.446
              THIS REFERENCE REFERS TO TRUE
           I  = INDIVIDUAL REFERENCE     RCCL08.448
              THIS REFERENCE REFERS TO TRUE
           H  = INDIVIDUAL REFERENCE     RCCL08.450
              THIS REFERENCE REFERS TO TRUE
           G  = INDIVIDUAL REFERENCE     RCCL08.452
              THIS REFERENCE REFERS TO TRUE
           F  = INDIVIDUAL REFERENCE     RCCL08.454
              THIS REFERENCE REFERS TO TRUE
           E  = INDIVIDUAL REFERENCE     RCCL08.456
              THIS REFERENCE REFERS TO TRUE
           D  = INDIVIDUAL REFERENCE     RCCL08.458
              THIS REFERENCE REFERS TO TRUE
           C  = INDIVIDUAL REFERENCE     RCCL08.460
              THIS REFERENCE REFERS TO TRUE
           B  = INDIVIDUAL REFERENCE     RCCL08.462
              THIS REFERENCE REFERS TO TRUE
           A  = INDIVIDUAL REFERENCE     RCCL08.464
              THIS REFERENCE REFERS TO FALSE
        STACK :
  ;
  WHILE DISPLAY

     STATE MODEL
        MSTATE

           ENVIRONMENT
              J  = INDIVIDUAL REFERENCE    RCCL08.2108
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
              I  = INDIVIDUAL REFERENCE    RCCL08.2110
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
              H  = INDIVIDUAL REFERENCE    RCCL08.2112
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
              G  = INDIVIDUAL REFERENCE    RCCL08.2114
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
              F  = INDIVIDUAL REFERENCE    RCCL08.2116
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
```

Example 9.

```
                E  = INDIVIDUAL REFERENCE    RCCL08.2118
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                D  = INDIVIDUAL REFERENCE    RCCL08.2120
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                C  = INDIVIDUAL REFERENCE    RCCL08.2122
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                B  = INDIVIDUAL REFERENCE    RCCL08.2124
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                A  = INDIVIDUAL REFERENCE    RCCL08.2126
                   THIS REFERENCE REFERS TO FALSE
              STACK :
      ;
      VAL J
   DO DISPLAY

      STATE MODEL
         MSTATE

            ENVIRONMENT
                J  = INDIVIDUAL REFERENCE    RCCL08.2188
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                I  = INDIVIDUAL REFERENCE    RCCL08.2190
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                H  = INDIVIDUAL REFERENCE    RCCL08.2192
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                G  = INDIVIDUAL REFERENCE    RCCL08.2194
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                F  = INDIVIDUAL REFERENCE    RCCL08.2196
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                E  = INDIVIDUAL REFERENCE    RCCL08.2198
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                D  = INDIVIDUAL REFERENCE    RCCL08.2200
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                C  = INDIVIDUAL REFERENCE    RCCL08.2202
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                B  = INDIVIDUAL REFERENCE    RCCL08.2204
                   THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
                A  = INDIVIDUAL REFERENCE    RCCL08.2206
                   THIS REFERENCE REFERS TO FALSE
              STACK :
      ;
      J  := VAL I  ;
      I  := VAL H  ;
      H  := VAL G  ;
      G  := VAL F  ;
      F  := VAL E  ;
      E  := VAL D  ;
      D  := VAL C  ;
      C  := VAL B  ;
      B  := VAL A  ;
      DISPLAY
```

Example 9.

```
    STATE MODEL
       MSTATE

          ENVIRONMENT
             J   = INDIVIDUAL REFERENCE     RCCL08.2208
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
             I   = INDIVIDUAL REFERENCE     RCCL08.2210
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
             H   = INDIVIDUAL REFERENCE     RCCL08.2212
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
             G   = INDIVIDUAL REFERENCE     RCCL08.2214
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
             F   = INDIVIDUAL REFERENCE     RCCL08.2216
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
             E   = INDIVIDUAL REFERENCE     RCCL08.2218
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
             D   = INDIVIDUAL REFERENCE     RCCL08.2220
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
             C   = INDIVIDUAL REFERENCE     RCCL08.2222
                 THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
             B   = INDIVIDUAL REFERENCE     RCCL08.2224
                 THIS REFERENCE REFERS TO FALSE
             A   = INDIVIDUAL REFERENCE     RCCL08.2226
                 THIS REFERENCE REFERS TO FALSE
          STACK :

   OD ;
   DISPLAY

   STATE MODEL
      MSTATE

         ENVIRONMENT
            J   = INDIVIDUAL REFERENCE     RCCL08.2306
                THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
            I   = INDIVIDUAL REFERENCE     RCCL08.2308
                THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
            H   = INDIVIDUAL REFERENCE     RCCL08.2310
                THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
            G   = INDIVIDUAL REFERENCE     RCCL08.2312
                THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
            F   = INDIVIDUAL REFERENCE     RCCL08.2314
                THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
            E   = INDIVIDUAL REFERENCE     RCCL08.2316
                THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
            D   = INDIVIDUAL REFERENCE     RCCL08.2318
                THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
            C   = INDIVIDUAL REFERENCE     RCCL08.2320
                THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
            B   = INDIVIDUAL REFERENCE     RCCL08.2322
                THIS REFERENCE REFERS TO UNKNOWN BOOLEAN
            A   = INDIVIDUAL REFERENCE     RCCL08.2324
```

Example 9.

```
                  THIS REFERENCE REFERS TO FALSE
          STACK :
  )
     0.6673828   SECONDS TO PRINT RESULTS
```