A study of two formal description languages and their application to PASCAL

This item was submitted to Loughborough University's Institutional Repository by the/ an author.

Additional Information:


Metadata Record: [https://dspace.lboro.ac.uk/2134/27043](https://dspace.lboro.ac.uk/2134/27043)

Publisher: © Mehdi Badii

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 2.5 Generic (CC BY-NC-ND 2.5) licence. Full details of this licence are available at: [http://creativecommons.org/licenses/by-nc-nd/2.5/](http://creativecommons.org/licenses/by-nc-nd/2.5/)

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
<table>
<thead>
<tr>
<th>AUTHOR/FILING TITLE</th>
<th>BADII, M</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESSION/COPY NO.</td>
<td>118773/32</td>
</tr>
<tr>
<td>VOL. NO.</td>
<td></td>
</tr>
<tr>
<td>LOAN COPY</td>
<td></td>
</tr>
</tbody>
</table>

**DATE DUE**

13 SEP 1984

UNLESS RECALLED

3 11111 1992
A STUDY OF TWO FORMAL DESCRIPTION LANGUAGES

AND THEIR APPLICATION TO PASCAL

BY

MEHDI BADII, M.Sc., M.A.

A Doctoral Thesis
Submitted in partial fulfilment of the requirements
for the award of Doctor of Philosophy
of the Loughborough University of Technology
July, 1981.

SUPERVISOR: D.J. Cooke, Ph.D.
Department of Computer Studies

© by Mehdi Badii, 1981.
DECLARATION

I declare that the following thesis is a record of research work carried out by me, and that the thesis is of my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a degree.

M. BADIL
The biggest debt of gratitude I owe is to my supervisor, Dr. John Cooke. No mere words seem adequate to express my appreciation and thanks for his enlightening criticism and patience throughout the work.

Also I would like to express my thanks to Miss J.M. Briers for her speedy and efficient typing of this thesis.
# CONTENTS

<table>
<thead>
<tr>
<th>Chapter 1: INTRODUCTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The Need for Formal Definition of Programming Languages</td>
<td>2</td>
</tr>
<tr>
<td>1.2 An Overview of Specification Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.3 The Aim of the Thesis and Its Structure</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2: THE LANGUAGE PSS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Ill-defined Aspects of PASCAL</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Inconsistencies Between Two PASCAL Implementations</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Syntax Modifications</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 3: A W-GRAMMAR DEFINITION OF PSS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 The W-Grammar System</td>
<td>29</td>
</tr>
<tr>
<td>3.1.1 Informal Discussion</td>
<td>29</td>
</tr>
<tr>
<td>3.1.2 Basic Notation and Definitions</td>
<td>32</td>
</tr>
<tr>
<td>3.2 Definition of PSS</td>
<td>36</td>
</tr>
<tr>
<td>3.2.1 General Hyper Rules</td>
<td>36</td>
</tr>
<tr>
<td>3.2.2 Modelling Specific Aspects of PSS</td>
<td>37</td>
</tr>
<tr>
<td>3.2.2.1 The Program</td>
<td>37</td>
</tr>
<tr>
<td>3.2.2.2 Subroutine Declarations</td>
<td>39</td>
</tr>
<tr>
<td>3.2.2.3 Subroutine Invocation</td>
<td>42</td>
</tr>
<tr>
<td>3.2.2.4 Constant Declarations</td>
<td>48</td>
</tr>
<tr>
<td>3.2.2.5 Type Declarations</td>
<td>49</td>
</tr>
<tr>
<td>3.2.2.6 Expressions</td>
<td>53</td>
</tr>
<tr>
<td>3.2.2.7 Statements</td>
<td>58</td>
</tr>
<tr>
<td>3.2.2.8 Input and Output</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4: SOME CONCEPTS FROM GRAPH THEORY AND COMPUTABILITY THEORY</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Lists</td>
<td>64</td>
</tr>
<tr>
<td>4.2 Directed Graphs</td>
<td>71</td>
</tr>
</tbody>
</table>
Chapter 5: A CPS DESCRIPTION OF PSS

5.1 The Carabiner (CPS) System ........................................ 86
  5.1.1 Philosophy .................................................. 86
  5.1.2 Formal Definitions ........................................... 88

5.2 Definition of PSS .................................................. 101
  5.2.1 The Program .................................................. 101
  5.2.2 Subroutine Declarations ..................................... 107
  5.2.3 Subroutine Invocation ........................................ 112
  5.2.4 Constant Declarations ....................................... 115
  5.2.5 Type Declarations ........................................... 116
  5.2.6 Expressions ................................................. 120
  5.2.7 Statements .................................................. 120
  5.2.8 Input and Output ............................................ 122

Chapter 6: CONCLUSIONS .................................................. 124

6.1 Comparisons ...................................................... 125
6.2 Modifications to PSS Formalisms ................................. 127
6.3 Implementation Considerations .................................. 131
6.4 A Hybrid System ................................................... 134
6.5 Summary ............................................................. 136

REFERENCES .................................................................... 137

Appendix 1: W-GRAMMAR DEFINITION OF PSS ....................... 145

Appendix 2: FORMAL DEFINITION OF PSS USING THE CARABINER SYSTEM .. 253
CHAPTER 1

INTRODUCTION
The theory of formal languages was initiated by Chomsky [40] who designed a mathematical model of a grammar for use in the study of natural languages. Chomsky first defined the rules for the so-called phrase structure grammars and then, by imposing restrictions on these, defined context sensitive, context free and regular grammars. These classifications were soon applied to artificial languages and, in 1959 a system called Bakus Normal Form (BNF) [41] was used to define the syntax of ALGOL 60 [4] (The context free grammars defined by Chomsky and the Backus system are equivalent [62] in the sense that they generate strings for the same languages). The semantics of ALGOL 60 are expressed in English sentences. This was not a precise definition of the language as it contained a number of errors and ambiguities [42]. Since, it is impossible to formally define the semantics of the language by BNF [43] then to remove these conflicts we have to look for other more powerful systems.

In the following chapter we show how lack of a precise definition of a language might lead to misinterpretation of their meaning. Next, we take a brief look at formal systems and then give an overview of the material which is covered in the subsequent chapters.

1.1 THE NEED FOR FORMAL DEFINITION OF PROGRAMMING LANGUAGES

Experience with common programming languages such as FORTRAN, ALGOL 60 and PASCAL [5,19] has shown that lack of a full formal description of a language can lead to serious misinterpretations and errors. A number of these conflicts are explained in section 2.1.

In order to prove certain properties of programs written in a language, we first have to define what constitutes a program. Ultimately, this gives rise to the problem of defining a formal system to specify the grammar of the language.
Often a language will be implemented on different computers. Therefore, it is desirable that there should be criteria to guarantee the full compatibility amongst the implementations. Experience has shown that lack of such criteria may cause different behaviours for the same program on various implementations (for example see the next chapter or [44], Chapter 7).

The programming languages have been designed to facilitate the description of algorithms. To prove that two programs formulate the same algorithm a precise frame of reference is needed to define their behaviour.

During the construction of a formal description of a language some of its inconsistencies and critical features might be discovered (cf. 2.1). Alternatively, the language designers can also use the formalism to review and analyse their work.

A formalism can thus serve as a means of communication between the language designer, the implementor and the programmer.

Past experience with informal descriptions of the semantics of ALGOL 60 and PASCAL [5] has shown that there are facts which the programmer needs to know but which cannot be quickly and clearly deduced from the corresponding report.

1.2 AN OVERVIEW OF SPECIFICATION SYSTEMS

The specification of the syntax of a language by the use of BNF notation is generally accepted and regarded as satisfactory. However, the specification of semantics is less clear. Although there are several systems for this purpose, there is no general agreement over which one to use. In fact, different systems are suitable for different purposes. For example, the axiomatic system [20], which has been used to define
PASCAL, is suitable for proving program properties, but it is less useful for language designers and implementors than, for example, the W-grammar or Carabiner system [6]. In this section some of the systems are listed and comparisons are made between these and the W-grammar and CPS systems which are described in subsequent chapters.

Landin [45,46,47], Böhm [48,49] and Strachey [50] designed their systems based on Church's \( \lambda \)-calculus [15]. Their philosophy was that there is a natural correspondence between ALGOL-like constructions and expressions of Church's \( \lambda \)-calculus. This means that they have tried to define a program in terms of functional notation. However, as Wirth and Weber [63] have pointed out, their systems do not reflect the dynamic structure of the programs. This can be explicitly represented in both the W-grammar and CPS models.

McCarthy's [52] formalism is based on \( \lambda \)-notation and a state vector \( \xi \). If a variable is included in an expression then the \( \lambda \)-notation bounds it to the expression. The state vector reflects the state of the program space and it will be changed on the assignment statements of a program. In this sense the state vector includes the concepts of stack and heap as defined in our W-grammar model or the CPS working space (cf. 5.1.2.5). McCarthy applied his system to Micro-ALGOL which is a small subset of ALGOL 60. The language only contains the assignment statement and the conditional statement 'if \( p \) then goto \( \ell \)'. However, the power of his system cannot be fully appreciated from Micro-ALGOL. But he admits that the concepts such as declarations and procedures require a more complex state vector.

The mathematical system of Scott-Strachey [53,54,55] is based on a small set of fundamental semantic constructs. They argue that this set
is adequate to formally define the aspects of the programming languages. However, the properties of the system do not reflect the pragmatic aspects of implementations.

The Vienna Definition Language (VDL) [56,57,58,59] defines a subroutine to generate a tree structure for a program. Those parts of the tree which are required for the run time semantics are transferred to an abstract form and this in turn is executed by a hypothetical machine. Compared with the W-grammar system the mechanism of this method is more difficult to understand.

A formal system defined by mathematical logic (i.e. axioms and rules of inference) has been used to describe most of the semantics of PASCAL [20]. However, there are some restrictions imposed by this system on the language (see pages 337 and 347 of [20]). For example, side effects are not allowed in the evaluation of subroutines and statements. This technique is a suitable tool for proving program properties but in comparison with the CPS system it is less useful to the implementor. Also in W-grammars the pseudo-English sentences (which describe the semantics of the language) for the semantic actions might be designed such that the reader can easily and quickly understand the meaning of the actions. In this sense it is more useful than the logic technique for the language designers.

Attribute grammars [60,61] generate a syntax tree for a program and define attributes attached to the nodes of the tree. Each attribute associated with a node is obtained from its ancestors and its descendants. Since the evaluation rules cannot be defined by this method and they must be defined by another system, this is not a complete method. This technique and the CPS system are more suitable to implementors, but the
CPS method is a complete system. The principal advantage of W-grammars over attribute grammars is that they are able to define the language completely and have wide descriptive possibilities.

Galler and Perlis [17] used Markov algorithms to define some aspects of programming languages, but as it is shown in 4.4 this technique is too clumsy to be considered as a practical system. The CPS method which also uses Markov algorithms is more structured and uniform.

1.3 THE AIM OF THE THESIS AND ITS STRUCTURE

In this thesis, we present two formal specification methods: the W-grammars and the Carabiner (CPS) systems, and use both systems to formally define the full specification of a fairly large subset of PASCAL. The material covered by the subsequent chapters can briefly be discussed as follows:

The second chapter deals with the language PASCAL. The language is shown to be not precisely defined and this is illustrated by various examples. In such cases, a specific interpretation is chosen for use in our formal definition. Several inconsistencies in local PASCAL compilers are noted and finally we introduced our PASCAL subset (PSS) which will be formally defined later.

Chapter 3 is concerned with the W-grammars [1,2,3] and should be read in association with Appendix 1 which contains the full detailed W-grammar description of PSS. A W-grammar definition of a language constitutes two sets A, B of rules. The set A contains the context free productions and B is a set of rules, where each rule is constructed from some pseudo-English phrases. By applying the elements of A to the elements of B for a program a third set C of context free productions will be obtained. The program is then fully defined by the rules of C.
As a trivial example, consider the following simple language. The syntax of the language is defined by BNF and the semantics are informally described by English phrases written in the curly brackets:

1. `<statement>::=b{generate space for b}
   :=<digit1>+<digit2>
   {assign<digit1><digit2>to b}

2. `<digit1>::=<digit>

3. `<digit2>::=<digit>

4. `<digit>::=one|oneone|oneoneone...

To describe the requirements of the statement: b:=one+oneone, we define the following string as a working space for the action {generate space for b}

   b has value undefined

The process of the action {assign<digit1><digit2>to b} is defined such that the above string (working space) is changed to:

   b has value oneoneone

The philosophy behind the W-grammars is the generation of a third set C out of A and B such that the productions of C can generate the following parse tree:

```
<statement>
   b generate space := <digit1> + <digit2> assign<digit1><digit2> to b
   for b
       <digit>
       one
   <digit>
   oneone
   b has value oneoneone
       EMPTY
```

```
b has value undefined
    ...
    EMPTY
```
Within this tree the semantic definitions are represented as sub-trees generating "EMPTY".

The fourth chapter contains some of the preliminary concepts needed for the system described in Chapter 5. Recursive functions are defined and their relationship with Markov algorithms is demonstrated. Finally, the connection between a computer program and the W-grammar system or the Markov algorithm technique is considered.

In Chapter 5, the Carabiner Program Space (CPS) is introduced and used to specify some features of our PASCAL subset. The full specification of the language defined by the Carabiner technique is given in Appendix 2.

The working space of this system is a list of items (structures). The list and graph traversing operators (defined in Chapter 4) may be used to insert, delete or connect items of the working space. The syntax of a program is defined by the normal BNF notation and the semantics are formally described by the list operations and three basic operators $e$, $S$ and $k$ which are inserted as semantic injections within the BNF. Therefore, we have a parse tree derived from the construction of the program and a collection of operations associated with the validation and execution of the program. By way of illustration we again consider the assignment statement: $b:=\text{one+oneone}$ as defined by the previous grammar. The working space (list) for the semantic action {generate space for $b$} is the list:

\[(b)\]

The semantic action {assign<digit1><digit2>to $b$} changes the space to:

\[
\begin{array}{c}
(b) \\
\downarrow \\
\text{oneoneone}
\end{array}
\]

The parse tree for the pure BNF syntax can be obtained from the previous
one by omitting the 'semantic branches'.

In order to have a machine independent formal description of the PASCAL subset the primitive operations are defined by Markov algorithms†.

In the final chapter, both systems and their corresponding PASCAL subset formalisms are compared from different points of view. Several modifications to both formalisms are suggested and a new system based on a mixture of the W-grammar and Carabiner techniques is formulated. Finally, some pattern matching algorithms for the implementation of our models are considered. In particular, an improvement of the Knuth-Morris-Pratt [39] pattern matching method is proposed.

†In the W-grammar formalisms these are consistently defined by the rules of the W-grammar.
CHAPTER 2

THE LANGUAGE PSS
The programming language PASCAL [19] was purposefully designed for the construction of reliable structured programs. It is widely accepted as:

1. a language that can be efficiently implemented on many computers,
2. an excellent language for teaching the principle concepts of programming,
and 3. a practical tool in many applications areas.

The definition of the language both in [19] and the revised report [5] is given in ALGOL60 [4] style, that is, the syntax of the programs is defined by BNF, and the semantics of the programs are described in English sentences. Although, apart from some ambiguities in the syntax of the language (e.g. if-then-else statement), the BNF of PASCAL is sufficient to define the structure of programs, the lack of an accepted formal definition for the semantic aspects allows different interpretations of the language. In recent years, attempts have been made to formalise this language. In a paper by Hoare and Wirth [20], the majority of the semantics of the language are defined, but their system which is based on mathematical logic, is not powerful enough to produce a complete definition (for example side-effects are not allowed in a PASCAL expression or statement†).

Another attempt to formalise PASCAL was made by Watt [22]. The reader comparing this system with the one in the next chapter, probably will find our system more readable. Moreover, the model in the next chapter does not need lots of pre-defined concepts to define a language, whilst Watt's system involves the definition of some basic tools to define PASCAL.

† On page 337 of [20] they assert: "The axioms and rules of inference given in this article explicitly forbid the presence of certain 'side-effects in the evaluation of functions and execution of statements" (for a proof see Brady [21], page 202).
It may be worth comparing his system with CPS (explained in Chapter 5).

In this chapter, the ill-defined points which are either inherent in the syntax of the language, or give rise to different interpretations are considered and suggestions about them are made. We then, give several examples of output from two compilers to illustrate some of these points. Finally, aspects of the language which are not considered in the systems of the Chapters 3 and 5 are mentioned.

2.1 ILL-DEFINED ASPECTS OF PASCAL

For the PASCAL language, one should expect to be faced with criticisms, comments and suggestions for extensions. In recent years a number of ill-defined aspects of the language have been made by Habermann [23], Conradi [24], Welsh, et al [25] and others. Although some of them have been rejected (see [26] page 146), various suggestions have been made to deal with others. Only an internationally accepted standard definition of the language can unify these suggestions. It can be demonstrated that lack of such a tool has caused misinterpretations when comparing PASCAL with other programming languages (for example see [27], page 319, where the third type declaration is not a PASCAL one).

In the following sub-sections we consider these ill-defined points. We have not suggested any changes or improvements but whenever there were two or more interpretations of the same feature, we say which one is chosen for our systems in Chapters 3 and 5.

2.1.1 Type consideration

The phrase "identical type" in Section 9.1.1 RR† is not precisely defined. Therefore, according to [25], we may have two interpretations

†Throughout this chapter the word "RR" denotes the "revised report" [5].
given as:

I. "Name equivalence", where two variables are of the same type if they are declared together or use the same type identifier.

II. "Structural equivalence", where two variables are of the same type if the components of their types have the same structure.

Example 1: var x:record a:integer, b:Boolean end; y:record c:integer, d:Boolean end.

The variables x and y are of the same type.

Referring to Tennent [28] and his discussion about type compatibility, we agree with his idea that structural equivalence imposes less limitations than name equivalence on type identity. Therefore, we use case II for "identical type". The strategy for the compatibility of the recursively defined types is based on the exercise 2.3.5, no.11 of [29]. This can be clarified by the following example:

Example 2: program ab;

    type t=array[1..10] of integer;
    acolour=(red,yellow,green,blue);
    anode=record a:acolour; b:t; c:anode end;
    procedure f;
      type bcolour=(red,yellow,green,blue);
      bnode=record d:bcolour; e:t; f:bnode end;
    var x:anode;
    y:bnode;
    .......... 
    .......... 
    .......... 
    .......... 

To find whether x and y are of the same type we follow the procedure given below:

    Let A=((anode_1,bnode_2)), where the subscripts 1 and 2 show that anode and bnode are declared in the first and the second blocks respectively. Add the pair (acolour_1,bcolour_2) to A, where acolour and bcolour are the type identifiers of the first components of the records,
respectively. So:

\[ A = \{(\text{anode}_1, \text{bnode}_2), (\text{acolour}_1, \text{bcolour}_2)\} \]

since \text{acolour} and \text{bcolour} have identical scalar type definitions, the first components of the record are equivalent. Add the pair \((t_1, t_1)\) to \(A\). That is:

\[ A = \{(\text{anode}_1, \text{bnode}_2), (\text{acolour}_1, \text{bcolour}_2), (t_1, t_1)\} \]

Since both elements of the pair are identical, the second components of the records are equivalent. The type identifiers of the third components make the pair \((\text{anode}_1, \text{bnode}_2)\), but this pair is already in \(A\) (we have a cycle). Since in this pair \(\text{anode}_1 \neq \text{bnode}_2\), the records are not equivalent.

2.1.1.2 **Scalar Types**

The procedure `write(In)` cannot be used for items of a scalar type defined by the programmer; that is for

\[ \text{type Boolean} = (\text{false}, \text{true}) \]

we cannot have "write(false)". But if we had not declared the identifier `Boolean`, we could have had this statement! This is not a uniform application of `write(In)` procedure for the values of scalar types.

2.1.1.3 **Subrange Types**

If the type of variable \(x\) is a subrange, it cannot be used as a case expression (or tagfield). This is not mentioned in Section 9.2.2.2 RR (or 6.2.2.2 RR).

**Example 3:**

```pascal
var x:1..10;
begin
    case x of
        2: ...
    end;
end.
```

According to section 4RR, the label 2 is of type integer (and not a sub-
range of integer). In 9.2.2.2 RR it is asserted that "the case statement consists of an expression (the selector) and a list of statements, each being labelled by a constant of the type of selector". Therefore the types x and 2 are not identical.

2.1.1.4 Variant Records

According to the Section 6.2.2 RR, a tagfield is introduced in a variant record to control the variants. What happens if the record is free (that is, a tagfield is not declared in the record)? To see this problem, look at the following example:

Example 4: program ab(input,output);
    type free=record
        case integer of
            1:(a:Boolean);
            2:(b:char)
        end;
    var x:free;
    y:char;
    begin
        x.a=false;
        y:=x.b
    end.

At run time, a memory unit is allocated to x, for loading either a boolean value or a character. This can be depicted as:

\[ x \rightarrow \square \]

The execution of the first statement results:

\[ x \rightarrow \text{false} \]

Now, for the second statement there is no information (no tagfield) to stop the extraction of a value from the location of x. To resolve the conflict we can allocate an extra location to load the label of the first field. Therefore, the execution of the first statement can be demonstrated as:

\[ x \rightarrow \text{false} \]

\[ \text{EXTRA} \rightarrow 1 \], where the number 1 is the label of the first field in the record.
To obtain a value from a selector of a field, the extra location must already be loaded with one of the labels of the field. Since 2 is not in EXTRA, the value of the location x.is not of type x.b and the execution of the second statement is illegal.

To make the definition of the language homogeneous to both free and discriminated records (which have a tagfield for each of their case clauses) an extra location is also allocated for the discriminated one. Therefore, when a value is assigned to a tagfield it will also be assigned to this extra location.

From the above discussion it seems that, it would be better if the free records are removed from the language forcing the programmer to control the variants himself (via the tagfield).

2.1.1.5 Sets

The subrange type declaration also creates conflicts here. Let us look at the following example:

Example 5:

```plaintext
type week=(mon,tues,wed,thur,fri,sat,sun);
weekday=mon..fri;
var x:set of weekday;
begin
  if mon in x then...
end.
```

The type of mon is week but the base type of x is weekday, hence they are not the same and the process of type checking halts at stage A. (see the Table of 8.1.4 RR).

The base type of the empty set is not defined. We assert that the base of can be of any scalar or subrange type. Furthermore, once is used in an operation, its type will be fixed to the type of the other operands. With this restriction, for example, the expression:

```
[1,2,3]*[ ]*['a','b']
```

which gives the same result as:


is not a valid PASCAL expression.

2.1.1.6 Subroutine Formal Parameters

When the formal parameter q of a subroutine is either a function or procedure the system of Chapter 5 is designed so that the number and the types of the parameters of q can be determined from its first call at compile time. Welsh [25] has claimed that the error of the following example (taken from Lecarme and Desjardins [30]) cannot reasonably be detected at the compilation stage.

Example 6: procedure p(procedure q);
begin
    q(2,'A')
end;

procedure r(x:Boolean);
begin
    write(x)
end;

By our method, at stage A, it is known, that the number of the parameters of q is 2 and their types are integer and character. Since the number and the type of the parameters of r are not compatible with the ones of q, the program fails to compile at stage B.

At first glance, it seems that this method is suitable, but this is not applicable to the Example 7, below. Since the requirements of f are not defined at stage A, it is not possible to determine those of t at this stage. At stage B it is found that:

- f has 2 parameters of types Boolean and character

and at C it is found that:

- t has 1 parameter of type integer

Now, look at the stage A again, the information obtained from B and C
shows that, since the number and the types of the parameters for \( f \) are not compatible with the ones of \( t \), the call \( p(t) \) was illegal.

**Example 7:**

```plaintext
program ab(input,output);
var b:Boolean;
procedure q(s:integer);
begin
  writeln(s)
end;
procedure h(m:Boolean;n:char);
begin
end;
procedure u(procedure t);
procedure p(procedure f);
begin
  if b=true then
    begin
      b:=false;
      end;
  p(t)
end;
f(true,'a')
p(h);
t(2)
begin
  b:=true;
  end;
u(q)
```

The full specification of the parameters of a subroutine formal parameter can be defined in either of the following cases:

I. The number and the types of the parameters may vary from one call to the other. In this case, it is better to define these requirements at the run time stage.

II. The number and the types of the parameters are fixed. One solution due to Lecarme and Desjardins [30] is to define these requirements in the syntax of the language. For example the headings of \( u \) and \( p \) of the previous example, may be declared

†The stages denoted by numbers are explained in the next section, Part V.
as follows:

```plaintext
procedure u (procedure t (integer))
procedure p(procedure f(Boolean,char))
```

By this assumption the full specification of the parameters of \( t \) or \( f \) can be defined at compile time stage.

N.B. Since we do not wish to change the basic framework of the language, we do not use this solution in our formalisms.

The solution to the PASCAL ambiguities and the requirements of the forward subroutine (i.e. one whose head is declared at a certain point but whose body is not declared until later in the current block) is formalised in both of our systems. These are left for the reader.

N.B. Forward subroutines are not mentioned in the report, but they are described in Section 11.C of the User Manual [5].

2.2 INCONSISTENCIES BETWEEN TWO PASCAL IMPLEMENTATIONS

When a language is not precisely defined, it is natural to expect diversification among different implementations. Even more important, we might not have a uniform interpretation for the requirements of the language on any single implementation.

In this section, some of these inconsistencies are listed below for two PASCAL implementations [31] and [32]\(^\dagger\), which we refer to as compilers A and B respectively. These inconsistencies were found before June 1981.

\(^\dagger\)If the reader is working with different PASCAL implementations, he can test them against one of our formalisms, described in Chapters 3 or 5. For example, he may take the system of Chapter 5 and write programs to search for inconsistencies which might occur in the places where the word "halt" (see Appendix 2) appears.
and may be different in later versions of the systems.

I. In the following declarations:

\[
\begin{align*}
&\text{const...; }\langle\text{identifier}\rangle = \langle\text{constant identifier}\rangle; \\
&\text{type...; }\langle\text{identifier}\rangle = \langle\text{type identifier}\rangle;
\end{align*}
\]

The constant (type) identifier on the right hand side should be declared before the current declaration (this is not mentioned in the report). In the following programs the right hand side identifier \( x \) is not defined elsewhere, but neither compilers detect this error.

\[
\begin{align*}
&\text{program } ab(\text{input, output}); \\
&\text{const } x = x;
\end{align*}
\]

II. Using the word "forward" to declare a forward subroutine. There is no need to treat it as a word delimiter. The compiler A accepts this policy but B does not. For example for the program:

\[
\begin{align*}
&\text{var } \text{forward:integer}; \\
&\text{procedure } f; \text{forward}; \\
&\text{..................}
\end{align*}
\]

The compiler A accepts the identifier forward as a variable but B does not.

III. Let \( \alpha \Theta \beta \) be a binary operation with the operator \( \Theta \), included in a PASCAL expression. If \( \alpha \) or \( \beta \) is an empty set and its type is not defined to be the base type of the second operand then by re-ordering the sub-expression \( \alpha \Theta \beta \) to \( \beta \Theta \alpha \) the new expression might not be a PASCAL one. This policy is carried out by the compiler A. The compiler recognizes:

\[
\begin{align*}
&[\text{'}A', \text{'}B'] \times [ ] \times [1,2,3] \\
&[\text{'}A', \text{'}B'] \times [1,2,3] \times [ ]
\end{align*}
\]

However, a better solution is that the base type of the empty
set is defined to be the type of the other operand. This is what the compiler B accepts. None of the above expressions is a PASCAL one according to the compiler B.

IV. It seems that the type checking for compilers A and B is based on the structural and name equivalence (cf. Sec. 2.1.1), respectively. However, neither compilers use each of these methods exclusively.

Example 8: program ab(input,output);
    type week=(mon,tues,wed,thur,fri,sat,sun);
    procedure f;
    type week=(mon,tues,wed,thur,fri,sat,sun);
    var x:week;
    y:week;
    begin
        x:=y
    end;
end.

program cd(input,output);
    type week=(mon,tues,wed,thur,fri,sat,sun);
    var x:record a:mon...sun end;
    y:record a:week end;
    begin
        x:=y
    end.
end.

Compiler A reports no error for the program ab, that is, the variables x and y are of the same type. But the program cd fails to compile at stage E, leaving the message "types of operands conflict". The compiler B rejects the program ab at stage D, leaving the message "assignment: left and right hand side incompatible", but accepts program cd!

V. To see the behaviour of our compilers on the program of Example 7 the concept of the run time stack is introduced. A stack is working space for the computational (execution) process. At the invocation of a subroutine the identifiers local to the subroutine are loaded into the top of the stack. The stack section which is
allocated for a subroutine identifiers is called \( \text{frame} \). The state of the stack for the corresponding stages in Example 7 can be demonstrated as follows:

- **stage 1**: \( b \leftarrow \text{true} \), \( q \), \( h \), \( u \)
- **stage 2**: \( b \leftarrow \text{true} \), \( q \), \( h \), \( u \), \( t \), \( p \)
- **stage 3**: \( b \leftarrow \text{true} \), \( q \), \( h \), \( u \), \( t \), \( p \), \( f \)
- **stage 4**: \( b \leftarrow \text{false} \), \( q \), \( h \), \( u \), \( t \), \( p \), \( f \), \( f \)

N.B. Upon the invocation of each subroutine the formal parameter of the subroutine is linked to its corresponding actual parameter.

Since \( f \) is linked to \( t \) and \( t \) is also linked to \( q \), for the subroutine call at stage 5 of Example 7 we have:

\[
\begin{align*}
\text{f}(\text{true},'a') & \equiv \text{t}(\text{true},'a') \equiv \text{q}(\text{true},'a')
\end{align*}
\]

but \( \text{q}(\text{true},'a') \) is not a PASCAL subroutine invocation.

None of the compilers detect a compile time error for this example.

To pursue the behaviour of the run time process, the statements writeln ('start-\( a \)') and writeln ('end-\( a \)') are added after the symbol begin and before the symbol end of the block of each procedure \( a \), where \( a \) is the procedure (program) name, respectively. The compiler A does not report any error at all and the execution terminates successfully! The output is:

\[ t \]

The precise behaviour of the run time stack are described in the systems of Chapters 3 and 5.
The compiler B fails to terminate the execution process, but the process does not stop in the right place (i.e. after the number 97 below). The output is:

```
start-ab
start-u
start-p
start-p
start-q
  1
end-q
end-p
start-h
end-h
end-p
start-q
  2
end-q
end-u
end-ab
```

2.3 SYNTAX MODIFICATIONS

The purpose of this section is to introduce a PASCAL subset (PSS) which is formalised in the systems described in subsequent chapters. The PSS is obtained from PASCAL by omitting the following features:

I. The omission of file types. Since the file type is omitted, there is no point in having any parameters in the program heading of a PSS program. Therefore, the corresponding syntax is changed to:

```
<program heading>::= program<identifier>;
```

Although the identifiers input and output do not appear in a PSS program heading, the input and output files are available for the program.
II. Label declarations and goto statements. Since the existence of goto statements may make the sequence of states in a computation difficult to trace, goto statements and label declarations are eliminated from our PSS. To formalise the whole of PASCAL it is better to first transform a program including goto statements to one excluding them, providing the sequence of computations is not changed. This can be done either by obtaining semi-structured programs [6], or changing the goto statements to while statements and introducing auxiliary variables [33]. However, as Knuth has proved ([34], page 25), there are programs which cannot be transferred to fully structured ones, without introducing auxiliary variables or changing the sequencing of a computation.

III. Packed array, record and set. The inclusion of packed arrays, records and sets in the language necessitates their representation in terms of bits. This is not difficult and can be defined (consulting [35] page 314 and [36] pages 29-34) but it was decided not to include them in our PSS in order to decrease the volume of work in this thesis.

IV. With statements. With statements are omitted. However, they can be formalised in a similar fashion to case statements.

V. Standard procedures.

A. File handling procedures. Since the file types are omitted from PSS, the procedures put, get, reset and rewrite (Section 10.1.1 RR) have no application in this language.

B. Dynamic allocation procedures. The following call is eliminated:

\[ \text{new} (p, t_1, \ldots, t_n) \]

where \( p \) is a pointer variable which points to an implicit
variable of the type record with variants and $t_1, \ldots, t_n$ are
tagfield values. The procedure $\text{new}(p)$ is extended to $\text{new}(p_1, \ldots, p_k)$, where $p_i$, $1 \leq i \leq k$ is a pointer variable, and means:
\[
\text{new}(p_1); \text{new}(p_2); \ldots; \text{new}(p_k)
\]
C. Data transfer procedure. The procedures pack and unpack
(Section 10.1.3 RR) are not applicable in PSS.

VI. Standard functions.
A. All the arithmetic functions ($\text{abs}, \text{sqr}, \text{sin}, \text{cos}, \text{exp}, \text{ln}, \text{sqrt}$,
\text{arctan}) of the Section 11.1.1 RR are deleted from our PSS. However, these can be declared by the programmer.
B. The function calls $\text{eof}(f)$ and $\text{eoln}(f)$ in Section 11.1.2 RR
where $f$ is a parameter of type file, are changed to the calls $\text{eof}$ and $\text{eoln}$. The call $\text{eof}$ or $\text{eoln}$ in a program indicates the
end of input file or end of current line of the input file
status.
C. Transfer functions. The functions $\text{trunc}$ and $\text{round}$ are omitted.

VII. Input and output procedures.
A. Because of the omission of the parameter of type file the
procedure calls:
\[
\begin{align*}
\text{read}(f, v_1, v_2, \ldots, v_n), \ & \text{readln}(f, v_1, v_2, \ldots, v_n) \ \text{readln}(f) \\
\text{write}(f, p_1, p_2, \ldots, p_n), \ & \text{writeln}(f, p_1, p_2, \ldots, p_n) \ \text{writeln}(f)
\end{align*}
\]
where $f$ is a parameter of type file, are omitted. These
procedure calls are only applied to the input and output files
by the following forms:
\[
\begin{align*}
\text{read}(v_1, \ldots, v_2), \ & \text{readln}(v_1, v_2, \ldots, v_n), \ \text{readln} \\
\text{write}(p_1, \ldots, p_2), \ & \text{writeln}(p_1, \ldots, p_n), \ \text{writeln}
\end{align*}
\]
where $v_i$, $1 \leq i \leq n$, denote a variable of type character, integer
Finally, several parts of the BNF of the report are changed to obtain two equivalent BNF's suitable for the systems described in Chapters 3 and 5. A different representation of the first BNF is embedded in Appendix 1 and the second BNF is in Appendix 2. These appendices are described in the subsequent chapters.
Chapter 3

A W-grammar Definition of PSS
To define a programming language, one needs to know a good deal more than just the structure (or syntax) of the language, we must also know about the semantics. The semantics of a program can be divided into two groups, those used to determine the static validity of the program and those which specify its meaning. It has become customary, where possible, to invoke validation semantics at compile time and execution semantics at run time. However, there is not a hard line between these two. Some semantic actions might be included in one group or the other. For example, the semantics of type compatibility for standard subroutines is defined at run time for the system used in this chapter, and at compile time for the one in Chapter 5.

W-grammars were devised by van Wijngaarden [1] and used to present formal definitions of the programming languages ALGOL68 [2] and ASPLE [3]. Since ALGOL68 is an extensive language, a quick understanding of the mechanism of W-grammars would be rather difficult. Moreover, [2] is not a complete formal definition of the language. The syntax and most of the compile time semantics are described by this method, but the run time semantics are defined in English sentences.

On the other hand, although the W-grammar definition of ASPLE is much easier to understand, and is complete, the language is so simple and unrealistic as to be impractical, and does not provide a convincing illustration of the power of this method.

Our PSS W-grammar neither has the limitations of ASPLE nor does it suffer from the incompleteness of the ALGOL68 definition. In fact, it satisfies all three requirements of a language definition system; that is it specifies the syntax, the compile time semantics and the run time semantics of PSS.
3.1 THE W-GRAMMAR SYSTEM

3.1.1 Informal Discussion

To illustrate the working of a W-grammar, we introduce a simple programming language. The context free syntax of this language can be formally defined by a system known as Backus Normal Form (BNF), or Backus Naur Form (after [4]).

A preliminary BNF definition of the language,

\[
\begin{align*}
1 & \quad \text{<program> ::= var <main>} \\
2 & \quad \text{<main> ::= <identifier>; <stmt>} \\
3 & \quad \text{<identifier> ::= alb} \\
4 & \quad \text{<stmt> ::= <identifier> := <expression>} \\
5 & \quad \text{<expression> ::= <bool> | not <bool>} \\
6 & \quad \text{<bool> ::= false | true}
\end{align*}
\]

The following are examples of programs generated by the grammar.

\textbf{Example 1:} \quad \text{var a; a := false}

\textbf{Example 2:} \quad \text{var b; a := not true}

Informally we describe the semantics of this language as follows:

\textbf{Compile time semantics:} The identifier on the left hand side of the assignment statement should have already been declared. Notice that, because of this restriction, although the grammar generates the program in Example 2, this is not a legal program.

\textbf{Run time semantics:} The right hand side of the assignment statement will be evaluated and associated with the left hand side identifier.

To have a formal definition of the syntax and the compile time semantics, we extend the BNF (and thus restrict the language generated) as follows:

A second BNF for the language,

\[
\begin{align*}
1 & \quad \text{<program> ::= var <main>} \\
2 & \quad \text{<main> ::= <a symbol>; <a symbol> ::= <expression>} \\
3 & \quad \text{<a symbol> ::= a} \\
4 & \quad \text{<b symbol> ::= b} \\
5 & \quad \text{<expression> ::= <bool> | not <bool>} \\
6 & \quad \text{<bool> ::= false | true}
\end{align*}
\]
This second BNF does not generate the program in Example 2.

Now consider what would happen if rule 3 of the first BNF definition was replaced by the following more realistic rules:

\[
\begin{align*}
\text{20-times} \\
\langle\text{identifier}\rangle & ::= \langle\text{chr}\rangle | \langle\text{chr}\rangle \langle\text{chr}\rangle | \ldots | \langle\text{chr}\rangle \ldots \langle\text{chr}\rangle \\
\langle\text{chr}\rangle & ::= a | b | c | \ldots | z
\end{align*}
\]

i.e. any sequence of up to 20 alphabetic characters.

To incorporate the compile time semantic restrictions of the second BNF definition in this new situation would require the addition of

\[
n = (26 + 26^2 + 26^3 + \ldots + 26^{20})
\]

rules similar to rule 2, and \(n\) rules similar to rule 3 in the above grammar.

Therefore, the extension of the BNF for defining the context sensitive requirements of a reasonably complex language is impractical.

However, it is possible, using the method described below, to take the set of all rules required for defining every program in the language and obtain from them the relatively small set of rules actually required to define a specific program. To do this we design the following Macro statements with the formal parameters \(\text{ID}, \text{EXP}\) and \(\text{BOOL}\).

N.B. The reader should understand that the following statements do not constitute a context sensitive grammar (although they generate a context sensitive language) and they can be treated as model productions. The notation used is a variant of BNF which can be understood by referring to the earlier grammar. It is formally defined in Section 3.1.2 below.

\[
\begin{align*}
1 & \text{ program:} \text{var} , \\
2 & \text{ main: ID symbol ,} \\
3 & \text{ ID symbol} , \\
4 & \text{ ID symbol} , \\
5 & \text{ ID symbol} , \\
6 & \text{ EXP expression} . \\
7 & \text{ a symbol :} \text{a} . \\
8 & \text{ b symbol :} \text{b} . \\
9 & \text{ false expression :false} . \\
10 & \text{ true expression :true} . \\
11 & \text{ not BOOL expression :not} , \\
12 & \text{ BOOL expression} .
\end{align*}
\]
For each particular program, the formal parameters ID, EXP and BOOL are replaced by a proper item of the right hand side of the following rules:

1. ID::a;b.
2. EXP::BOOL·c:'c
3. BOOL::false;true.

Example 3:

```
var a;
a:=not false
```

Replacing each ID, EXP and BOOL by "a", "not false" and "false" respectively in the Macro statements, we obtain the following context free productions:

1. `program :var ,
   main .`
2. `main :a symbol ,
i ,
a symbol ,
i = ,
not false expression .`
3. `a symbol :a .`
4. `not false expression :not ,
   false expression .`
5. `false expression :false .`

The parse tree for Example 3 is therefore:

```
program
  /
var
  /
main
    /
  a symbol ;
    /
    a symbol
      /
a
      /
not false expression
        /
not
        /
false

FIGURE 3.1`
The method outlined above can also be used to define the run time semantics of the language. However, since our aim was to describe the operation of the system, we omit a formal description of the run time semantics of this simple language. We will of course explain all semantic aspects for our PASCAL subset in detail later.

3.1.2 Basic Notation and Definitions

A W-grammar is a two level grammar containing two finite sets of rules, the "Metaproductions" which are the production rules of the first context free grammar level, and the "Hyper rules" which are the model productions. By the use of the metaproductions in hyper rules a set of production rules of the second context free grammar level will be obtained. This set is capable of describing the syntax, the compile time semantics and the run time semantics of the language. In order to describe the rules of the grammar, we need some terminology.

3.1.2.1 Metanotions, Protonotions and Hypernotions

To define the rules of the W-grammar for a language, we use certain characters classified as follows:

I. Small characters:
"a", "b", "c", ..., "z".

II. Large characters:

III. Other characters:
"+", ",", ",", ",", ..., ",", ",", ",", "\".

A "Metanotion" is a word consisting of large characters.

e.g. NAME, NAME1, ALPHA, LETTERSETY, EMPTY

† A word is a string of one or more characters.
A "Protonotion" is:

I. A string of small characters
e.g. letter a letter b, a symbol, symbol a

II. A sequence of words, each of them formed from small characters preceded (or followed) by a large character
e.g. letter a letter B, A symbol, symbol A

III. A string of underlined characters
e.g. begin, A,[, ...

IV. The empty string.

The "Hypernotion" is either a string of small and (or) large characters, or a string of underlined characters:
e.g. letter ALPHA LETTERSETY, NAME identifier, A symbol, a symbol.

A, begin, +.

3.1.2.2 Metaproductions, Hyper Rules and Production Rules

A "Metaproduction" is a context free production for the first level grammar with the following characters:

I. The non-terminal symbol is a metanotion.

II. A terminal symbol is a protonotion (excluding the underlined string).

III. A double colon separates the left and right hand sides of the production rule.

IV. Different alternatives in a rule are separated by a semicolon.

V. Different non-terminals in an alternative are separated by spaces.

VI. The rule is terminated by a dot.

e.g. NAME::=letter A LETTERSETY.
    LETTERSETY::=LETTERS;EMPTY.
    LETTERS::=LETTER;LETTERS LETTER.

†In BNF style this metaproduction is written as:
    <letters>::=<letter>|<letters><letter>
A "Hyper rule" is a model production for the second level context free grammar with the following properties:

I. The rule consists of hypernotions.

II. A colon is used to separate the left and right hand sides of the rule.

III. A semicolon is used to separate two alternatives in the same rule.

IV. Different hypernotions in the same alternative are separated by a comma.

V. A dot terminates the rules.

e.g. † Hsy96 the syntax of OBJECT be checked: const,
    where OBJECT is constant;
    type,
    where OBJECT is type;
    var,
    where OBJECT is variable.

FIGURE 3.2

A "Production (rule)" is a context free production, obtained by expanding metanotions in a hyper rule. The set of productions constitute the second level context free grammar, and have the following characteristics:

I. The non-terminals are protonotions (excluding the underlined strings).

II. A terminal symbol is either the empty string or a string of underlined characters.

III. A colon separates the left and right hand sides of the rule.

IV. Two alternatives in the same production are separated by a semicolon.

†In Appendix 1, the hyper rules and metaproductions are grouped in twelve different sections each under a special name. We refer to a hyper rule or a metaproduction by the form:

Hmn or Mn

respectively, where m is the section name and n is the rule number.
V. Different protonotions (terminal or non-terminal symbols) in the same alternative are separated by comma.

VI. The production terminates with a dot.

To obtain a production from a hyper rule, we impose the following restrictions:

A. Each metanotion in a hyper rule is replaced by the corresponding terminal string, generated by the metaproduction.

B. If the same metanotion occurs two or more times, then all occurrences must be replaced by the same metaproduction. This process is called the Uniform Replacement Rule (URR).

In the set of hyper rules, the use of the same metanotion with different subscripts in any rule, shows that although they use the same metaproduction, they possibly generate different productions. Therefore, URR only will be applied to the same metaproduction with identical subscripts.

Example 3: When the metanotion OBJECT in the hyper rule of Figure 3.2 is replaced by the protonotion "constant", applying M124, then the following production will be obtained:

the syntax of constant be checked :

const ,
where constant is constant ;
type ,
where constant is type ;
var ,
where constant is variable.

FIGURE 3.3

The representation of the terminal symbols in a programming language depends on its implementation. However the underlined terminal strings in this grammar are the same as the underlined symbols and the characters of the PASCAL language [5].
3.2 DEFINITION OF PSS

3.2.1 General Hyper Rules

Specification of the compilation time and the run time semantic requirements of the language are based on three hyper rules as follows:

\[ t_{\text{Hpr1}} \]
where \( \text{CHARSETY} \) is \( \text{CHARSETY} : \text{true} \)

\[ t_{\text{Hpr3}} \]
where \( \text{CHARS} \) is not in \( \text{CHARSETY} \):

- the string consist of \( \text{ONES} \) characters \( \text{CHARS} \),
- \( \text{ONES} \) of \( \text{CHARS} \) can not be in \( \text{CHARSETY} \);
- the string consist of \( \text{ONES} \) characters \( \text{CHARS} \),
- \( \text{CHARSETY} \) consist of less than \( \text{ONES} \) characters.

\[ t_{\text{Hpr7}} \]
where \( \text{CHARSETY} \) \( \text{CHARSETY1} \) \( \text{CHARSETY2} \) contains \( \text{CHARSETY1} : \text{true} \).

A production of the hyper rule \( t_{\text{Hpr1}} \) checks whether two strings are the same. The mechanism of the process can be clarified by an example.

Example 4: Assume there is a derivation terminated by the production:

where \( \text{abc} \) is \( \text{abc} \)

By using the metanotion \( t_{\text{M27}} \), we have the derivation:

\[ \text{CHARSETY} \rightarrow \text{CHARS} \rightarrow \text{CHAR} \text{CHARS} \rightarrow \text{a} \text{CHARS} \rightarrow \ldots \rightarrow \text{abc} \]

and by substitution "abc" for \( \text{CHARSETY} \) in \( t_{\text{Hpr1}} \), applying URR, we obtain:

where \( \text{abc} \) is \( \text{abc} : \text{true} \).

The hyper rule \( t_{\text{Hpr2}} \):

\[ \text{true} : \text{EMPTY} . \]

and the metaproduction \( t_{\text{M49}} \):

\[ \text{EMPTY} : : : . \]

shows that the empty string can be produced from true. The production tree is:

\[ \uparrow t_{\text{Hpr1}} \]
where \( \text{abc} \) is \( \text{abc} \)

\[ \uparrow t_{\text{Hpr2}} \]
true

\[ \downarrow \text{EMPTY} \]

A production of the hyper rule \( t_{\text{Hpr3}} \) checks that a string is not in another string.

\[ t_{\text{Hpr}} \text{ see P252, M27 see P146, M49 see P147 etc.} \]
Example 5: Suppose there is a derivation terminating in the non-terminal production:

where ab is not in acb

Using the related hyper rules and metaproductions the production tree is shown in Figure 3.4.

3.2.2 Modelling Specific Aspects of PSS

Most compilers construct a symbol table containing all the identifiers and their modes. This is used to check various conditions imposed at compile time. The execution of the statements and other requirements at run time necessitates a tool for space allocation. This can be handled by stack manipulation. These concepts provide the basis for our PSS W-grammar, details of which are now considered.

3.2.2.1 The Program

A valid PSS program and its meaning are defined as a program which has a production tree whose terminals (from left to right) form the program, the value of the input file before execution and the value of the output file after execution. If the program is not valid, or causes an invalid state during execution, the process of constructing the tree will fail. The investigation of error conditions is beyond the scope of our work.

Starting from Hde1, each PSS program is treated as a procedure nested in an outer range at the head of which the standard types and subroutines are declared and a call of the program is the only statement in its main compound statement. This is represented in the following scheme:

The standard type declarations;
The standard subroutine declarations;
The program;
begin
  Program call
end.

FIGURE 3.5
In order to save space several notations are used for the examples. The notations $\bar{0}$, $\bar{1}$, $\bar{2}$, ... are specified for the protonotions "zero", "one" and "one one", ..., respectively. Other notations for protonotions will be explained as they are encountered.
The standard type declarations in Figure 3.5 are defined by Hde2. The standard subroutine declarations are divided into two categories:

A. Computational subroutines which can be invoked (called) by the programmer (such as odd, sin, cos, etc.). We have taken the odd function as an example and define its requirements in hyper rule Hde3. Others can be defined similarly.

B. Subroutines which interact with the run time environment (such as eof, write, read, etc.). These are defined individually by various hyper rules in the model.

Since the program acts like a PSS subroutine we first discuss subroutine declarations and subroutine calls.

3.2.2.2 Subroutine Declaration (Hde70)

In order to consider the various parts of a subroutine some information about the identifiers, declared in outer ranges is required. This information can be collected in "symbol table" or "local tables" TABLES†. To see how the various tables are inter-related consider the following example.

Example 6: 

```
program ab;
var i:real;
function Bisect......
............... 
............... 
```

The state of the symbol table, before the procedure is declared, is represented schematically in Figure 3.6.

†A production generated by metanotion α is represented by $\bar{α}$. 
Ti, i=1,2,3 denotes the i-th table of the symbol table. To avoid the need for calling the program as a procedure, the necessary information concerning the program (except its name) is stored in the first table. This is shown by the last row of Table 1. The second table consists of the formal parameter of the odd function. The parameter has the pseudo-name x.

The identifiers local to the subroutine and their corresponding information is defined by TABLE. This table in turn will be joined to TABLES to form a new symbol table, i.e.:

\[
\text{TABLES}_1 = \text{TABLES} \cup \text{TABLE}
\]

The subroutine's local table is of the form:

\[
\text{the ONE}^{\text{st}} \text{ LOCSETY slink to ONESETY table .}
\]

The protonotions ONE and ONESETY specify the "table number" and the "linkage" of the table. The internal representation of local identifiers and their characteristics are defined by the "locations" LOCSETY. The location of subroutine is inserted in the current table (not the subroutine's
table) and \textbf{TABLE} is linked to it by \textbf{ONESETY}. This location is defined as:

\begin{verbatim}
loc \textbf{NAME} be \textbf{FUNPROC} of \textbf{PARNUMETY} type \textbf{TYPE} and \textbf{ONES} as its local table \textbf{USAGETY} end
\end{verbatim}

where \textbf{NAME}, \textbf{FUNPROC}, \textbf{PARNUMETY}, \textbf{TYPE} and \textbf{ONES} define the name, the kind of subprogram (function or procedure), the number of parameters, the type of the subroutine and its table number respectively. According to whether the subroutine block is defined currently or at a later point, the protonotion \textbf{USAGETY} is "body is ahead" or the empty string respectively.

\textbf{Example 7:} The program of Example 6 can be completed as:

\begin{verbatim}
program ab ;
var i:real ;
function Bisect(function f:real ;a,b:real ;var z:real):integer;
var m :real ;
procedure abs(n :real);
begin
    ...
end;
begin
    ...
end;
begin
    i:=-1 ;
i:=Bisect(odd,i+1,i,i) .
end.
\end{verbatim}

This requires the construction of the tables illustrated in Figure 3.7.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example7}
\caption{FIGURE 3.7}
\end{figure}
The corresponding protonotion is:

\[
\text{TABLES} = \text{TABLE:1} \ \text{TABLE:2} \ldots \text{TABLE:4}
\]

where:

\[
\begin{align*}
\text{TABLE1} &= \text{the } 1\text{th \ LOC1 \ LOC2} \ldots \text{LOC6 slink to table}, \\
\text{TABLE2} &= \text{the } 2\text{th \ LOC7 slink to 1 table}, \\
\text{TABLE3} &= \text{the } 3\text{th \ LOC8 \ LOC9 slink to } 1 \text{ table}, \\
\text{TABLE4} &= \text{the } 4\text{th \ LOC10 \ LOC11} \ldots \text{LOC15 slink to } 3 \text{ table},
\end{align*}
\]

and

\[
\text{LOC9} = \text{loc Bisect be function of 4 parameters type integer in } 1 \text{ table and 3 as its local table end}
\]

Other locations are produced from their rules in a similar fashion.

The compound statement, forming the body of the subroutine, corresponds to the internal statements:

\[
\text{Name and its ONES table STMT STMT } \ldots \text{STMT exit .}
\]

Since this protonotion is part of the internal version of the program, the unique string "\text{Name and its ONES table}" and immediate "exit" determine the beginning and the end of the corresponding compound statement.

3.2.2.3 Subroutine Invocation (Hru2)

In order to appreciate the execution of a PSS program, it is required to have a broad understanding of the "stack" \text{STACK}. The protonotion \text{STACK} consists of a sequence of frames \text{FRAME} as:

\[
\text{STACK=FRAME}_1 \ldots \text{FRAME}_n
\]

A frame holds local information required for a subroutine call.

\text{Example 8: The state of the stack after the execution of:}

\[
i := -1
\]

in Example 7 can be demonstrated pictorially in Figure 3.8.

\footnote{To save space, the internal representation of a special identifier is presented by its over lined form. e.g. the metaproduction \text{M116} generates:}

\[
\text{Bisect = letter B letter i letter s letter e letter c letter t}
\]
Note that the types from the first table do not appear in the first frame (constant identifiers are replaced by their values, and type definitions can be accessed via pointers).

The invocation of a subroutine changes STACK to STACK1 by adjoining an extra frame:

\[
\text{FRAME} = \text{FRAME} \quad n+1
\]

on the top of the stack.

**Example 9:** The contents of the stack upon the invocation of procedure Bisect in the Example 7 (before the execution of its block) are shown by Figure 3.9.
The symbol Bisect is the representation of frame's head in this picture.

The protonotion FRAME is of the form:

the ONESt frame from ONESt table HEADETY and FLOCESTY
link to ONESETY frame

where ONES, ONESt and ONESETY define the "frame number", the corresponding table number of the subroutine and the "static link" respectively. Each "frame location" FLOC of FLOCESTY introduces one of the parameters, nested subroutine names and variables. The frame location is defined formally as:

loc NAME refers to NBOXESETY end
where \texttt{NB}O\texttt{X}E\texttt{S}ET\texttt{Y} is the run time "memory units" allocated for the corresponding identifier.

The internal representation of the subroutine call with \(k\) parameters is:

\[
\text{SUBCALL=NAME with COMMA \ldots COMMA} \quad 1 \quad 2 \quad \ldots \quad k
\]

where \texttt{COMMA}, \(1 \leq i \leq k\), represents the \(i\)th actual parameter, which may be a value, a variable or a subroutine (procedure or function) name parameter. Corresponding hyper rules define an internal constant value \texttt{CON}ST, an address \texttt{ADD}RESS or a subroutine name \texttt{DNAME}.

Each item in turn is located in the memory unit(s) of the corresponding formal parameter in the stack or heap (as explained below). At this stage the actual parameters (if any) have been loaded and the internal statements of the subroutine can be executed.

The stack as working space is not enough to define the execution semantics of the language. We also need heap \texttt{HEAP} as a separate working space. To see this let us look at the following example.

\textbf{Example 10:}

```pascal
program ab;
type t= array[1..10] of integer;
var x : t;
procedure f;
begin
  new(x);
  begin
    f;
  end;
end.
```

If at run time we design the system so that ten memory units are created on the top frame in the stack, then on the exit from the block of f the top frame will vanish and the legal statement: \(x[3] := -5\) cannot be

\(\texttt{DNAME} is either \texttt{NAME} for the standard subroutine name (except odd) or \texttt{NAME} for subroutines declared in the program.\)
executed. To overcome this conflict ten memory units are created on a heap and the single unit of x points to the first unit corresponding to xt[l]. The situation of the stack and heap after the execution of the assignment statements are represented by Figure 3.10.

Since there are some special cases in the subroutine declaration and the subroutine call we note the following comments:

Note 1. After all subroutine declarations the symbol table is searched for the absence of:

\underline{USAGETY}=body ahead .

This process is defined by Hde82 and means that any forward subroutine has a block declared later in the current block.

Note 2. Since the locations of the symbol table corresponding to the constant and type declarations are not transferred to the stack, this table is needed for run time semantic requirements.

Note 3. So far we have described the "ordinary subroutine call" which are declared in the program. However there are two more cases explained below:
Note 3.1. Call of a subroutine passed as a formal parameter.

The internal representation is the same as in the normal case, that is:

\[
\text{FUNPROC NAME with actuals COMMASETY call },
\]

where NAME corresponds to the subroutine name (formal parameter). Starting from Hru15 (defining the run time semantics), Hru16 presents the corresponding frame location given by:

\[
\text{loc NAME refers the ONES3th DNAME ht end}.
\]

Rule Hru15 and several other hyper rules introduce either an ordinary or standard subroutine call. Their calling mechanism can be understood from the following algorithm:

1. Assume \( i := 0 \), \( \overline{\text{DNAME}} = \overline{\text{DNAME}}_i \)

2. If \( \overline{\text{DNAME}} \) represents an ordinary subroutine name then goto Step 3, otherwise if \( \overline{\text{DNAME}}_i \) is the internal representation of a formal parameter, then goto Step 4, otherwise if:

\[
\overline{\text{DNAME}} = \#\text{NAME}_i
\]

i.e. \( \text{NAME}_i \) corresponds to a standard procedure or function identifier then goto Step 5.

3. Use Hru4 to define the semantic actions for:

\[
\text{FUNPROC DNAME with actuals COMMASETY call } \overline{\text{DNAME}}_i
\]

and then stop.

4. Find the frame location of \( \overline{\text{DNAME}}_i \), the contents of the corresponding memory unit being specified by the protonotion \( \overline{\text{DNAME}}_i \). Update \( i := i + 1 \) and goto Step 2.

5. Use Hru20 to define the requirements of the standard subroutine call:
The determination of the number of formal parameters and the type checking between them and their corresponding actual parameters in this call may defined as run time semantic requirements. However, these can be defined at compile time at the expense of having more complex hyper rules.

**Note 3.2. Standard subroutine call.**

The internal form of this call is:

\[ \text{FUNPROC } \#\text{NAME with actuals } \text{COMMASETY call.} \]

The symbol \# is appended to the front of the internal subroutine name \text{NAME}, to show there is no corresponding location in the symbol table or frame location.

### 3.2.2.4 Constant Declarations

Rules Hde9-Hde12 and other related hyper rules define a location in the current local table for a declared constant. To illustrate the process we explain one of the several cases of the constant declarations. The others can be studied directly from the formal definitions. If the 'value part' of a constant declaration is an identifier which is also used in enumerated type then the corresponding location is:

\[ \text{loc } \text{NAME be constant of konst } \text{NAMEI in } \text{ONES table tsnok end} \]

where \text{NAME} and \text{NAMEI} are associated with the constant identifier and its value respectively. The protonotation \text{ONES} represents the number of the table containing a location of a scalar type including \text{NAMEI}.

**Example 11:**

```plaintext
program ab;
procedure f;
const x=false;

.............
.............
.............
```

The symbol table for this example is shown schematically in Figure 3.11.

The protonotion representing the first location of the fourth table is:

\[ \text{loc } x \text{ be constant of konst false in } T \text{ table tsnok end.} \]

The way in which we represent a scalar identifier avoids the need to look for this identifier (or its successor, predecessor, etc.) in various tables. For example, to evaluate succ(x) at run time we find the value true directly from the Ones=1 table.

3.2.2.5 Type Declarations

A type declaration is associated with a location in the current table. This location is the protonotion:

\[ \text{loc NAME be TYPE end} \]

where NAME and TYPE represent the type identifier and its type definition.
In this section we describe two type definitions, the array type and the record type. The others are more straightforward and can be followed from their hyper rules.

3.2.2.5.1 The Array Type

To see the idea behind the representation of a type declaration in this case we consider the example below:

Example 12:  
program ab;
  type t=array[Boolean,(u,v)] of real

Figure 3.12 represents the symbol table. The first location of the current table corresponds to the type declaration. Any information related to the first index type is obtained via its linkage from the first table. For the second index type a local table (the fourth one) involving a single location, which contains any necessary information for the index, is appended to the symbol table. Therefore, if the index type is not an identifier it is treated as a block nested within the block in which the array type is declared. The idea of the previous example is used through several hyper rules starting from Hde32 to define a corresponding location for a k dimensional array type as:

$$\text{loc NAME be type of array defined by } k \text{ dimensional INDEX}_1 \ldots \text{INDEX}_k \text{ of TYPE needs ONES boxes end}$$

Protonotion \( \text{TYPE} \) corresponds to the component type and \( \text{ONES} \) defines the run time memory units required by a variable of the array type. Protonotion \( \text{INDEX}_i \), \( 1 \leq i \leq k \), is associated with the \( i \)-th index type and are of the following two kinds:

A. The index type is a type identifier, then:

$$\text{INDEX}_i=\text{left NAME}_i \text{ in ONES}_j \text{ table right}$$

states that the location of the type identifier is in the table number \( \text{ONES}_j \).
B. The index type is declared explicitly, then:

\[ \text{INDEX}_i = \text{left in ONES}_j \text{ table right} \]

with the extra local table:

\[ \text{TABLE} = \text{the ONES}_j \text{th loc be type of scalar defined by SCALAR needs } j \text{ boxes } \]
\[ \text{end} \]
\[ \text{slink to ONES}_1 \text{ table} \]

where SCALAR is the internal representation of index type and ONES\(_1\) is the linkage to the table in which the location of array type is stored.

**Example 13:** The protonotion corresponding to Figure 3.12 is:

\[ \text{TABLES} = \text{TABLE}_1 \text{ TABLE}_2 \text{ TABLE}_3 \text{ TABLE}_4 \]

where TABLE\(_1\) and TABLE\(_2\) are the standard table and the table of identifier local to the odd function respectively and:

\[ \text{TABLE}_3 = \text{the } 3\text{th loc } \ell \text{ be type of array defined by } 2 \text{ dimensional left Boolean in 1 table right left in 4 table right of real in 1 table needs 4 boxes } \]
\[ \text{end} \]
\[ \text{slink to 1 table} \]

\[ \text{TABLE}_4 = \text{the } 4\text{th loc be type of scalar defined by left } \nu \text{ right left } \nu \text{ right needs } 1 \text{ boxes } \]
\[ \text{end} \]
\[ \text{slink to 3 table} \]

3.2.2.5.2 The Record Type

In this section, we concentrate on the variant part of a record. To illustrate the definition we introduce the following example:

**Example 14:**

```fortran
program ab;
type r=record
case s:integer of
-1,2:(x,y:real);
5,7:(z:Boolean)
end;

........................
........................
........................
```

```
Following the discussion in 2.1.1.4 about the need for an extra-tag field, we present the figures:

<table>
<thead>
<tr>
<th>identifier</th>
<th>type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for a selector and the extra tag field respectively. The symbol table is shown in Figure 3.13. Starting from Hde49, the internal representation of the variant part (which is a part of a record and this in turn is a part of a location) is:

```
VARPART=case TAGFIELD of VARIANT_1 VARIANT_2...VARIANT_n .
```

where either:

```
TAGFIELD=selection NAME of TYPE with offset UINT endsel extra TYPE with UINT endext
```

for a discriminated record with NAME, TYPE and UINT are associated with the tagfield identifier, its type and offset respectively, or:

```
TAGFIELD=extra TYPE with UINT endext
```

for free record. Each VARIANT_i, i=1..n, is of the form:

```
variant label KONSTANT_1...KONSTANT_p endlab FIELDLIST endvar
```

where KONSTANT_i, i=1..p, associates with the i-th label and FIELDLIST represents the corresponding field list.

**Example 15:** The protonotion corresponding to the location of the third table in the previous example is shown in Figure 3.14.

### 3.2.2.6 Expressions

A PSS expression is a combination of constants, variables, function calls or sets separated by operators. The constant values defined by several hyper rules some of which are discussed in Section 3.2.2.4. Here, we describe some of the major points of the other constituents of an
FIGURE 3.13
loc \( t \) be type of record defined by

\[\text{record}\]

\[\text{case}\]

\[\text{selection } s \text{ of integer in } 1 \text{ table with offset } 0\]

\[\text{endsel}\]

\[\text{extra integer in } 1 \text{ table with offset } 1\]

\[\text{endext}\]

\[\text{of}\]

\[\text{variant}\]

\[\text{label konst } \text{uminus}_1 \text{ tsnok}\]

\[\text{konst uplus}_2 \text{ tsnok}\]

\[\text{endlab}\]

\[\text{selection } x \text{ of real in } 1 \text{ table with offset } 2\]

\[\text{endsel}\]

\[\text{selection } y \text{ of real in } 1 \text{ table with offset } 3\]

\[\text{endsel}\]

\[\text{endvar}\]

\[\text{variant}\]

\[\text{label konst } \text{uplus}_5 \text{ tsnok}\]

\[\text{konst uplus}_7 \text{ tsnok}\]

\[\text{endlab}\]

\[\text{selection } z \text{ of Boolean in } 1 \text{ table with offset } 2\]

\[\text{endsel}\]

\[\text{endvar}\]

\[\text{needs 4 boxes}\]

end

end

\[\text{FIGURE 3.14}\]
expression. The hyper rules associated with the run time semantics of a variable introduce a corresponding internal type TYPE from the symbol table and an address ADDRESS from the stack. This address determines the first memory unit of the variable. To make the point clear let us examine the following example:

Example 16:  

```plaintext
program ab;
  type count=(one,two,three);
  r=record
    num:count;
    case s:integer of
      1,-2: (x,y:real);
      5,7: (z:Boolean)
    end;
  var a:char;
  b:r;
  c:real;
begin
  a:='A';
  b.s:=-2;
  c:=b.s;
end.
```

The symbol table and the run time stack are shown by the scheme set out in Figure 3.15. The corresponding addresses are listed below:

The addresses of the variables a and c are represented as:

(1) and (7)

respectively. The addresses of the tagfield b.s at the left and the right hand sides of the second and the last statements are shown as:

(3,tagfield) and (3)

respectively. Notice the differences here, the phrase "tagfield" shows that not only the value -2 in statement b.s=-2 will be associated with memory unit number 3, but also with the memory unit number 4 (cf. 2.1.1.4). The protonotion ADDRESS, defined by Hex50, is:

```
stack 3th tagfield box
```

It is the protonotion LEFTETY in relevant hyper rules which makes the difference between a left and right hand side variable in an assignment.
FIGURE 3.15

extra memory unit for the extra tag field
statement. For left hand side:

\text{LEFTETY}=\text{in left side}

and for right hand side the protonotion \text{LEFTETY} is the empty string.

If the variable is of a referenced one then it has a single run time memory unit. This can be used either as a pointer to heap \text{HEAP} or a unit to allocate the constant \text{nile}.

Example 17: \textit{program ab;}
\begin{verbatim}
type count=...;	rx=...;
var x,y,z:t;r
begin
x=nil;
new(y,z)
end.
\end{verbatim}

The type declarations \text{count} and \text{r} are the same as the ones in the previous example. The state of the symbol table, the stack and the heap after the execution of the routine 'new' is presented in Figure 3.16. The protonotions representing the heap, memory units of \text{x}, \text{y} and \text{z} are:

\text{HEAP}=\text{the Ith undefined ht the 2th undefined ht...the 10th undefined ht}
the Ith konst nil tsnok ht \hspace{1cm} \text{memory unit of x}
the 2th heap Ith box ht \hspace{1cm} \text{memory unit of y}
the 3th heap 6th box ht \hspace{1cm} \text{memory unit of z}.

The requirements of a function call have already been described in section 3.2.2.3. The definition of set is straightforward and can be followed from various hyper rules, starting from Hst43 and Hext59.

3.2.2.7 Statements

Since the assignment statement changes the state of the stack and the heap at run time, it is basic to our PSS run time specification. Here, we describe some of its requirements. As noted above procedure statement was explained in Section 3.2.2.3. Others are left for the reader to follow from the corresponding hyper rules. The run time semantic definitions
FIGURE 3.16
(Hex4) provide an address ADDRESS and an internal type for the left hand side variable. Also, it obtains an address ADDRESS1 (for the structure variable), constant KONSTANT or string STRING for the right hand side expression. The value(s) obtained from ADDRESS1, the constant or the string will be added to stack or heap at the address ADDRESS.

Example 18:

```plaintext
program ab;
  var x: real;
  begin
    x := -2
    end.
```

The state of the stack before and after the execution of the assignment statement are illustrated in Figure 3.17. The protonotions corresponding to these states are:

- STACK1 = the Ith frame from 1 table...link to frame the 2th frame from 3 table and loc X refers to the Ith undefined ht end link to 1 frame

- STACK2 = the Ith frame from 1 table...link to frame the 2th frame from 3 table and loc X refers to the Ith uminus X E uminus O ht en link to 1 frame.

The left hand side address and the value of the right hand side expression are represented as:

- ADDRESS = stack Ith box
- CONSTANT = konst uminus 2 tskon.

3.2.2.8 Input and Output

The protonotions INFILE and INFILE1 from Hdel define the state of the input file before and after processing its first character (if any). This can be considered as a buffer by which the existence of the first character can be tested. The situation in this stage is illustrated as:

```
line counter

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
```

end-file checker
We assume the input file is subdivided into lines. The position of the first character in the current line is indicated by the line counter. This will be tested against the (implementation-dependent) maximum length line (INLINELEN) to check the end of line situation. The end-file checker is either the string "one" or "zero" depending on whether the file is empty or not. Hence, "one" signals the end of file. The state of the input file after reading the numbers 23.2 (character by character) is demonstrated as:

```
4
ab*5...
0
```

and the corresponding protonotion is:

```
INFILE=char a char b char + char 5...end of zero file
4 characters in a line.
```

Hyper rules Hfi6 and Hfill are designed to read a character and a number respectively. The structure of the output file is similar to the one of the input file. The reader is referred to the rules Hfi32, Hfi34, Hfi35 or Hfi38 for appending a string, a character, a boolean value or a number respectively.
CHAPTER 4

SOME CONCEPTS FROM GRAPH THEORY AND

COMPUTABILITY THEORY
The computational model which is introduced in the next chapter employs a list as a working space for defining the requirements of a programming language. The items included in the list may be inter-related in some way. This necessitates a brief study of some elementary concepts from graph theory. Since our goal is to present a machine-independent model, we use Markov normal algorithms [12] to define the basic functions such as addition, subtraction, etc., instead of producing assembly codes for these operations.

The chapter is concluded with a discussion of the inter-relationship between effectively computable functions, computer programs and Markov algorithms.

4.1 LISTS

In this section we extend the concept of an ordered set [13] to define lists. Several list operations are also defined for use in the next chapter.

**Definition**: Given a finite set $L$ together with a total order relation $R$, then the pseudo-list $L_R$ is the set $L$ ordered by $R$. That is:

$$\forall x, y \in L_R \text{ and } x \neq y \text{ then either } x R y \text{ or } y R x.$$

Therefore:

$$L_R = (o_1, o_2, \ldots, o_n) \text{ for some } n \in \mathbb{N}$$

where $o_i \ R \ o_j \ 1 \leq i < j \leq n$.

**Definition**: The "left-augmentation" and "right-augmentation" for the pseudo-list:

$$L_R = (o_1, o_2, \ldots, o_{n-1}, o_n)$$

are defined as:

$$\text{augl}(L_R, x) : L_R \rightarrow (x, o_1, o_2, \ldots, o_{n-1}, o_n)$$

and

$$\text{augr}(L_R, x) : L_R \rightarrow (o_1, o_2, \ldots, o_{n-1}, o_n, x)$$

respectively.

**Definition**: We define the concept of the list as follows:

1. Each pseudo-list is a list.
II. If $L_R$ is a list, then in:

$\text{augl}(L_R, x) : L_R \leftrightarrow M_R$

$\text{augr}(L_R, x) : L_R \leftrightarrow N_R$

$M_R$ and $N_R$ are lists.

The concept of augmentation is generalized to insert a finite set to a list.

**Example 1:** Given list $L_R = (x, x, y, z)$

then:

$\text{augl}(L_R, \{a, b, c\}) : L_R \leftrightarrow (a, b, c, x, x, y, z)$

$\text{augr}(L_R, \{a, b, c\}) : L_R \leftrightarrow (x, x, y, z, a, b, c)$

i.e.

$\text{augl}(L_R, \{a, b, c\}) = \text{augl}(L_R, c) ;$

$\text{augl}(L_R, b) ;$

$\text{augl}(L_R, a) ;$

and

$\text{augr}(L_R, \{a, b, c\}) = \text{augr}(L_R, a) ;$

$\text{augr}(L_R, b) ;$

$\text{augr}(L_R, c) ;$

N.B. The character ";' is used to separate two operations.

**Definition:** If $L$ and $M$ are the sets of items included in the lists $L_R$ and $M_R$ respectively, then:

$x \in L_R \equiv x \in L$

$L_R \subseteq M_R \Rightarrow L \subseteq M$

$L_R \cup M_R \equiv L \cup M$

$L_R \cap M_R \equiv L \cap M$

N.B. Hereafter when the binary relation $R$ is clear from the context, we denote the list $L_R$ by $L$.

**Definition:** Given an identifier $a$. To (re) assign a list $(o_1, o_2, \ldots, o_{n-1}, o_n)$ to $a$ we apply the operation $\text{is}$ as:

$a \text{ is } (o_1, o_2, \ldots, o_{n-1}, o_n)$

**Example 2:** The initialization of the list $(x, y, z)$ to the identifier $\text{STORE}$ is defined by:
Note: We differentiate between the symbols "is" and "=" so that in:

$$\alpha = (o_1, o_2, \ldots, o_{n-1}, o_n)$$

\(\alpha\) is the same as the list \((o_1, o_2, \ldots, o_{n-1}, o_n)\). But in:

$$\alpha \text{ is } (o_1, o_2, \ldots, o_{n-1}, o_n)$$

\(\alpha\) is an identifier as the name of the list. The operation is leads to the further extension of the previous definitions as follows:

Definition: Given the lists:

- \(L\) is \((o_1, o_2, \ldots, o_n)\)
- \(M\) is \((p_1, p_2, \ldots, p_n)\)

then:

- \(\text{augl}(L, x) : L \text{ is } (o_1, o_2, \ldots, o_n) \Rightarrow L \text{ is } (x, o_1, o_2, \ldots, o_n)\)
- \(\text{augr}(L, x) : L \text{ is } (o_1, o_2, \ldots, o_n) \Rightarrow L \text{ is } (o_1, o_2, \ldots, o_n, x)\)

Similarly, these can be generalized to augment a finite set to the list.

Details of this are omitted.

Also:

- \(x \in L \equiv x \in (o_1, \ldots, o_n)\)
- \(L \subseteq M \equiv (o_1, \ldots, o_n) \subseteq (p_1, \ldots, p_n)\)
- \(L \cup M \equiv (o_1, \ldots, o_n) \cup (p_1, \ldots, p_n)\)
- \(L \cap M \equiv (o_1, \ldots, o_n) \cap (p_1, \ldots, p_n)\)

We may wish to distinguish between an element and its position in a list, this leads to the following definition:

Definition: Given the list \(L\) is \((o_1, \ldots, o_n)\) and the symbol \(\$_i \uparrow\), \(i \in \mathbb{N}\), then the position specifier "of" is defined as:

$$\$_i \text{ of } L = \begin{cases} \text{the } i\text{-th position of the list } L, \text{ where } 1 \leq i \leq n \\ \emptyset, \text{ where } i > n \end{cases}$$

Moreover, the last position of the list is denoted by \(\$ \text{ of } L\), i.e. \(\$_n \text{ of } L = \$ \text{ of } L\)

N.B. The symbol \(\emptyset\) in this report is used to represent the undefined value.

\(\uparrow\) The symbol \(\$\) is denoted by \(\$\) in the Appendix 2.
Definition: Given the list \( L = (o_1, \ldots, o_n) \) where \( n \in \mathbb{N} \) then the operation "ordinal" is defined as:

\[
\text{ordinal}(\$ i \text{ of } L) = \begin{cases} 
i & \text{where } 1 \leq i \leq n \\
\Omega & \text{where } i > n
\end{cases}
\]

Definition: Given a list \( L = (o_1, \ldots, o_n) \) then the operations \( n \) (stands for name) and \( l \) (stands for list) are defined for the \( i \)-th position of \( L \), \( 1 \leq i \leq n \) as follows:

\[
n \$ i \text{ of } L = x
\]
where either \( o_i = x \) is \((p_1, \ldots, p_m)\), or
\[
o_i = x.
\]
We further extend the operation \( n \) as:

\[
n(o_1, \ldots, o_n) \equiv (n \cdot o_1, \ldots, n \cdot o_n).
\]

Also,

\[
l \$ i \text{ of } L = \begin{cases} (p_1, \ldots, p_m) & \text{if } o_i = x \text{ is } (p_1, \ldots, p_m) \\
\Omega & \text{if } o_i = x \text{ and } x \text{ is an identifier}
\end{cases}
\]

Definition: We extend the operation \( l \) to specify the right-most position of an element of a list identified by its name. The operation \( \text{lm} \) (left-most element), is also defined similarly, that is, for the list:

\[
L = (o_1, \ldots, o_n)
\]

\[
x \text{ of } L = \begin{cases} \$ i \text{ of } L, \text{if } 1 \leq i \leq n, n \cdot o_i = x \text{ and } x \not\in (o_{i+1}, \ldots, o_n) \\
\Omega \text{, } \forall i, 1 \leq i \leq n, n \cdot o_i \neq x
\end{cases}
\]

Also,

\[
x \text{ lm } L = \begin{cases} \$ i \text{ of } L, \text{if } 1 \leq i \leq n, n \cdot o_i = x \text{ and } x \not\in (o_1, \ldots, o_{i-1}) \\
\Omega \text{, } \forall i, 1 \leq i \leq n, n \cdot o_i \neq x
\end{cases}
\]

Example 3: Given \( L = (a, bc \text{ is } (x, y), de, de \text{ is } (u, v), f) \), then:

\[
bc \text{ of } L = \$ 2 \text{ of } L
\]

\[
n(bc \text{ of } L) = n(\$ 2 \text{ of } L) = bc
\]

\[
l(bc \text{ of } L) = l(\$ 2 \text{ of } L) = (x, y).
\]

Also,
de \text{l} \text{m} L = \text{§}_3 \text{ of } L
\text{n}(\text{de \text{l} \text{m} L}) = \text{de}
\text{l}(\text{de \text{l} \text{m} L}) = \Omega.

\textbf{Definition:} Given a list \( L \) is \((o_1, \ldots, o_n)\), then the operators \text{next} and \text{prev} are defined as:
\[
\begin{align*}
\text{prev}(\text{§}_i \text{ of } L) &= \begin{cases} \\
\text{§}_{i-1} \text{ of } L & , \text{ for } 2 \leq i \leq n \\
\Omega & , \text{ otherwise}
\end{cases} \\
\text{next}(\text{§}_i \text{ of } L) &= \begin{cases} \\
\text{§}_{i+1} \text{ of } L & , \text{ for } 1 \leq i \leq n-1 \\
\Omega & , \text{ otherwise}
\end{cases}
\end{align*}
\]

To duplicate part of a list the following definition is provided.

\textbf{Definition:} Given the list \( L \) is \((o_1, \ldots, o_n)\) then
\[
\begin{align*}
\text{copy}_1(L, X) &= \begin{cases} \\
(o_1, \ldots, o_{i-1}), \text{ where either } X = \text{§}_i \text{ for } 2 \leq i \leq n & \\
\text{or } \exists j: 2 \leq j \leq n, \text{ } \text{n}(o_j) = X \\
\text{and } \text{l} \text{m} L = \text{§}_i \text{ of } L & \\
( ) & , \text{ otherwise}
\end{cases} \\
\text{copy}_r(L, X) &= \begin{cases} \\
(o_{i+1}, \ldots, o_n), \text{ where either } X = \text{§}_i \text{ for } 1 \leq i \leq n-1 & \\
\text{or } \exists j: 1 \leq j \leq n-1, \text{ } \text{n}(o_j) = X \\
\text{and } \text{of } L = \text{§}_i \text{ of } L & \\
( ) & , \text{ otherwise}
\end{cases}
\end{align*}
\]

\text{N.B.} the empty list in this report is denoted by either ( ) or \( \emptyset \).

\textbf{Example 4:} Given the list of Example 3 then:
\[
\begin{align*}
\text{copy}_1(L, \text{de}) &= (a, bc \text{ is } (x, y)) \\
\text{copy}_r(L, \text{de}) &= (f)
\end{align*}
\]

To insert an element into the list in a proper position, the following
operations are defined:

**Definition:** Given the list \( L = (o_1, \ldots, o_n) \) then:

\[
\begin{align*}
\text{insertl}(L, X, p): & \quad 1 \mapsto (o_1, \ldots, o_{i-1}, p, o_i, o_{i+1}, \ldots, o_n), \\
& \quad \text{where either } X = \$ \text{ for } 1 \leq i \leq n \\
& \quad \text{or } \exists i : 1 \leq i \leq n \land o_i = X \\
& \quad \text{and } X \text{ of } L = \$ \text{ of } L \\
\end{align*}
\]

\[
\begin{align*}
\text{insertr}(L, X, p): & \quad 1 \mapsto (o_1, \ldots, o_n, X), \\
& \quad \text{where } X \text{ is an identifier and } \\
& \quad X \not\in n(o_1, \ldots, o_n) \\
\end{align*}
\]

\[
\begin{align*}
& \quad (o_1, \ldots, o_{i-1}, p, o_{i+1}, \ldots, o_n), \\
& \quad \text{where either } X = \$ \text{ for } 1 \leq i \leq n \\
& \quad \text{or } \exists i : 1 \leq i \leq n \land o_i = X \\
& \quad \text{and } X \text{ of } L = \$ \text{ of } L \\
\end{align*}
\]

\[
\begin{align*}
& \quad (o_1, \ldots, o_{i}, X, o_{i+1}, \ldots, o_n), \\
& \quad \text{where } X \text{ is an identifier and } \\
& \quad X \not\in n(o_1, \ldots, o_n). \\
\end{align*}
\]

N.B. Notice that, if \( X \not\in (o_1, \ldots, o_n) \) then

\[
\begin{align*}
\text{insertl}(L, X, p) = \text{augl}(L, X) \\
\text{insertr}(L, X, p) = \text{augr}(L, X)
\end{align*}
\]

**Example 5:** For the list \( L = (a, b, c, c, d) \) and

\[
\begin{align*}
\text{insertr}(L, x, 2) : & \quad 1 \mapsto (a, b, x, c, c, d) \\
\text{and} \quad \text{insertr}(L, c, x) : & \quad 1 \mapsto (a, b, c, c, x, d).
\end{align*}
\]

To delete an element of a list the following operators are defined.

**Definition:** Given the list \( L = (o_1, \ldots, o_n) \), then

\[
\begin{align*}
\text{deall}(L, X): & \quad 1 \mapsto (o_1, \ldots, o_{i-1}, o_i, o_{i+1}, \ldots, o_n), \\
& \quad \text{where either } X = \$ \text{ for } 1 \leq i \leq n \\
& \quad \text{or } \exists i : 1 \leq i \leq n \land o_i = X \\
& \quad \text{and } X \text{ of } L = \$ \text{ of } L \\
& \quad (o_1, \ldots, o_n), \text{ where } X \text{ is an identifier and } \\
& \quad X \not\in n(o_1, \ldots, o_n)
\end{align*}
\]


\[ \text{defr}(L, X) : 1 \cdot L + \begin{cases} (o_1, \ldots, o_{i-1}, o_{i+1}, \ldots, o_n), \\ \text{where either } X = \$_i, \ l \leq i \leq n-1, \\ \text{or } \exists j : 1 \leq j \leq n, n(o_j) = X \\ \text{and } X \text{ of } L = \$_i \text{ of } L \end{cases}. \]

Example 6: For the list \( L \) is \((a, b, c, d, d \text{ is } (x, y), e, f)\)

we have: \( \text{defr}(L, \$_4) : 1 \cdot L \rightarrow (a, b, c, d \text{ is } (x, y), e, f) \)

and \( \text{defr}(L, d) : 1 \cdot L \rightarrow (a, b, c, d, e, f) \).

Finally, to delete a portion of a list we define the following operations:

**Definition:** Given a list \( L \) is \((o_1, \ldots, o_n)\), then

\[ \text{triml}(L, X) : 1 \cdot L \rightarrow \begin{cases} (o_1, \ldots, o_n), \\ \text{where either } X = \$_i, \ 1 \leq i \leq n-1, \\ \text{or } \exists j : 1 \leq j \leq n, n(o_j) = X \\ \text{and } \text{Im} L = \$_i \text{ of } L \end{cases}. \]

**Example 7:** For the list \( L \) is \((a, b, c, d, d \text{ is } (x, y), e, f)\)

we have: \( \text{triml}(L, \$_4) = (a, b, c) \)

and \( \text{triml}(L, d) = (a, b, c, d) \).

**Notice:** Hereafter and especially in Appendix 2, whenever the operator of

is applied and the name of any list \( a \) is clear from the context, we write
β and §₁ for β of α and §₁ of α respectively. Here β is a system identifier †.

4.2 DIRECTED GRAPHS

The operator k in the next chapter is defined to inter-relate the various items of our program space. This operator is an extension of the relation ρ which is described in the definition of a directed graph.

Definition: A (finite) directed graph (or digraph) G is a pair (A, ρ) where:

I. A is a finite set of elements called nodes.

II. ρ is a binary relation on A ††.

Since an item in the program space may be related to itself, by including (x, x) ∈ ρ for x ∈ A, we have slightly departed from the definition of directed graph in Harary [14].

Example 8: If G = (A, ρ), where

A = \{a₁, a₂, a₃, a₄, a₅, a₆\}

and ρ = \{(a₁, a₂), (a₂, a₂), (a₂, a₃), (a₃, a₂), (a₃, a₄), (a₃, a₅), (a₄, a₁)\}

then the directed graph can be represented as:

![Diagram](image)

FIGURE 4.1

† A system identifier can be defined in BNF as follows:

<system identifier>::=<identifier> | #<identifier>

†† A relation ρ on A is a subset of AxA
The relation $\rho$ may also be described in the following manner:

$\rho a_1 = \{a_2\}$
$\rho a_2 = \{a_2, a_3\}$
$\rho a_3 = \{a_2, a_4, a_5\}$
$\rho a_4 = \{a_1\}$
$\rho a_5 = \emptyset$
$\rho a_6 = \emptyset$

**Definition:** Given a directed graph $D=(A,\rho)$, then the pseudo-inverse $\rho^{-1}$ is defined as:

$\rho^{-1}y = \{x : (x, y) \in \rho\}$

**Example 9:** The pseudo-inverse $\rho^{-1}$ of Example 8 for each node is:

$\rho^{-1}a_1 = \{a_4\}$
$\rho^{-1}a_2 = \{a_1, a_2, a_3\}$
$\rho^{-1}a_3 = \{a_2\}$
$\rho^{-1}a_4 = \{a_3\}$
$\rho^{-1}a_5 = \{a_3\}$
$\rho^{-1}a_6 = \emptyset$

4.3 MARKOV ALGORITHMS AND RECURSIVE FUNCTIONS

In this section we define, formally, recursive functions and show that any such function can be defined by a Markov algorithm.

4.3.1 Recursive Functions

First we define the following set of basic functions:

I. The zero function. $z : x \mapsto 0 \quad x \in \mathbb{N}$

II. The successor function. $s : x \mapsto x+1 \quad x \in \mathbb{N}$

III. The projection functions. $p_{k}^{i} : (x_1, x_2, \ldots, x_k) \mapsto x_i, x_i \in \mathbb{N}, 1 \leq i \leq k$

From these functions we can construct new ones by three processes, composition, recursion and minimalization. These are explained below.
1. Composition: Given a $m$-adic function $g$ and the $k$-adic functions $h_1, h_2, \ldots, h_m$, then the $k$-adic function $f$ where,

$$f: (x_1, x_2, \ldots, x_k) \mapsto g(h_1(x_1, \ldots, x_k), \ldots, h_m(x_1, \ldots, x_k))$$

is the "composition" of $g$ with $h_1, h_2, \ldots, h_m$.

2. Recursion: Given a $k$-adic function $g$ and a $k+2$-adic function $h$, let the $k+1$-adic function $f$ be defined from $g$ and $h$ as follows:

$$f: (x_1, \ldots, x_k, y) \mapsto \begin{cases} 
g(x_1, x_2, \ldots, x_k), & \text{if } y = 0 
\h(x_1, x_2, \ldots, x_k, y-1, f(x_1, \ldots, x_k, y-1)), & \text{if } y \geq 1.
\end{cases}$$

The function $f$ is said to be obtained by "recursion" from $g$ and $h$.

3. Minimalization: Given a $k+1$-adic function $g$ then, if for any values $x_1, x_2, \ldots, x_k$ there exist a number $y$ such that:

$$g(x_1, x_2, \ldots, x_k, y) = 0$$

then, the least number $y$ which satisfies this relation and denoted by:

$$\mu y(g(x_1, \ldots, x_k, y) = 0)$$

is said to be achieved by "minimalization" from $g$. $\mu$ is called the minimalization operator.

**Definition:** If a function $f$ can be obtained from the basic functions by a finite number of applications of composition and recursion then $f$ is said to be "Primitive recursive".

**Definition:** A function $f$ is "recursive" if it can be defined by a finite number of the applications of the composition, recursion and the minimalization operator $\mu$ beginning with the basic functions.

**Example 10:** The function: $\text{add}(x, y) = x+y$

is primitive recursive. In fact, we can define:

$$\text{add}(x, y) = \begin{cases} 
p_1^1(x), & \text{if } y = 0 
\h(x, y-1, \text{add}(x, y-1)), & \text{if } y > 0
\end{cases}$$
For an example of a (non-primitive) recursive function see [11], page 250.

There are a number of primitive functions (operations) needed for the model of the next chapter (cf. Table 5.2). Let $A$ be a primitive function, $C$, $F$ and $D$ be functions such that:

- $A : U^k \rightarrow V^1$
- $C : U^k \rightarrow \mathbb{N}^m$
- $F : \mathbb{N}^m \rightarrow \mathbb{N}^n$
- $D : \mathbb{N}^n \rightarrow V^1$

where $k, l, m, n \in \mathbb{N}\backslash\{0\}$, $U$ and $V$ are sets (not necessarily of integers) and:

$$D \circ F \circ C \equiv A .$$

Therefore, $C$ and $D$ allow the input and output for $A$ to be translated into (coded to and decoded from) numerals. It is possible to prove that $F$ is a primitive recursive function. We show this for a primitive function, others are left to the reader.

**Example 11:** The concatenation function is defined as:

$$A_{\text{con}}(x, y) = xy,$$

where $x$ and $y$ are strings of characters.

If $A$ is the list of characters and the string $x = a_1 a_2 \ldots a_m$ corresponds to the unique prime factorisation [16]:

$$\hat{a}_1 \hat{a}_2 \ldots \hat{a}_m,$$

where $\hat{a}_i = \text{ordinal of } a_i \text{ of } A$, $1 \leq i \leq m$ then the functions $C$ and $F$, for the strings $x = a_1 a_2 \ldots a_m$ and $y = b_1 b_2 \ldots b_n$, are defined as follows:

$$C(x, y) = (2^{a_1} 3^{a_2} \ldots p_m^{a_m}, 2^{b_1} 3^{b_2} \ldots p_n^{b_n})$$

$$F(u, v) = 2^{a_1} 3^{a_2} \ldots p_m^{a_m} p_{m+1}^{b_1} p_{m+2}^{b_2} \ldots p_{m+n}^{b_n},$$

where $u = 2^{a_1} 3^{a_2} \ldots p_m^{a_m}$ and $v = 2^{b_1} 3^{b_2} \ldots p_n^{b_n}$. 

For a string $x = a_1 a_2 \ldots a_m$, the function $S(m)$ is defined as:

$$S(m) = \sum_{a_i \in x} i.$$
It is possible to prove that $F$ is a primitive function (cf. [11], Example 4). Therefore, $F$ can be written in terms of the basic functions. If we define $D$ as:

$$D(t) = a_0 a_1 \ldots a_m,$$

where $t = 2^{a_0} 3^{a_1} \ldots p_m^{a_m}$.

Then, for the strings $x$ and $y$:

$$D \circ F \circ C(x,y) = A_{\text{con}}(x,y).$$

However, this method is not suitable for the system described in the next chapter, because of the following reasons:

I. Since $C$ changes the values of the arguments of the primitive functions to their corresponding prime factorization, if the process of defining a PSS program is stopped in the middle, these numbers (if they appear) are not known to the person considering the process.

II. The system of the next chapter is highly dependent on pattern matching (for example, its syntax analyser), but the recursive functions are based on arithmetic operations (in terms of high level language). This is a form of inconsistency between the system and the primitive functions.

As it is possible to prove that each function $F$ corresponding to a primitive function $A$, is a recursive one, and it is proved that each recursive function is Markov-computable ([11], p.220, Proposition 5.8), it therefore follows that we can define these functions in terms of Markov algorithms which are based on pattern matching.

4.3.2 Markov Algorithms

The Markov algorithms are based on a substitution operation which is applied to an (input) string from a given set of characters. That is, it takes a string $x$ and through a number of steps (productions) transforms it
to the (output) string g. This string is flexible and can be altered in length.

We shall consider the specification of a transformation within an alphabet A and assume that the symbols + (the right arrow) and . (the dot) are not in A. A "simple production" or "terminal production" is a statement of the form:

\[ x \rightarrow y \text{ or } x \rightarrow .y \text{, where } x, y \in \mathcal{A}^* \]

respectively. The production \( x \rightarrow (.)y \) denotes either a simple or terminal production. \( x \rightarrow (.)y \) is said to be applicable to the input string \( u \), if \( x \) occurs in \( u \). In this case the application of the production to \( u \) causes substitution of the left most occurrence \( x \) in \( u \) by \( y \).

**Example 12:** The production \( 'e \rightarrow de' \) in the set of English letters A is applicable to string "evil" and changes it to "devil", but it is inapplicable to the string "God".

A "Markov scheme", in a set A is a finite ordered set:

\[
P_1 \rightarrow (.)Q_1 \\
P_2 \rightarrow (.)Q_2 \\
\vdots \\
P_n \rightarrow (.)Q_n
\]

where \( P_i, Q_i \in \mathcal{A}^* \), \( 1 \leq i \leq n \). The execution of the Markov algorithm based on this sequence is demonstrated in Figure 4.2.

Let the integer numbers 0, 1, 2, 3... be represented by 1, 11, 111, 1111, ... . If the symbol * is used to separate the operands of a binary operation then we can define the basic integer arithmetic operations in the alphabet \{1, *\}.

**Example 14:** The algorithm for computing \( m-n \), where \( m \geq n \) is:

1. \( 1*1** \)
2. **+1

Let us trace through the execution of this algorithm given an input string.

\( \mathcal{A}^* \) is the set of strings over the alphabet \( A \) including the empty string \( \lambda \).
working string := input string

\[ i := 1 \]

\[ \text{yes} \quad \text{No} \]

\[ i > n \]

\[ P_i \text{ in working string} \]

\[ \text{No} \]

\[ i := i + 1 \]

\[ \text{substitute first occurrence of } P_i \text{ by } Q_i \]

\[ P_i \not\rightarrow (\cdot) Q_i \text{ terminal} \]

\[ \text{No} \]

\[ \text{Yes} \]

\[ \text{output := working string} \]

halt

FIGURE 4.2
Example 15: The multiplication for \( m \times n \), where \( m, n > 0 \), is defined by the scheme:

1. \(*1 \rightarrow a\beta\)
2. \(1a \rightarrow a\gamma1\)
3. \(\alpha \rightarrow \Lambda\)
4. \(1\gamma \rightarrow \gamma1\)
5. \(\beta \rightarrow *\)
6. \(1 \rightarrow *\)
7. \(\gamma \rightarrow \gamma\delta\)
8. \(\gamma\delta \rightarrow \delta1\)
9. \(\delta \rightarrow \Lambda\)
10. \(* \rightarrow \Lambda\)

An execution trace for this algorithm on the input string \( 111 \times 11 \) is:

\[
111 \times 11 \rightarrow 111a\beta1 \rightarrow 111\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma1 \rightarrow 111\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma\gamma \}

The design of Markov algorithms for integer addition and division are left to the reader. However, we appreciate that the calculation of an arithmetic expression such as \( 2576 \times 5769 - 100 \times 5769 \) would be tedious and impractical. Moreover, since our goal is to define the semantic requirements of PSS in terms of the characters used in the program, we would use the alphabet \( \{0, 1, 2, \ldots, 9, +, -, *, /\} \) instead of \( \{1, *\} \) when describing arithmetic operations. It is also more convenient to use the so-called Extended Markov algorithm (EMA) [6].
A Markov algorithm operates on a single input string only. But an EMA contains \( n, n \geq 1 \), input strings. In each production \( R_i \rightarrow (.)S_i \) of an EMA, \( R_i \) and \( S_i \) are vectors of length \( n \). Execution is similar to that of a MA, i.e. upon the execution of each EMA production, the calculation process starts from the \( R_i \rightarrow (.)S_i \). Since the output might be more than one string the symbol \( \varepsilon \) is appended to one of the output strings, to show that it is the \( i \)-th component.

Example 16: The following EMA scheme executes the function: 

\[
f(s,u,v) = (su,sv)
\]

where \( s, u, v \) are strings of characters in \( A = \{a, b, c, \ldots, z, \ldots\} \)

\[
\begin{align*}
\alpha \xi & \rightarrow \xi \alpha & \xi \in A \\
I & \begin{cases}
\lambda + \lambda \\
\lambda + \lambda \\
\lambda + \lambda \\
\lambda + \lambda \\
\lambda + \lambda \\
\end{cases} & \begin{cases}
\alpha_1, \beta_1, \gamma, \gamma_1 \in A \\
P, Q, R \in A^* \\
P, Q, R \in A \\
P, Q, R \in A^* \\
P, Q, R \in A^* \\
\end{cases} \\
II & \beta \xi \rightarrow \xi \beta \\
III & \gamma \xi \rightarrow \xi \gamma \\
IV & \begin{cases}
\alpha \gamma_1 \rightarrow \lambda \\
\beta \gamma_1 \rightarrow \varepsilon P Q \\
\gamma \gamma_1 \rightarrow \varepsilon P R \\
\end{cases} \\
V & \begin{cases}
\lambda + \alpha_1 \\
\lambda + \beta \beta_1 \\
\lambda + \gamma \gamma_1 \\
\end{cases}
\end{align*}
\]

Finally, we describe the composition of the Markov algorithms. Let \( A \) and \( B \) be algorithms in an alphabet \( A \), with productions \( q_1, q_2, \ldots, q_n \) and productions \( p_1, p_2, \ldots, p_m \) respectively.

The string \( z \), obtained from \( A \circ B (x) \),
where: \( x = b_1 b_2 \ldots b_k, 1 \leq b_i \leq k, b_i \in \mathbb{A} \)

and

\[ z = c_1 c_2 \ldots c_k, 1 \leq c_i \leq \xi, c_i \in \mathbb{A} \]

is not obtained from the scheme \( p_1, p_2, \ldots, p_m, q_1, q_2, \ldots, q_n \), because the execution may be terminated in one of the terminal productions of \( B \) leaving:

\[ y = B(x) \]

where

\[ y = a_1 a_2 \ldots a_t, 1 \leq i \leq t, a_i \in \mathbb{A}. \]

At first glance, it seems that if we remove the termination "dots" from \( p_i, 1 \leq i \leq m \), then we let the process of execution continue, and hence it might cause the execution of \( q_i, 1 \leq i \leq n \). But this is not the solution, because when the string \( y \) appears, the execution might continue from the first production (i.e. \( p_1 \)) rather than \( q_1 \). To overcome this conflict we define the alphabet \( \mathbb{A}^1 \), isomorphic to alphabet \( \mathbb{A} \) (e.g. if \( \mathbb{A} = \{a, b, c\} \) then \( \mathbb{A}^1 = \{a^1, b^1, c^1\} \)), we also extend the concept of the empty string \( \mathbb{A} \), such that \( \lambda \in \mathbb{A} \) and \( \lambda^1 \in \mathbb{A}^1 \). A scheme for \( \mathbb{A} \circ \mathbb{B} \) can then be obtained by the following process:

I. Replace the termination dots of the scheme \( B \) by \( a, a^1 \in \mathbb{A} \) to the production \( P_1, P_2, \ldots, P_m \) from \( p_1, p_2, \ldots, p_m \) respectively. The application of \( P_1, 1 \leq i \leq m \), to string \( x \) produces the string:

\[ w = a_1 a_2 \ldots a_j-1 a_j a_j^1 \ldots a_t, \alpha \in \mathbb{A}. \]

II. Append the scheme \( \xi a \rightarrow a \xi, a \xi+1 a \rightarrow \xi a, a \rightarrow A^1 \) to \( P_1, P_2, \ldots, P_m \) to obtain:

\[ \xi a \rightarrow a \xi, a \xi+1 a \rightarrow \xi a, a \rightarrow A^1, P_1, P_2, \ldots, P_m, \xi \in \mathbb{A}, \xi^1 \in \mathbb{A}^1. \]

The first three productions change \( w \) to:

\[ w^1 = a_1^1 a_2 \ldots a_j-1 a_j a_j^1 \ldots a_t. \]

III. The productions \( q_1, 1 \leq i \leq n \), are not applicable to \( w^1 \). In order to make this possible, change each constant \( a \in \mathbb{A} \), variable \( \xi \in \mathbb{A} \) and marker \( \beta \) to \( a^1, \xi^1 \) and \( \beta^1 \) respectively. Replace each termination dot to \( a^1, a^1 \notin \mathbb{A}^1 \) to obtain: \( Q_1, Q_2, \ldots, Q_n \) and produce the new scheme:
The execution of this scheme on the string \( x \) yields:
\[
z = \underbrace{c_1 \, c_2 \, \ldots \, c_j \, a \, c_{j+1} \, \ldots \, c_k}_1
\]

IV. To obtain \( z = c_1 \ldots c_k \), the productions
\[
\xi \alpha \rightarrow \alpha \xi, \alpha \xi \rightarrow \xi \alpha, \alpha \rightarrow \Lambda
\]
change each \( c_i \) to \( c_i \), \( 1 \leq i \leq k \), remove \( \alpha \) and terminate the process.

Therefore, by embedding these productions in the previous ones we obtain the necessary scheme for \( \mathcal{A} \circ \mathcal{B} \) as follows:

1. \( \xi \alpha \rightarrow \alpha \xi \)
2. \( \alpha \xi \rightarrow \xi \alpha, \alpha \rightarrow \Lambda \)
3. \( \alpha \rightarrow \Lambda \)
4. \( \xi \alpha \rightarrow \alpha \xi \)
5. \( \alpha \xi \rightarrow \xi \alpha \)
6. \( \alpha \rightarrow \Lambda \)
7. \( P_1 \)
8. \( P_2 \)
\vdots
m+6. \( P_m \)
m+7. \( Q_1 \)
m+8. \( Q_2 \)
\vdots
m+n+6. \( Q_n \)

Notice that to compose the Markov algorithms \( \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_j \) in \( \mathcal{A} \) to give:
\[
\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3 \circ \ldots \circ \mathcal{A}_j
\]
we need \( j \) isomorphic alphabets
\[
\mathcal{A}, \mathcal{A}^1, \mathcal{A}^2, \ldots, \mathcal{A}^{j-1}
\]
Notice also that this construction can also be extended to EMA's.
4.4 LOGICAL INTER-RELATIONSHIP BETWEEN W-GRAMMARS, MARKOV ALGORITHMS AND
COMPUTER PROGRAMS

Let $P_w$ be a parser to run on a computer (finite state machine) and to
recognise the language generated by our PSS W-grammar. Further, let $A$ be
the set of strings with the following characteristics:

I. For each $x$ in $A$, $x=yz$, where $y$ is a PSS program and $z$ denotes an
   input string.

II. The parser $P_w$ processes $x$ and the machine halts, leaving a string
    $u$ as the output of the program $y$.

If we treat the W-grammar parser as a function $w$ with the set $A$
for its domain, since there is an algorithm (the program $P_w$) to
compute the function $w(x)$, $x\in A$, then $w$ is an effective computable
function.

If we accept Church's thesis [15], that is, every effectively computable
function can be defined by a recursive function, then $w$ can be specified
by a recursive function. Moreover, any such function has an equivalent
Markov algorithm (see [11], Proposition 5.9 of Chapter 5).

The direct construction of a single MA to define the full requirements
of our PSS is too clumsy and bulky to be considered. Therefore, following
the style of the hyper rules (Appendix 1), we may design some MA's and
compose them (in some way) to satisfy the PSS requirements. But this
solution also is impractical because for the composition of each pair of

---

†Because we use a W-grammar, only a parser is required. This corresponds
to what in general would be a compiler and run time system.

‡‡This is decidable on a finite state machine.

+++By an effective computable function $f(x)$, we mean a function that can
   be algorithmically specified; that is, $f(a)$ for the value $a$ can be
determined in a precisely specified set of instructions in a mechanical
way. It is clear that neither the term "algorithmically" nor the phrase
"mechanical way" is precisely defined. However, for further details the
reader may refer to [18], Chapter 5.
MA's we need a new alphabet isomorphic to the original alphabet. Moreover, comparing with Appendix I, if we design a MA for each hyper rule, then the reader can imagine that we possibly have to obtain a MA of the composition of more than six-hundred MA's! To overcome these conflicts, it is possible to extend the definition of MA by introducing the concepts of sub-algorithms and labels [17]. The labels remove the need for combining MA's (and thus, the need for isomorphic alphabets) and the design of the sub-algorithms allows us to concentrate on each part of the definition of PSS independently. However we still need some markers (see [17] page 42) for each sub-algorithm of the MA.

Let us recap the points so far:

I. We can define a single MA but this is impractical.

II. We add the concept of sub-algorithms to break the whole MA into a sequence of sub-algorithms, so we have:

   MA & subalgorithms

III. To decrease the number of isomorphic alphabets the concept of labels is introduced. Now we have,

   MA & subalgorithms & labels

Gradually, we are moving from a pure MA to a model in which basic MA's are implemented with an overall framework. This framework, which could, if necessary, be formalised by further MA's allows the structure of the system to be more easily seen and helps in the tracing of individual computations. This is the basic philosophy behind the CPS system described in the next chapter.
Chapter 5

A CPS DESCRIPTION OF PSS
The Carabiner Program Space (CPS) model was introduced by Cooke [6] in 1975 and consists of three parts: the program space, a high level language and a set of Carabiner rules.

The program space is a finite set of lists each of which contain a number of items. These items may be linked to each other in some way. Here, the language is our PASCAL subset. The set of rules are a mixture of BNF productions and some Macro rules. This set with the help of the program space defines the syntax, the compile time semantics and the run time semantics of the language.

The Carabiner program space was used to define the language X[6] and the PLECS languages [7,8] based on Ledgard's ten Mini languages [9]. However, these were not able to demonstrate in detail the full potential of the system. The CPS for our (PSS) language shows that this method is as powerful as W-grammars for defining substantial high level languages. However, several new points are included to aid the development of the PSS system. These are:

I. From a theoretical point of view, we redefine some of the operations of the system.

II. Some of the suggestions by Cooke such as type declarations are redefined and some other features for example subroutine invocations are completely changed.

III. New language features not considered in [6] such as constant declarations, set type declarations and heap items are defined in terms of the model.

IV. The PSS language is recursive and hence makes full use of the dynamic storage allocation inherent in the CPS model.
5.1 THE CARABINER (CPS) SYSTEM

5.1.1 Philosophy

The concept of the extended Markov algorithms (EMA) explained in Chapter 4 for describing the requirements of a high level language leads to a highly complicated, bulky and impractical definition of the language. We replace this method by the more pragmatic CPS system. The CPS model is based on:

I. The BNF syntax definition of the language.

II. The operations $e, S, k$.

III. The program space.

IV. The list operations (cf. 4.1).

V. EMAs for preliminary operations.

The operations $e, S, k$ are described in the following subsections. Here we use the extended Markov algorithms to describe a machine (i.e. implementation) independent definition corresponding to the primitive operations of the language. In fact, EMAs represent the 'hardware' functions of the system.

To demonstrate the working of the CPS, we consider the language defined in Section 3.1.1. The BNF of this language is:

1. <program>::=var <main>
2. <main>::=<identifier>;<stmt>
3. <identifier>::=a|b
4. <stmt>::=<identifier>::=<expression>
5. <expression>::=<bool>|not <bool>
6. <bool>::=false|true

We need a structure in which to store the identifier in order to check the compile time restrictions. This can be represented by the initially empty list called STORE. We insert the identifier declared in the declaration part of the program in STORE.
Example 1: var a;
           ........
           ........

At this stage: STORE is (a)

Using the list operations defined in Section 4.1, this action can be described by: 
    augr(STORE,a)

In an assignment statement the left hand side identifier must be in STORE. This shows that the identifier is already declared.

Example 2: var a;
           a:=...

Here, the semantic definition is:

    if a \notin STORE then halt

If the identifier is not in STORE, (e.g. var a;b:=...) then the execution of the semantic injection causes unsuccessful termination of the process, leaving the word "halt".

The second version of the grammar including the compile time semantic injections is defined as follows:

1. <program>::=var <main>
2. <main>::=<identifier> ;
   2.1 \$augr(STORE, <> )\$
   2.1 <stmt>
3. <identifier>::= a|b
4. <stmt>::=<identifier> :=
   4.1 \$if <> \notin STORE then halt\$
   4.1 <expression>
5. <expression>::=<bool>|not <bool>
6. <bool>::= false|true

where, <> denotes the string generated by the preceding non-terminal labelled by n, and \$...\$ delimit the semantic injection(s).

We will show later that the conditional statement if...then... can be defined in terms of \$S operation (cf. 5.1.2.6).
5.1.2 Formal Definitions

The context sensitive restrictions of a programming language (here PSS), in CPS, are defined in terms of three operations $S, k, e$. The composition of these operations and the list operations (cf. 4.1) form the Macro rules (Appendix 2) to specify the semantics of the language.

5.1.2.1 The $S$ Operator

The Carabiner $S$ operator [6] is an extended form of the one defined by Wessellamper [10]. This operator is used to model equality tests and replacements. If we use 'v' and 'p' to denote value and position specifiers, respectively, then $S$ is defined as:

I. $S v_1, v_2, v_3$ yields, if $v_1 = v_2$ then $v_3$ else $v_1$
II. $S v_1, v_2, p_3$ yields, if $v_1 = v_2$ then $p_3$ else $v_1$
III. $S v_1, p_2, v_3$ = $v_1$
IV. $S v_1, p_2, p_3$
V. $S p_1, v_2, v_3$ = $p_1$
VI. $S p_1, v_2, p_3$
VII. $S p_1, p_2, p_3$ If $p_1 = p_2$ then substitute $p_3$ into $p_1$ otherwise do nothing
VIII. $S p_1, p_2, v_3$ If $p_1 = p_2$ then substitute $v_3$ into $p_1$ otherwise do nothing

Since, in our model the position specifiers and values are not comparable, the definitions III-VI are not applicable. We also make the assertion that the comparison tests in the PSS language are based on 2-valued logic (i.e. false, true) and they do not produce a position specifier therefore the second definition of $S$ cannot be used. Although, the definition VII can be used in this model we apply the operator $k$ which is

† Much of the material can also be found in [6].
more general and applicable for graph traversing (see the next section). Hence, the CPS takes the first and last definitions to specify the equality test and replacement respectively. Cooke [6] and Wesselkamper [10] have shown that the operator $S$ may change the state of the machine (here the program space).

The operators $S, k, e$ are so inter-related that their applications to PSS cannot be clearly and fully shown until all of them have been defined.

5.1.2.2 Operator $k$

The operator $k$ is introduced by Wesselkamper, for his CRAMMPON machine [10] to relate an identifier to its corresponding value. Here, this operator is extended to define the inter-relationship of the items (nodes) of various lists. Comparing $k$ with $p$ as a graph traversing operator in Section 4.2, we define $k$ to connect the positions of the nodes (in different lists) rather than the nodes themselves.

**Definition:** Given a set of finite lists,

$$A = \{L_1, L_2, L_3, \ldots, L_n\}$$

then $D = (P, k)$ is a digraph, where,

I. $P = \{\text{§}_i \text{ of } L_j | \text{§}_i \text{ is a position of } L_j, 1 \leq j \leq n\}$

II. $k$ is an anti-symmetric binary relation on $P$

**Example 3:** Let $A = \{\text{ATTRIB, STORE, HEAP}\}$

where: \text{ATTRIB is (integer, Boolean, real)}

\text{STORE is (x, y, z)}

\text{HEAP is (nil)}

then the digraph of:

\text{ATTRIB is (integer, Boolean, real)}

\text{STORE is (x, y, z)}

\text{HEAP is (nil, LE2)}
is the representation of: \( D = (p, K) \)

where:

\[
P = \{ \text{§ 1 of ATTRIB, § 2 of ATTRIB, § 3 of ATTRIB, } \\
\text{§ 1 of STORE, § 2 of STORE, § 3 of STORE, } \\
\text{§ 1 of HEAP, § 2 of HEAP} \}
\]

and:

\[
K \begin{array}{ll}
\text{§ 1 of ATTRIB} = \{ \text{§ 1 of STORE, § 2 of STORE} \\
\text{§ 2 of ATTRIB} = \emptyset \\
\text{§ 3 of ATTRIB} = \{ \text{§ 2 of HEAP} \\
\text{§ 1 of STORE} = \emptyset \\
\text{§ 2 of STORE} = \emptyset \\
\text{§ 3 of STORE} = \{ \text{§ 1 of HEAP} \\
\text{§ 1 of HEAP} = \emptyset \\
\text{§ 2 of HEAP} = \emptyset 
\end{array}
\]

The inverse operation \( K^{-1} \) (Comparing with \( p^{-1} \)) can be defined similarly.

N.B. Hereafter, we show \( x \) instead of \( \{x\} \) as the result of \( K \) (or \( K^{-1} \)) operator, e.g.

\[
K \begin{array}{ll}
\text{§ 3 of ATTRIB} = \text{§ 2 of HEAP} 
\end{array}
\]

for the previous example.

5.1.2.3 Operator \( e \)

The third basic operator \( e \) for CPS is defined for the following purposes:

I. To extract an item from its corresponding position in a list.

**Example 4:** In the structure:

\[
\text{STORE is } (\ldots, x, \ldots) \\
\text{HEAP is } (\text{nil, -1, false, } \ldots)
\]

we have: \( e(\text{§ 2 of HEAP}) = -1 \)

or \( e(K(x \text{ of STORE})) = -1 \)
II. To activate an operation, we differentiate between an active and non-active operation (function) by its representation in a single or double underlined form. By inserting the operator \( e \) in front of a non-active operation (function), it will be changed to an active one.

**Example 5**: Given \( L \) is \((a, b, c)\), then:

\[
\text{augr}(L, d)
\]

is an active operation and produces:

\[
L \text{ is } (a, b, c, d)
\]

But, \( \underline{\text{augr}}(L, d) \) is a non-active operation and does not change the list \( L \).

The effect of operator \( e \) is:

\[
e \ \underline{\text{augr}}(L, d) = \text{augr}(L, d)
\]

III. To execute the extended Markov algorithms defined in the model. If \( A \) is an EMA the operation:

\[
e^2 A = e \ e A
\]

means execute \( A \) and extract its result.

IV. Finally, given \( L \) is \((o_1, o_2, \ldots, o_n)\) we define:

\[
e(o_1, o_2, \ldots, o_n) = (e \ o_1, e \ o_2, \ldots, e \ o_n)
\]

5.1.2.4 Other Operations

The relation between two nodes either in the same list or different lists can be defined as follows:

**Definition**: Given the lists \( L_1, L_2, \ldots, L_n \), then for \( x \in L_i \) and \( y \in L_j \), \( 1 \leq i, j \leq n \):

\[
\begin{align*}
\text{link}(x, y) : & \quad \begin{cases} k \ x \mapsto k \ x \cup \{y\} \\
-1 \ y \mapsto k^{-1} \ y \cup \{x\} \end{cases} \\
\text{break}(x, y) : & \quad \begin{cases} k \ x \mapsto k \ x \subseteq \{y\} \\
-1 \ y \mapsto k^{-1} \ y \subseteq \{x\} \end{cases}
\end{align*}
\]
Example 6: If:

\[ L_1 \text{ is } (a, b, c) \]
\[ L_2 \text{ is } (e, f, g, h) \]
\[ L_3 \text{ is } (p, q, r) \]

then,

\[
\text{link}(b, f): \begin{cases} 
  k_b = \{s_2 \text{ of } L_2\} \\
  k_{-1} f = \{s_2 \text{ of } L_1\}
\end{cases}
\]
\[
\text{link}(b, r): \begin{cases} 
  k_b = \{s_2 \text{ of } L_2, s_3 \text{ of } L_3\} \\
  k_{-1} r = \{s_2 \text{ of } L_1\}
\end{cases}
\]

The connections between the lists are demonstrated as:

\[
\begin{array}{c}
\text{L is } (a, b, c) \\
\text{L is } (e, f, g, h) \\
\text{L is } (p, q, r)
\end{array}
\]

and the execution of:

\[
\text{break}(b, f): \begin{cases} 
  k_b = \{s_3 \text{ of } L_3\} \\
  k_{-1} f = \emptyset
\end{cases}
\]

causes the figure above to be changed to:

\[
\begin{array}{c}
\text{L is } (a, b, c) \\
\text{L is } (e, f, g, h) \\
\text{L is } (p, q, r)
\end{array}
\]

By a combination of the operations link and break, we can define new operations as follows:

Definition: Given lists \( L_1, L_2, \ldots, L_n \), then for \( a \in L_i, b \in L_j \) and \( c \in L_k \), \( 1 \leq i, j, k \leq n \):

\[
\text{adj}(a, b, c, d) = \begin{cases} 
  \text{break}(a, b) \\
  \text{link}(c, d)
\end{cases}
\]
Thus causing the link between a and b to be replaced by a link between c and d. Similarly we define:

\[ \text{adj}_1(a, b, c) = \text{adj}(a, b, a, c) \]
\[ \text{adj}_2(a, b, c) = \text{adj}(a, c, b, c) \]

5.1.2.5 Carabiner Space for PSS

In order to describe the requirements of PSS by the Carabiner system, we have prepared a suitable BNF grammar to define the syntax of the language. The context sensitive restrictions of PSS are described by a set of Macro rules (c.f. Appendix 2). These are used in different places in the BNF productions and act on the abstract program space. We define a list consisting of various lists and other nodes as the PSS program space. At each instance the space looks like the list:

\[
\text{SPACE is } (\text{OBJECT is } (...) , \\
\text{TYPE is } (...) , \\
\text{ATTRIB is } (...) , \\
\text{STORE is } (...) , \\
\text{HEAP is } (...) , \\
\text{INPUT is } (...) , \\
\text{OUTPUT is } (...) , \\
...... \\
...... \\
...... \\
\)
\]

where:

\[
\text{OBJECT is } (\text{constant, variable, procedure, function, formal value,} \\
\text{formal variable, formal procedure, formal function} \\
\)
\]

The list OBJECT determines the kind of an identifier or temporary identifier used either in ATTRIB or STORE. The list:

\[
\text{TYPE is } (\text{subrange, scalar, type identifier, array, record, set,} \\
\text{pointer})
\]

defines the characteristics of any type in ATTRIB. None of the lists OBJECT and TYPE will be changed during the process.
STORE is a list of the representations of local constant identifiers, temporary identifiers, variable and subroutine headings (as lists of formal parameters) at both compile time and run time. This is similar to the symbol table (excluding type definition) and the run time stack.

ATTRIB is a list of the modes of the identifiers in STORE, the internal representation of the type declarations and the temporary type identifiers.

HEAP contains the heap values. The lists INFILE and OUTFILE are internal representations of the input and output files, respectively. Other items (values, lists of values or lists of Carabiner codes) required in the validation and execution of a program are held in the list SPACE.

Example 7: The representation of:

```plaintext
const x=5;
y='abcd'
```

in the program space is:

```plaintext
SPACE is (OBJECT is (...........,constant,...),
    TYPE is (...scalar,...,array,...),
    ATTRIB is (...integer,...,ST...),
    STORE is (...........,x,y),
    HEAP is (...),
    INPUT is (...),
    OUTPUT is (...),
    +5,
    M is (a,b,c,d)
)
```
The structure of the types of $x$ (i.e. the integer of $\text{ATTRIB}$) and $y$ (i.e. $\#T$ of $\text{ATTRIB}$) are explained elsewhere. The list $M$ in $\text{SPACE}$ is the representation of a string. Although the above figure is a valid definition of the program space a better representation is presented in Figure 5.1.

![Diagram](image)

**FIGURE 5.1**

5.1.2.6 The $S$ Definition of CPS Statements and Set Operations

The way we defined the semantic requirements of the Example 2 (i.e. if a $\notin \text{STORE}$ then halt) may be thought of as contradicting the assertion that all semantic actions of the language are defined in terms of $S,e,k$ and list operations. However this is not so, because we can define CPS statements such as:

- if...then...
- if...then...else...
- while...do...
- repeat...until...

and the CPS predicates $\neg,\neq,\in,\notin,\cup,\cap,\subseteq,\supseteq,\lor$ (logical or) and $\land$ (logical and) in
terms of the operation $S$. We consider the first conditional statement and the operations $=, \vee, \in$. Others are left to the reader.

I. The conditional statement: if $\alpha$ then $\beta$ else $\gamma$

is equivalent to: $S(S \text{true}, \alpha, \beta), \text{true}, \gamma$

where $\alpha$, $\beta$ and $\gamma$ are in terms of the basic operators $(e, k, S)$ and list operations and $\alpha$ produce a boolean result (i.e. false or true), but not $\beta$ or $\gamma$. If $\beta$ or $\gamma$ produces a boolean result then we have a more complex form (see [6] and [10]).

II. The operation: $\alpha = \beta \equiv S(S \alpha, \beta, \text{true}), \alpha, \text{false}$

where $\alpha$ and $\beta$ are in terms of CPS operations and do not produce boolean values. If $\alpha$ and $\beta$ result in boolean values then:

$\alpha = \beta \equiv S(S \text{true}, \alpha, \beta), \text{true}, (S \text{true}, \beta, \alpha)$

III. If $p$ and $q$ are given in terms of the CPS operations and produce a boolean value then:

$p \vee q \equiv S(p, \text{false}, q)$

IV. The operation: $x \in \{x_1, \ldots, x_n\}$ is converted to:

$(x = x_1) \vee (x = x_2) \vee \ldots \vee (x = x_n)$

5.1.2.7 CPS Markov Algorithms

The Markov algorithms discussed in Section 4.3 provide a tool to define the semantics of the primitive operations such as integer and real addition and subtraction etc. Although these operations can be defined in terms of the operator $S$, this gives rise to a large string and is impractical. As an example, let us define the evaluation of $x+y$, in terms of $S$, where $x$ and $y$ are unsigned integers. If MAXINT is the maximum (implementation-dependent) unsigned integer then
Referring to the previous section, this CPS conditional statement can be written in terms of a string involving the operator $S$. However, depending on MAXINT this might be a very long string. Comparing with EMAs, the primitive operations can be defined in a much more sensible way consisting of several rules. Here we define the algorithms $A_{\text{sc}}$ and $A_{\text{rm}}$ on the set of unsigned integer $\text{INT} = \{0, 1, 2, 3, \ldots, 9\}$, others are left to the reader consulting [6] and[11].

I. $A_{\text{sc}}$: This algorithm defines the successor of an unsigned integer as follows:

1. $\alpha \xi \rightarrow \xi \alpha \quad \xi \in \text{INT}$
2. $\alpha \rightarrow \beta \gamma$
3. $0 \beta \rightarrow 1$
4. $1 \beta \rightarrow 2$
5. $2 \beta \rightarrow 3$
6. $3 \beta \rightarrow 4$
7. $4 \beta \rightarrow 5$
8. $5 \beta \rightarrow 6$
9. $6 \beta \rightarrow 7$
10. $7 \beta \rightarrow 8$
11. $8 \beta \rightarrow 9$
12. $9 \beta \rightarrow 0$
13. $6 \beta \rightarrow 1$
14. $\delta \text{MAXINT} \gamma \rightarrow \text{halt}$
15. $\gamma \rightarrow \Lambda$
16. $\delta \rightarrow \Lambda$
17. $\Lambda \rightarrow \delta \alpha$

N.B. Recall from Section 4.3.2, the symbol $\Lambda$ represents the empty string and also that MAXINT must be replaced by the actual maximum integer.
II. $A_{rm}$: This algorithm removes the insignificant zeroes from a number
(integer or real).

1. $a0 \rightarrow a \quad a, E \notin INT$
2. $a\xi \rightarrow .\xi \quad \xi \in INT \setminus \{0\}$
3. $a_+ \rightarrow .0$
4. $a E+.0E$
5. $a+0$
6. $A+0$

The necessary Extended Markov Algorithms for PSS are listed in the following table. The definition of each algorithm should be followed from its corresponding hyper rule in PSS $W$-grammar.

<table>
<thead>
<tr>
<th>EMA</th>
<th>input(s)</th>
<th>output(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{scx}(x)$</td>
<td>$x$ is unsigned integer</td>
<td>$x+1$</td>
</tr>
<tr>
<td>$A_{prx}(x)$</td>
<td>$x$ is unsigned integer</td>
<td>$x-1$</td>
</tr>
<tr>
<td>$A_{i+}(x,y)$</td>
<td>$x$ and $y$ are integers</td>
<td>$x+y$</td>
</tr>
<tr>
<td>$A_{i-}(x,y)$</td>
<td>$x$ and $y$ are integers</td>
<td>$x-y$</td>
</tr>
<tr>
<td>$A_{i*}(x,y)$</td>
<td></td>
<td>$x*y$</td>
</tr>
<tr>
<td>$A_{i/}(x,y)$</td>
<td></td>
<td>$x$ div $y$</td>
</tr>
<tr>
<td>$A_{mod}(x,y)$</td>
<td></td>
<td>$x$ mod $y$</td>
</tr>
<tr>
<td>$A_{r+}(x,y)$</td>
<td></td>
<td>$x+y$</td>
</tr>
<tr>
<td>$A_{r-}(x,y)$</td>
<td></td>
<td>$x-y$</td>
</tr>
<tr>
<td>$A_{r*}(x,y)$</td>
<td>$x$ and $y$ are reals</td>
<td>$x*y$</td>
</tr>
<tr>
<td>$A_{r/}(x,y)$</td>
<td></td>
<td>$x/y$</td>
</tr>
<tr>
<td>$A_{ir}(x)$</td>
<td>$x$ is integer</td>
<td>$x E+0$</td>
</tr>
<tr>
<td>$A_{sg}(x,y)$</td>
<td>$x$ is $-$ or $+$ and $y$ is real or integer</td>
<td>$-1<em>y$ or $+1</em>y$</td>
</tr>
</tbody>
</table>
| $A_{rch}(x)$ | $x$ is real in any form   | $x$ is in the form $a E b$ integer or real, obtained from $x$ by deleting left nulls (if any) $xy$(concatenation) $z$, where $y=xz$
| $A_{rm}(x)$  | $x$ is real or integer, possibly starting with nulls |
| $A_{con}(x,y)$| $x$ and $y$ are strings   | $z$, where $y=xz$       |
| $A_{sep}(x,y)$| $x$ and $y$ are strings   |                          |

FIGURE 5.2
5.1.2.8 Other PSS Requirements

In this section some notation is introduced to abbreviate the operations and the elements of CPS.

I. Notations:

\[
\begin{align*}
\text{sl} \ x, y & \quad \text{is written for} \quad S \ x, x, y \\
\text{loc}(x) & \quad \text{is written for} \quad \text{augr}(\text{STORE}, x) \\
\text{at}(x) & \quad \text{is written for} \quad \text{augr}(\text{ATTRIB}, x)
\end{align*}
\]

where \( x \) and \( y \) are elements of CPS, \( \text{loc}(x) \) is written for \( \text{augr}(\text{STORE}, x) \) and \( \text{at}(x) \) is written for \( \text{augr}(\text{ATTRIB}, x) \) where \( x \) is either an element or a set of elements of CPS. We also use the operator \( \text{trans} \) in the following manner:

If \( \text{STORE} \) is represented as:

\[
\text{STORE is } (\ldots, F, \ldots, \#\text{BPTR}, \ldots)
\]

\[
\text{BODY is } (\ldots)
\]

then:

\[
\text{trans}(a) = \text{augr}(k \ #\text{BPTR}, a)
\]

and

\[
\text{trans}(a_1; a_2; \ldots; a_n) = \text{trans}(a_1); \text{trans}(a_2); \ldots; \text{trans}(a_n)
\]

where \( a, a_1, a_2, \ldots, a_n \) are non-active Carabiner operations.

II. Implementation-dependent lists are defined as:

A. \( \text{IN} \)†† list of all integers
B. \( \text{R} \) list of all reals
C. \( \text{CHAR} \) list of all characters
D. \( \text{BOOL} \) list of boolean values, i.e. \( \text{BOOL} = \{\text{false, true}\}^†\)

III. Implementation-dependent sets are defined as:

A. \( \text{IDSET} \) set of all identifiers
B. \( \text{WD} \) set of all word delimiters

† The implementor of the language may follow different approaches to define these lists and sets. For example, he may use range of integer rather than \( \text{IN} \) above.

†† Note: \( \text{IN} \) and \( \text{IN} \) are different.
C. \text{MAXCHAR}=\{1,2,3,\ldots,n\}, \text{The set of all valid PSS string lengths.}

D. \text{UINT} \quad \text{set of all unsigned integers.}

IV. The CPS set of positions of the standard subroutines are classified in the following sets (see Figure 5.5):

\[ AB=\{\text{succ } \text{lm STORE, pred } \text{lm STORE}\} \]
\[ \text{STANDF}=\{\text{ord } \text{lm STORE, chr } \text{lm STORE}\} \cup AB \]
\[ \text{STANDP1}=\{\text{real } \text{lm STORE, readln } \text{lm STORE, new } \text{lm STORE, dispose } \text{lm STORE}\} \]
\[ \text{STANDP2}=\{\text{write } \text{lm STORE, writeln } \text{lm STORE}\} \]
\[ \text{STANDP}=\text{STANDP1} \cup \text{STANDP2} \]

V. We also use the notation \text{eq, ineq, subset and supset} instead of \text{=, \#, \subseteq, and}\$
\text{\$ respectively so as not to have any confusion between =, \neq, \subseteq and \subseteq \text{ etc. in their underlined forms.}$

VI. Finally, the following priority table is defined for some of the CPS operators to avoid using many parantheses.

<table>
<thead>
<tr>
<th>operator(s)</th>
<th>priority number</th>
</tr>
</thead>
<tbody>
<tr>
<td>\lor, \land</td>
<td>1</td>
</tr>
<tr>
<td>eq, ineq, \in, \notin, subset, supset</td>
<td>2</td>
</tr>
<tr>
<td>\cup, \cap</td>
<td>3</td>
</tr>
<tr>
<td>s1</td>
<td>4</td>
</tr>
<tr>
<td>l, n</td>
<td>5</td>
</tr>
<tr>
<td>e</td>
<td>6</td>
</tr>
<tr>
<td>of, \text{lm}</td>
<td>7</td>
</tr>
<tr>
<td>k, k^{-1}</td>
<td>8</td>
</tr>
</tbody>
</table>

\text{FIGURE 5.3}

The operators \text{k, k^{-1}, of, lm, e, l and n} associate from right to left and the others associate in the normal way (i.e. from left to right).

\[\text{Where n is the maximum length of a PSS string.}\]
5.2 DEFINITION OF PSS

The strategy here follows the same philosophy as the W-grammar system for PSS (cf. 3.2.2). We use the program space to interpret the symbol table and run time stack. The only difference is that at run time we do not have the corresponding symbol table.

5.2.1. The Program

The definition of the standard prelude (Figure 3.5, lines 1,2) introduces a state of CPS containing various standard constant, types and subroutines which are linked to each other accordingly. The diagrams of Figure 5.4 and 5.5 represent the state of CPS at this stage.

Since the types of the nil value and the base type of the empty set may be of any scalar type [5] (except real), the lists:

\[ \text{#ANYPOINTER is (WIDTH is } (1)) \]

and

\[ \text{#ANYSET is (WIDTH is } (1)) \]

are augmented to ATTRIB to represent their corresponding types. The symbol # shows the difference between auxiliary (temporary) identifiers and those of PSS in the program space. The list:

\[ \text{WIDTH is } (n) \]

defines the number of run time units. Other lists in ATTRIB define the standard types integer, real, character and boolean. Figure 5.5 illustrates the definition of the standard constants (nil, false, true), subroutines and nameless procedure A. The items included in the non-empty lists of STORE represent the formal parameters of subroutines. However, most of these lists are not a complete representation of their corresponding routines because the type and the number of their parameters (e.g. read) cannot always be determined at this stage, and may vary from one call to another. In the W-grammar model this information is not initially
pointer ∈ TYPE

set ∈ TYPE

scalar ∈ TYPE

ATTRIB is( #ANYPONTE is(WIDTH is(1)), #ANYSET is(WIDTH is(1)), integer is(DEF is N, WIDTH is(1))

scalar ∈ TYPE

ATTRIB continued

real is(DEF is R, WIDTH is(1)), char is(DEF is CHAR, WIDTH is(1)),

scalar ∈ TYPE

ATTRIB continued

Boolean is(DEF is BOOL, WIDTH is(1))

FIGURE 5.4
FIGURE 5.5
available. The contents of $k$ odd (i.e., the Macro rule 204) is a sequence of CPS operations which specify the execution of the function odd. The subsequent sections show how these operations are used.

The rightmost node labelled by $A$ is associated with the program (compare with the last row of the label in Figure 3.6). Initially the list $kA$ is empty but at the end of compilation, this will contain a set of Carabiner operations corresponding to the program. In BNF rule 1 (Appendix 2) the operator $e$ in:

$$e(1(kA \text{ of } \text{STORE}))$$

activates the contents of $kA$ to initiate the computational process of the program.

The assignment of a nil value to a variable at run time suggests that the memory unit of the variable points to the nil value in HEAP. Therefore initially:

$$\text{HEAP is (nil)}$$

The identifiers in a program are augmented to STORE and ATTRIB. In order to define the scope of the nested identifiers we introduce a marker (delimiter) $\$ to separate them from the ones of the outer range. This process can be defined more precisely as follows:

Algorithm A,

I. On the creation of a nested block the marker $\$ is augmented to STORE and ATTRIB.

II. The type identifiers and the types of the variables, functions etc. are augmented to ATTRIB, and others to STORE.

III. On reaching the end of the block, STORE and ATTRIB are trimmed from the rightmost element to the most recent marker $\$.

Therefore, at compile time the information from one marker $\$ to the other one in STORE and ATTRIB corresponds to a local table defined in the W-grammar of PSS (cf. 3.2.2).
Example 8: program ab;

A  procedure p;

B  function r : integer;

C  begin

D  end;

E  function q : real;

F  begin

G  end;

H  end.

Declarations of the program
Declarations of the subroutine p
Declarations of the subroutine r
Declarations of the subroutine q

The corresponding stages at compile time can be shown diagramatically as follows:

Stage A \{ ATTRIB is (α,γ,γab) \\
\{ STORE is (β,α,δab,p) \}

Stage B \{ ATTRIB is (α,γ,γab,φ,γp) \\
\{ STORE is (β,α,δab,p,φ,δp,r) \}

Stage C \{ ATTRIB is (α,φ,γab,φ,γp,γr) \\
\{ STORE is (β,α,δab,p,φ,δp,r,φ,δr) \}

Stage D \{ ATTRIB is (α,φ,γab,φ,γp) \\
\{ STORE is (β,α,δab,p,φ,δp,r) \}

Stage E \{ ATTRIB is (α,γ,γab,φ,γp) \\
\{ STORE is (β,α,δab,p,φ,δp,r,q) \}
Stage F
\[
\begin{align*}
\text{ATTRIB is} & \ (a, \hat{a}, yab, \hat{\psi}, y_p, \hat{\psi}, y_q) \\
\text{STORE is} & \ (\beta, A, \hat{\psi}, \delta ab, \hat{\delta} p, \hat{\phi}, \delta p, y, q, \delta q)
\end{align*}
\]

Stage G
\[
\begin{align*}
\text{ATTRIB is} & \ (a, \hat{\psi}, yab) \\
\text{STORE is} & \ (\beta, A, \hat{\psi}, \delta ab, p)
\end{align*}
\]

Stage H
\[
\begin{align*}
\text{ATTRIB is} & \ (a) \\
\text{STORE is} & \ (\beta, A)
\end{align*}
\]

where:
- \(a\) denotes the representation of the standard types.
- \(\beta\) denotes the representation of the standard subroutines and the standard constants.
- \(\gamma x\) denotes the representation of the type declarations and the type of the identifiers local to \(x\).
- \(\delta x\) denotes the representation of the identifiers (except the type identifiers) local to \(x\).

The lists \text{STORE} and \text{ATTRIB} also play the role of the run time stack. Here the marker \(\hat{\phi}\) represents both the beginning and the end of a frame of the stack. Also, \(\hat{\phi}\) is used to indicate the static and dynamic links. To define the mechanism of the stack, we have the following algorithm:

Algorithm B,

I. On the invocation of a subroutine the marker \(\hat{\phi}\) is augmented to \text{STORE} and \text{ATTRIB}.

II. The marker \(\hat{\phi}\) in \text{STORE} is linked to the end of the frame (signed by \(\hat{\phi}\) in \text{STORE}) which includes the representation of the subroutine declaration.

III. The identifiers local to the subroutine (including the parameters except type and constant identifiers) are augmented to \text{STORE}.

The type of those identifiers are also specified in \text{ATTRIB}.
IV. On termination of the subroutine, STORE and ATTRIB are trimmed from the right to the associated marker $\psi$.

Example 9: Suppose in Example 8, the program is constructed such that the following sequence of calls is executed:

\[
ab + p + q + r + r
\]

then STORE and ATTRIB may be represented below:

\[
\begin{align*}
\text{ATTRIB} & \text{ is } (\alpha, \phi, \delta ab, \delta p, \delta p, \delta q, \delta r, \phi, \delta r) \\
\text{STORE } & \text{ is } (\beta, \Lambda, \gamma ab, \rho, \gamma p, \gamma r, \phi, \gamma r, \phi, \gamma r, \gamma r)
\end{align*}
\]

where:

$\alpha$ and $\beta$ are the same as in Example 8.

$\gamma x$ denotes the representation of the identifiers (except types and constants) local to $x$.

$\delta x$ denotes the representation of the types of the identifiers local to $x$.

$\phi$ denotes the head of the frame associated with the function call $y$.

Finally, a PSS program and its meaning are defined provided that:

I. The process of constructing the syntax tree does not fail.

II. The semantic injections do not produce the word "halt".

Since a program is treated as a subroutine, we first consider subroutines and their calls.

5.2.2 Subroutine Declarations

The CPS structure for a subroutine declaration is associated with a sequence of Carabiner operations. These operations define the semantics of the subroutine and its corresponding call. The following diagram represents a procedure and a function in CPS.
Here, F is the subroutine name, α denotes the structure of the formal parameters and β is a sequence of Carabiner statements which define the run time semantic requirements of the subroutine formal parameters and the subroutine block. In Figure 5.6(b) γ represents the type of the function.

To see in more detail the definition of a subroutine in CPS at compile time, consider the following examples:

Example 10:

```
program ab;
  var i: real;
  function Bisect(function f:real ;a,b: real ; var z: real):integer;
  var m: real;
  procedure abs(n: real);
  begin
    ...
  end;
begin
  ...
  a:=f(i);
  ...
  end;
begin
  i := -1;
  i := Bisect(odd, i+1, i, i)
end.
```

The state of STORE at the stage A is illustrated in Figure 5.7.
function ∈ OBJECT           formal variable ∈ OBJECT
variable ∈ OBJECT          formal function ∈ OBJECT     formal value ∈ OBJECT

real ∈ ATTRIB             integer ∈ ATTRIB

STORE is(..., A is( ), $, i, Bisect is(f is(#LATER), a, b, z))

BODY
is
log($) ;
at($) ;
.....

FIGURE 5.7
In Figure 5.7 the list Bisect (in STORE) and the type integer (in ATTRIB) corresponds to the structures \( a \) and \( \gamma \) in Figure 5.6 respectively. The word \#LATER in the list \( f \) shows that the parameters of the function \( f \) have not yet been defined. These will be specified at the first corresponding call.

Since the formal parameters of a subroutine and its other local identifiers should be in the same level in CPS, Figure 5.8 for stage B of Example 8 shows that the parameters are copied after the right most \( \checkmark \) in STORE.

The internal representation of the compound statement forming the subroutine body is a sequence of Carabiner codes, included in BODY of the subroutine.

When the compilation process is finished the state of STORE and ATTRIB are as shown in Figure 5.4 and 5.5, except that \( kA \) consists of a sequence of the program execution codes. The run time structure of each subroutine corresponds to a set of Carabiner operations included in \( kA \). The following diagram illustrates the structure of the function Bisect in the run time stack (i.e. STORE and ATTRIB). The symbol \( \omega \) is used throughout this report to denote an unassigned value.
formal function \in \text{OBJECT}

formal value \in \text{OBJECT}

formal variable \in \text{OBJECT}

variable \in \text{OBJECT}

real \in \text{OBJECT}

\text{STORE \is (... , \phi , i , \text{Bisect \is (} \text{f \is (} \#\text{LATER}\text{), a, b, z}\text{)} , \phi , \text{f \is (} \#\text{LATER}\text{), a, b, z, m, abs \is (n)}\text{)}\text{)}

\text{BODY \is } \\
\begin{align*}
&... \\
&... \\
&...
\end{align*}
5.2.3 Subroutine Invocation

Our CPS and W-grammar models of PSS are different in that in CPS, the type checking process is defined at compile time. This necessitates that, whenever a subroutine is declared as a formal parameter, the numbers and the types of its formal parameters are determined, by reference to its first call at the compilation stage.

Example 11: Suppose f(i) is the first call of f in stage C of Example 10, then the structure representing Bisect in Figure 5.7 is changed to that depicted in the following diagram.

```
STORE is (...\, i, Bisect is (f is (x), ...)...) 
```

The formalism associated with the invocation of a user defined subroutine requires an implementation of algorithm B in 5.2.1. To see this, we consider the following example.

Example 12: Diagram 5.9 illustrates the relevant section of STORE at the evaluation of: Bisect(odd ,i+1 ,i ,i)
at stage D of the Example 10, after loading the parameters.

```
STORE is (...odd is (x),...,\,i ,Bisect is (f ,a ,b ,z)...) 
```

The execution process of Bisect, i.e. \( e_{1k} \) Bisect
defines a frame on top of the stack and evaluates the internal codes associated with the main statement of the subroutine. The frame contains a frame head, i.e.:

```
# OUTPUT FROM Bisect
```

doing the formal parameters, the variable and the subroutine(abs) which is local to Bisect. (This situation is represented in Diagram 5.10).

The last two Carabiner operations in k Bisect remove the frame from the stack.

There are however, special cases such as forward subroutines, call and invocation of a routine, which is declared as a formal parameter of the other, and the standard subroutines. We conclude this section with detailed consideration of the forward subroutines, others are left for the reader to follow them from the corresponding rules and Macros.

A forward subroutine F is one whose head is declared but whose body is not declared until later in the current block. Diagramatically it is represented as:

```
STORE is (# FORWARD, ....... , F is ( ... ) ... )
```

```
TEMP is (# AHEAD)
```

The word #AHEAD shows that the body of F is declared later. When this is done, the list TEMP is replaced by a list BODY and #FORWARD and F are disconnected, i.e.:

```
STORE is (# FORWARD, .... , F is ( ... ))
```

```
BODY is (...) 
```

When the subroutine declaration part of the current block is parsed, the relation

```
k # FORWARD = ∅
```

shows that the body of each subroutine has been declared.
FIGURE 5.10
5.2.4 Constant Declarations

The structure associated with the Carabiner codes of a constant declaration is:

\[
\begin{align*}
\text{constant} & \in \text{OBJECT} \\
\beta & \in \text{ATTRIB} \\
<\text{identifier}> & \in \text{STORE}
\end{align*}
\]

where \( \alpha \) and \( \beta \) are the internal representation of the constant value and its type respectively. The structure in Figure 5.11 helps to clarify this.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5_11.png}
\caption{Figure 5.11}
\end{figure}

\begin{verbatim}
c = const a=-0.05 ; b=1.2 ; c='u'; d=true; e='xyz' ; e=f
\end{verbatim}
5.2.5 Type Declarations

The Carabiner structure corresponding to a type declaration is as follows:

\[ \beta \in \text{TYPE} \]
\[ \alpha \in \text{ATTRIB} \]

where \( \alpha \) and \( \beta \) represent the construction and the kind (subrange, scalar, array, etc.) of the type declaration. Since the information related to a type is needed at run time, the corresponding run time codes are also associated with the same structure. In this section, we restrict consideration to scalar, array and record types. To illustrate the motivation behind the definitions we give the following examples:

Example 13:  

```plaintext
program ab;
  type colour = (red, blue, yellow);
  var i : colour;
  ............
  ............
```

The figure below represents the Carabiner space after processing these declarations.

\[ \text{ATTRIB is } (\ldots \hat{\beta} \text{ colour is (DEF is(red,blue,yellow),WIDTH is (1}))} \]

Since the value identifiers (e.g. red, false, nil) and other identifiers in the current block are actually in the same level, both the value
identifiers and the others (except the type identifiers) are inserted in the same area of STORE. Note that the structure of a constant declaration and a value identifier are the same (see also nil, false, true in Figure 5.5). This technique is also used for the run time semantic checks.

Example 14: The structure of the indexed type:

```plaintext
type t = array[Boolean, (plus, minus, times)] of real
```

is illustrated in Figure 5.12. The symbol #T in the space is used to define an internal type identifier. The list #DIMEN consists of an integer representing the number of dimensions.

The structure associated with a record type declaration is:

```plaintext
record ∈ TYPE

r is (#F, a, #VP, β, #EVP, #EF, WIDTH is (n)) ∈ ATTRIB
```

where r is the type identifier, a and β are structures representing the fixed part and variant part (possibly empty) of the record, and n is an unsigned integer corresponding to the width of the record type. The reader should understand that, the identifiers of the fixed part and the ones of the variant part of a field list are in the same level. Therefore β, which involves the information of the variant part identifier is not modelled by a nested list.

Example 15: The diagram representing:

```plaintext
type r = record
    case s: integer of
      -1, 2: (x, y: integer;
        u: char
      );
      5, 7: (z: Boolean)
    end
```

is presented in Figure 5.13.
Items from TYPE: scalar array type identifier scalar type identifier

ATTRIB \texttt{is}(...,real...,Boolean,...),

\texttt{t is(DIMEN is(2),#T,#T is(DEF is(plus,minus,times),WIDTH is(1)),#T,WIDTH is(6))}

FIGURE 5.12
5.2.6 Expressions

A PSS expression is associated with a sequence of Carabiner codes, generated during the compilation stage. The activation of these codes causes the production of a temporary variable #STEMP in STORE which is linked to the result of the expression. The internal type of the expression is also linked to #STEMP. We indicate how this is done by the following example.

Example 16: program ab;
var B:array[(red,blue,white)] of integer
begin
   B[succ(red)]:=5
end.

The corresponding state of STORE, before and after the evaluation of the indexed expression:

succ(red)

are illustrated in Figure 5.14. The extra space after the rightmost $ in Figure 5.14b is equivalent to the extra local table mentioned in case B of 3.2.2.5.1. The type checking process of this kind of index type also creates a similar extra area in STORE.

5.2.7 Statements

In a way similar to 3.2.2.7, we now consider some aspects of the assignment statement. The procedure statement (call) is explained in 5.2.3. The CPS definition of structured statements is fairly straightforward and is not discussed in detail. Here we try to give an idea of the structure of the heap (list HEAP) and show how the state of CPS is changed during the assignment of heap items at run time. Again we give an example.
Items from TYPE:

array  scalar  type identifier

integer ∈ ATTRIB

#T is(#DIMEN is(1), #T is(DEF is(red, blue, white), WIDTH is(1)), #T,
WIDTH is(3)) ∈ ATTRIB

variable ∈ OBJECT

constant ∈ OBJECT

STORE is(..., A, B, ..., red, blue, white, #STEMP)

M is(ω, ω, ω)  red blue white

(a)

Items from TYPE:

array  scalar  type identifier

integer ∈ ATTRIB

#T is(#DIMEN is(1), #T is(DEF is(red, blue, white), WIDTH is(1)), #T,
WIDTH is(3)) ∈ ATTRIB

variable ∈ OBJECT

constant ∈ OBJECT

STORE is(..., A, B, ..., red, blue, white, #STEMP)

M is(ω, ω, ω)  red blue white blue

(b)

FIGURE 5.14
Example 17: 

```pascal
program ab;
    type r=...;
    var x,y,z:tr;
    begin
        x:=nil;
        new(y,z);
        z.t.s:=5
    end.
```

where \( r \) is the record declared in Example 15. The final state of \( \text{STORE} \) and \( \text{HEAP} \) is depicted in Figure 5.15. The memory units of \( x \) and \( y \) point to their corresponding first memory unit on the heap. Note that since \( s \) is a tag-field identifier, not only the memory unit of \( s \) but also its next one (associated with the extra tagfield, i.e. \#EXTRA) is assigned by 5 (compare to Figure 3.13).

5.2.8 Input and Output

Since the working space corresponding to input and output files in this model consists of two lists rather than two strings of characters (\( W \)-grammar), the input and output handling in this model is simple. Diagramatically:

`\text{INPUT} \text{is} (a_1,a_2,...,a_n,#EOF)`

`\text{OUTPUT} \text{is} (a_1,...,a_n)`

where \( a_i \) is either a value or \#EOLN to define the end of line. The symbol \#EOF signals the end of input file. Reading from the input stream is equivalent to a left deletion of \( \text{INPUT} \).

Example 18: If the structure of \( \text{STORE} \) and \( \text{INPUT} \) at run time is as follows:

```
real
STORE is (...)i,...)
M is(\( \omega \))
```

`\text{INPUT} \text{is} (#EOLN,+1.2,a,b)`

then after the activation of:
read (i)

it is changed to:

real

STORE is (....,i,...)

M is (+12E-2)

INPUT is (a,b)

r is (....,#CASE,s,#OFFSET is(0),#EXTRA,#OFFSET is (1),#T,#SWITCH,...)£ATTRIB

Pointer£TYPE

type identifier£TYPE

#T is (#T,WIDTH is (1))

STORE is (....,A,b,x,y,z)

M is (ω) M is (ω) M is (ω)

 HEAP is (nil,ω,ω,ω,ω,ω,ω,+5,+5,ω,ω,ω)

FIGURE 5.15
CHAPTER 6

CONCLUSIONS
This chapter begins with the comparison of the W-grammar and CPS systems and their corresponding PSS models. We then suggest ways in which the models can be modified. Since the manipulation of the PSS W-grammar and the primitive operations in CPS depend on string processing, the implementation of the models from the pattern matching point of view is considered. In the later part, a system which is based on W-grammars and CPS system is introduced. Finally, a summary of the material covered by this thesis is presented.

6.1 COMPARISONS

6.1.1 Comparison of the Systems

The W-grammar and CPS can partly be compared as follows:

I. W-grammar system is based on the familiar notion of context free grammars. Therefore, the mechanism of the system can easily and quickly be understood. The understanding of the CPS system requires knowledge of many elementary concepts. The whole of Chapter 4 and half of the Chapter 5 are devoted to the fundamental definitions for the CPS system whilst only a relatively small portion of Chapter 3 is needed in order to be able to understand W-grammars.

II. To test a context sensitive condition by a W-grammar we must simultaneously follow protonotions generated by metanotions in the same hyper rule keeping in mind many possible replacements and combinations. But in the CPS system several macro rules are applied one at a time, that is, upon the termination of a macro rule the next one will be applied. This is a great advantage of this system.

III. The use of set operators $\in, \subseteq, \cap$, etc. is familiar to the reader. Moreover each set theoretical operation is equivalent to several meta-productions and hyper rules in the W-grammar system. This is one of
the reasons why the W-grammar definition of a programming language is usually bigger than its corresponding formalisation in the CPS system.

6.1.2 Comparison of the PSS Formalisms

The comparison of our two PSS formal definitions is based on the techniques which we have used to define the requirements of the language. Since various methods may be used to define the PSS by these two systems (e.g. different internal representation, working space, etc.), different types of comparison might be made. Our PSS formalisms can be partially compared as follows:

I. Since the syntax specification of the programming languages in BNF is a well known method, the syntax of PSS in CPS is more comprehensible and readable than the other one.

II. We believe that the isolation of the context free from the context sensitive requirements of a language is important for the purpose of clarification. In this sense, the CPS definition of the language is better.

III. Since a hypernotion is written as a pseudo-English sentence, a semantic action can be understood from the content of its corresponding hypernotion. But this may not be understood from the name of a macro rule in the CPS model. However, it is possible to write rather long phrases instead of names of macro rules to resolve this problem.

IV. Neither of our models give explicit indication of the errors in a program. But a semantic error in CPS is reported by the word "halt" whilst a syntax error in this model or any error in the
other model stops the process of constructing the tree, leaving no error message. However, it is possible to add error indication to both models, by appropriate sentences.

V. The internal representations of an identifier and a number in CPS model are in terms of their high level language representations. But in the other model they are shown differently (e.g. -3 is shown by uminus one one one and the identifier AB2 by letter A letter B letter 2).

VI. To follow the definition of the primitive operations in the CPS model a background of the mechanism of the Extended Markov algorithms is required. But these definitions are straightforward in the other model. They can be followed from their corresponding hyper rules.

6.2 MODIFICATIONS TO PSS FORMALISMS

6.2.1 W-grammar

The PSS W-grammar can be improved to define the compatibility of the recursively defined types. It can also be modified to represent the numbers in their decimal forms. This causes modification of the rules corresponding to the arithmetic operations.

I. The compatibility of the recursive types.

The following metaproductions and hyper rules are defined based on the solution to the structural type compatibility mentioned in 2.1.1.

PAIRSETY::PAIRS;EMPTY
PAIRS::PAIR;PAIRS PAIR.
PAIR::(IDEN,IDEN).

The protonotion generated from PAIRS is a sequence of PAIR, where each IDEN (defined by the following hyper rule) corresponds to a type identifier and the table number, which contains the location of this type
identifier. Therefore $PAIRS$ represents the set mentioned in 2.1.1.

The hyper rule $H_{ty20}$ is replaced by the rules:

$H020$ \quad \text{TABLES the TYPE and TYPE1 are identical \textit{PAIRSETY}:}$

- \quad \text{the type identifiers TYPE and TYPE1 would be the same as IDEN and IDEN1 respectively,}$
- \quad \text{where (IDEN, IDEN1) is not in \textit{PAIRSETY},}$
- \quad \text{TABLES the implicit TYPE and TYPE1 give the explicit TYPE2 and TYPE3,}\@ty18
- \quad \text{TABLES the explicit TYPE2 and TYPE3 are identical (IDEN, IDEN1) \textit{PAIRSETY},}\@ty21
- \quad \text{TABLES the implicit TYPE and TYPE1 give the explicit TYPE2 and TYPE3,}\@ty18
- \quad \text{TABLES the explicit TYPE2 and TYPE3 are identical \textit{PAIRSETY}.}\@21

the type identifiers IDEN and IDEN1 would be the same as IDEN2 and IDEN3:

- \quad \text{where IDEN2 is IDEN,}$
- \quad \text{where IDEN3 is IDEN1.}$

This rule confirms that both types are of type identifiers. The modification of the last hypernotion of $H020$ above, causes the addition of the metanotion \textit{PAIRSETY} to the left hand side hypernotion and some of the right hand side hypernotations of the rules 21, 23, 25-36 and 41 of the section TY of Appendix 1. This is left to the reader.

II. Numbers in decimal representation.

The representation of a number by a sequence of the strings "one"s (and possibly including some of the strings uminus, uplus, zero for $-\,+,0$ and the character $E$ for the exponential part of a real number) is not suitable and practical. Therefore, it is better to represent the numbers in their decimal notations. This changes the definition of the arithmetic operations. Since all these operations are based on the definition of the successor and predecessor functions for integer, we define the successor function. Other definitions are left to the reader. In the following definition, we assume that the integer is in the range $[-\text{MAXINT},+\text{MAXINT}]$.
and also the unsignificant digits has been removed from the number.

\[
\text{MAXINT}: \text{DIGIT DIGIT...DIGIT. \{n-times\}} \\
\text{DIGIT}: 0;1;2;...;9. \\
\text{DIGITSETY: DIGITS;EMPTY.} \\
\text{DIGITS: DIGIT;DIGITS.}
\]

The corresponding hyper rules are defined as follows:

- successor of \text{DIGITSETY} \text{DIGIT} would be \text{DIGITSETY1 DIGIT1:}
  - where \text{DIGIT} differs from 9,
  - where \text{DIGITSETY} \text{DIGIT} differs from \text{MAXINT},
  - add one to \text{DIGIT} to give \text{DIGIT1},
  - where \text{DIGITSETY1 is DIGITSETY};
  - where \text{DIGIT} is 0,
  - where \text{DIGITSETY} \text{DIGIT} differs from \text{MAXINT},
  - where \text{DIGIT1 is 1},
  - \text{DIGITSETY1 may give the successor DIGITSETY1}.

- add one to 0 to give 1:true.
- add one to 1 to give 2:true.
- ...
- add one to 8 to give 9:true.

\text{DIGITSETY} may give the successor \text{DIGITS:}
- where \text{DIGITSETY is EMPTY},
- where \text{DIGITS is 1;}
- successor of \text{DIGITSETY} would be \text{DIGITS}.

6.2.2 CPS

The CPS model should be modified for the compatibility of the
recursively defined types and the full specification of subroutine formal
parameters.

I. Since the operations of the CPS system are based on the set
theoretical operators, the formal definition for the compatibility
of recursive types only modifies the Macro rules 209 and 82 of
Appendix 2. The rule 209 is changed to:

\[
\text{typecheck}(a, b) = \text{augl}(\text{STORE, } \#\text{PAIRSET}); \\
\text{gilk}(\#\text{PAIRSET}, \text{m STORE}), \text{PAIRS is ( );} \\
\text{doll(STORE, } \#\text{PAIRSET)}
\]
and the rule 82 to:

\[
\text{identical}(\alpha, \beta) = \\
\text{if}(e(k - \text{TYPE}) \text{ eq type identifier}) \cap \\
(e(k - \text{TYPE}) \text{ eq type identifier}) \\
\text{then} \\
\text{if } \text{PAIR is}(\alpha \text{ of ATTRIB}, \beta \text{ of ATTRIB}) \neq \\
1k \#\text{PAIRSET lm STORE} \\
\text{then} \\
\text{augr}(k(#\text{PAIRSET lm STORE}), \\
\text{PAIR is}(\alpha \text{ of ATTRIB}, \beta \text{ of ATTRIB}) \\
); \\
\text{else halt ,}
\]

where \(\Gamma\) and \(\Theta\) are the sequences of the Carabiner codes on the right hand side of the macro rules 209 and 82 in the Appendix 2, respectively.

II. The compile time specification of the number and the types of the formal parameters of a subroutine parameter in the present formalism is complex and not general (cf. Example 7 of 2.1.1.6). Therefore, the model should be modified to define these requirements at run time. To do this, the complex related rules, starting from the BNF rule 70 is replaced by more suitable rules such that we have the type checking process between the actual and formal parameters at run time. These rules already exist in different places in the model and the reader may use them to improve the present formalism.

We again, strongly recommend that to have an efficient compile time definition for the full specification of the subroutine formal parameters, it is better to improve the language to have an explicit declaration of the number and the types of the parameters for this type of formal parameter (cf. 2.1.1.6, II).
6.3 IMPLEMENTATION CONSIDERATIONS

Since the implementation of the PSS W-grammar definition and the primitive operations of the CPS definition of PSS are based on string manipulation, here we consider some pattern matching methods. The implementation of the PSS formalisms is beyond the scope of this work.

A simple way to find a pattern (sub-string) in a text string is to start from the i-th, i=1,2,..., character of the text and check whether the first, second,..., characters of the pattern is matched with the i-th, (i+1)-th,... characters of the text respectively. This method can be very inefficient (slow). In recent years a number of pattern matching algorithms have been designed. Here we consider the ones of Aho-Corasick [37], Boyer-Moore [38] and Knuth-Morris-Pratt [39]. The algorithm of [37] is designed to look for a number of sub-strings in a text. The method is to design a finite state machine from the sub-strings. The finite state machine is then used to find each sub-string and its position in the text. Since only one pattern will be sought in the implementation of our formalisms at each time, this algorithm is not suitable. This is because the construction of a finite state machine for a single pattern is time consuming. Moreover, the process of pattern matching is based on the simple method above and so it is not efficient.

In the Boyer-Moore algorithm the search begins by looking for the last character of the pattern in the text string. Since this method is similar to the one of Knuth-Morris-Pratt, we explain this algorithm. In this method, there is an expression $j\text{-next}[j]$ which we name $\text{kdelta}_2(j)$, i.e.

\[
\text{kdelta}_2(j) = j\text{-next}[j],
\]

where $\text{next}[j]$ is the position of the next current character in the pattern after a mismatch at the $j$-th position of the pattern. In this algorithm the search starts from the left most character and moves towards the
right of the pattern. We claim that if the following function, kdelta_1 is included in the algorithm, it works faster. Let α be an arbitrary character then:

\[ kdelta_1(\alpha) = \begin{cases} j & \text{if } \alpha \text{ does not occur in the pattern and } j \text{ is the position of its current character in the matching process.} \\ 0 & \text{otherwise.} \end{cases} \]

The Knuth-Morris-Pratt method and our improvement (using delta_1) are illustrated by the following example:

**Example 1:** To compare the two methods we consider the example of [39], with a slight modification in their text string.

\[ \text{text} = a\ b\ c\ a\ x\ a\ b\ c\ a\ x\ b\ c\ a\ b\ c\ a\ c\ a\ b\ c\ \]

The pattern, next[j] and kdelta_2(j) is also defined as:

\[ j = 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10 \]
\[ \text{pattern} = a\ b\ c\ a\ b\ c\ a\ c\ a\ b \]
\[ \text{next}[j] = 0\ 1\ 1\ 0\ 1\ 0\ 5\ 0\ 1 \]
\[ kdelta_2(j) = 1\ 1\ 2\ 4\ 4\ 5\ 7\ 3\ 9\ 9 \]

Suppose we have the following situation:

\[ \text{pattern: } a\ b\ c\ a\ b\ c\ a\ c\ a\ b \]
\[ \text{text: } a\ b\ c\ a\ x\ a\ b\ c\ a\ b\ c\ a\ x\ \ldots \]

\[ \uparrow \]

There is a mismatch. Based on the reasoning of their method the pattern should be moved 4 positions (since \( j=5 \) and \( kdelta_2(5)=4 \)) over the text so that 'a' (first character) aligned with x. In our method the pattern should be slid n positions to the right, where:

\[ n = \max(kdelta_1(x), kdelta_2(5)) = \max(5, 4) = 5 \]

N.B. Since the current character of the pattern is the one over x (indicated by \( \uparrow \)), therefore \( j=5 \).
After moving the pattern over the text by our method we have:

\begin{align*}
\text{pattern:} & \quad a \ b \ c \ a \ b \ c \ a \ c \ a \ b \\
\text{text:} & \quad b \ a \ b \ c \ x \ a \ b \ c \ a \ b \ c \ a \ x \ b \ c \ a \ b \ c \ a \ b \ c \ ...
\end{align*}

But their method presents:

\begin{align*}
\text{pattern:} & \quad a \ b \ c \ a \ b \ c \ a \ c \ a \ b \\
\text{text:} & \quad b \ a \ b \ c \ x \ a \ b \ c \ a \ b \ c \ a \ x \ b \ c \ a \ b \ c \ a \ b \ c \ ...
\end{align*}

Here, we have a mismatch at the first character. Since $k_{\Delta Z}(l) = 1$, the pattern is moved by one position and the situation will be the same as (I).

Again there is a mismatch in (I). Since $k_{\Delta Z}(8) = 3$, by their method we have:

\begin{align*}
\text{pattern:} & \quad a \ b \ c \ a \ b \ c \ a \ c \ a \ b \\
\text{text:} & \quad b \ a \ b \ c \ x \ a \ b \ c \ a \ b \ c \ a \ x \ b \ c \ a \ b \ c \ a \ b \ c \ ...
\end{align*}

But by our method the pattern is moved on by:

\[
\max(k_{\Delta Z}(x), k_{\Delta Z}(8)) = \max(8, 3) = 8
\]

positions and the situation is:

\begin{align*}
\text{pattern:} & \quad a \ b \ c \ a \ b \ c \ a \ c \ a \ b \\
\text{text:} & \quad b \ a \ b \ c \ x \ a \ b \ c \ a \ b \ c \ a \ x \ b \ c \ a \ b \ c \ a \ x \ b \ c \ a \ b \ c \ ...
\end{align*}

Here, the pattern is moved by:

\[
\max(k_{\Delta Z}(b), k_{\Delta Z}(8)) = \max(0, 1) = 1
\]

position and we have:

\begin{align*}
\text{pattern:} & \quad a \ b \ c \ a \ b \ c \ a \ c \ a \ b \\
\text{text:} & \quad ... \ a \ x \ b \ c \ a \ b \ c \ a \ b \ c \ a \ c \ a \ b \ c \ ...
\end{align*}

If the pattern is again slid by:
position we have a complete match as follows:

**pattern:** a b c a b c a c a b

**text:** ... a x b c a b c a b c a c a b c

Note: For a mismatch there is no need to search for the current character in the pattern. Instead, before starting the matching process we can collect the characters of the pattern in a set and define the function $k\delta_1$.

### 6.4 A HYBRID SYSTEM

As noted, the use of Macros in the CPS system is not readily understood from their names and the use of EMA's requires extra preliminary work to be done. We suggest a new system which is a mixture of the CPS and $W$-grammar systems to overcome these conflicts. It has the same Carabiner working space and operators. The primitive operations are defined by hyper rules rather than the Extended Markov algorithms. Therefore, for most of the semantic actions, the CPS syntax tree is extended. The reader might think that we have done nothing but the replacement of the Markov algorithms by $W$-grammar definitions. He might also say we still need the same amount of background (more or less) to appreciate the working of the system. To a certain extent this is correct, however, this system would be more readable. The conditional Carabiner statements (cf. 5.1.2.6) can be defined in much more readable $W$-grammar definitions than a large, complex and difficult to understand string, consisting of lots of $S$ operators.

To have a consistent form in this system, the metaproductions may
be presented in BNF style. The Macro rules 1, 2 and 3 of Appendix 2 can be defined as follows:

I. The corresponding metaproductions:

   A. The corresponding metaproductions:

      <pointer> ::= <auxiliary> prev(<pointer>)
      <auxiliary> ::= #AITEMP #STEMP
      <set> ::= [<elementset>]
      <elementset> ::= <element> | <empty>
      <element> ::= <string> | <elements>
      <Bool> ::= [false, true]
      <string> ::= <string> | <empty>

   B. The corresponding hyper rules

      A. The first Macro rule is changed to:

         1. adjust <pointer> to the first element to which pointer is linked:

            \textit{adj}(<pointer>, k<pointer>, \#1 of k<pointer>) \textcircled{1}

      B. The second Macro rule is changed to:

         2. adjust both <pointer> and <pointer>:

            adjust <pointer> to the first element to which pointer is linked \textcircled{2}
            adjust <pointer> to the first element to which pointer is linked \textcircled{1}

      C. The third Macro rule is defined as:

         3. <pointer> is the result of the type checking of operand

            <pointer> the and operator operand <pointer>:
             \textit{e}(k<pointer>) \textcircled{2}
             \textit{e}(k<pointer>) \textcircled{2}
             \textit{adj} both <pointer> and <pointer> \textcircled{1}
             where \textcircled{1} is <Bool> \textcircled{1}
             where \textcircled{1} is <Bool> \textcircled{1}
             link (<pointer>, Boolean attr \textcircled{1} \textcircled{1} \textcircled{1})

*The symbols , , and ; in hyper rules are replaced by \textcircled{1}, \textcircled{2} and \textcircled{3} respectively.
4. `<string>belongs to [<elementsety>]`:
   
   where, `<string>`, is in , `<elementsety>`, \(\varnothing\)

5. `where <string> is in <stringety><string><stringety>` true \(\varnothing\)

6. `true:<empty> \(\varnothing\)`

6.5 **SUMMARY**

The language PASCAL has been found to be imprecise because some features (e.g. type compatibility, subrange types, free variant records and subroutine formal parameters) of the language are not well-defined and for these some suggestions have been made. The lack of an accepted formalism creates different (probably erroneous) interpretations in specific implementations. The W-grammar system is introduced and a full formal definition for a fairly large PASCAL subset (PSS) is given and its main features are described. A second system (CPS) is described and used to give a full formalism for PSS. Both systems and their formalisms are compared from different points of view and a new (hybrid) method based on W-grammars and CPS is proposed.
REFERENCES


[31] PASCAL COMPILER #PAS Q/2A (SUSSEX VERSION 005).


APPENDIX 1

W-GRAMMAR DEFINITION OF PSS
Metaproductions

M001 ADDOP::plus|minus|or|and.
M002 ADDRESS::STKHP ONESh TAGETY box.
M003 ALDIG::ALPHA|DIGIT.
M004 ALPHA::A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z.
M005 AMOUNT::OBJECT of KIND.
M006 AROP::PMTD|mod.
M007 ARRAY::ONES dimensional INDEXES of TYPE.
M008 BNAME::left NAME right.
M009 BNAME5::BNAME IS NAME SNAMES.
M010 BNAME5ETY::BNAME IS EMPTY.
M011 BOOLEAN::letter b letter o letter a letter l letter e letter a letter n.
M012 BOOLPAIR::left FALSE right left TRUE right.
M013 BOOLTYPE::SCBOOLTYPE; RGBOOLTYPE.
M014 DOOP::orland.
M015 DOP::AROP|DOOP|REALTIONOpIn.
M016 DBOX::undefined|DBOX.
M017 EUINT::left UINT right.
M018 EUINTs::BUINT|BUINT|BUINTs.
M019 EUINTSETY::BUINTS IS EMPTY.
M020 CASEELEMETY::CASEELEM IS EMPTY.
M021 CASEELEM::LABLE STMTETY.
M022 CASEELEMS::CASEELEM IS EMPTY.
M023 CASEELEMS::CASEELEMS IS EMPTY.
M024 CHAR::ALDIO|SYMBOL.
M025 CHARACTER::letter c letter h letter a letter r.
M026 CHARs::CHAR|CHAR|CHAR.
M027 CHARSETY::CHARS IS EMPTY.
M028 CHARTYPE::SCCHARTYPE; RGCHARTYPE.
M029 CHR::letter c letter h letter r.
M030 COMM::comma EXP ammoc, comma DNAME ammoc.
M031 COMMAS::COMMA IS COMMAS.
M032 COMMASETY::COMMAS IS EMPTY.
M033 COMPVAR::INDEXVAR|FDVAR.
M034 CONDITION::INFILE OUTFILE with STACK and HEAPETY.
M035 CONST::KONSTANT|STRING.
M036 DBOX::KONSTANT|SET|ADDRESS|DNAME.
M037 DECIMAL::one one…one {d_times}.
M038 DIGIT::0|1|2|3|4|5|6|7|8|9.
M039 DIGITS::DIGIT|DIGIT DIGITS.
M040 DIGITSETY::DIGITS IS EMPTY.
M041 DISPOSE::letter d letter i letter s letter p letter o letter r.
M042 DNAME::NAME IS NAME.
M043 DOTO::toldownto.
M044 ELEMENT::elem EXP elem|elem EXP dot dot|elem.
M045 ELEMENTs::ELEMENT|ELEMENT ELEMENTs.
M046 ELEMENTSETY::ELEMENTS IS EMPTY.
M047 ELSE::else STMTETY.
M048 ELSEETY::ELSE IS EMPTY.
M049 EMPTY::
M050 ENTVAR::NAME.
M051 EOF::letter e letter o letter l
M052 EOLN::letter e letter o letter l letter n.
M053 EONOTEQ::equal to unequal to.
M054 EXPON::E T.
M055 EXPONEY::EXPONE EMPTY.
M056 EXP::SEXPS EXP RELOP SEXP.
M057 EXTRA::extra IDEN with OFFSET endext.
M058 FACTOR::VARIABLE::CONST open bracket EXP close bracket
SET inst FACTOR if function SUBCALL call.
M059 FALSE::letter f letter a letter l letter s letter e.
M060 FDVAR::field VARIABLE point NAME select.
M061 FIELDLIST::SECTIONS:: SECTIONS VARPART IVARPART.
M062 FLOC:: loc NAME refers NBOSEXESY end.
M063 FLOCS::FLOC I FLOCS.
M064 FLOCS::I FLOC FLOCS.
M065 FORMALTYPE::formal type TYPE I EMPTY.
M066 FORMAL::SHAPE of TYPE.
M067 FORM::SCGi array record set pointer.
M068 FRAIM::HEADETY and FLOCSETS.
M069 FRAMESETS::FRAMES I EMPTY.
M070 FRAMES::FRAME IFRAME FRAMES.
M071 FRAME::the ONES th frame from ONES table FRAIM link to ONESETY frame.
M072 FUNPROC::function I procedure.
M073 HEADETY::HEAD I EMPTY.
M074 HEAD::the block NAME be FUNPROC refers NBOXETY.
M075 HEAPETY::NBOXES I EMPTY.
M076 IDEN::NAME in ONES table.
M077 INDEXES::INDEXES I EMPTY.
M078 INDEXES::INDEX I INDEX INDEXES.
M079 INDEX::left TYPE right.
M080 INDEXVAR::matrix VARIABLE indices COMMAS array.
M081 INFILE::KCHARSETY end of UINT file ONESETY chars in a line.
M082 INLINELEN::i one one...one. {il_times}
M083 INTABETY::at ONES table I EMPTY.
M084 INTEGER::letter i letter n letter t letter e letter g
letter e letter r.
M085 INTERVAL::integer from INT to INT.
M086 INFFORMAT::i one one...one. {inf_times}
M087 INTVALEY::INTERVAL I EMPTY.
M088 INTTYPE::SCINTTYPE I RGINTTYPE.
M089 INT::UOP UINT.
M090 KARSET::charachar b char c char d...........char.
M091 KCHAR::char CHAR.
M092 KCHARS::KCHARS I EMPTY.
M093 KCHARS::KCHAR KCHAR KCHAR.
M094 KIND::CONSTITYPE I VAR I PROCFD I DEC I FORMAT.
M095 KONST::konst VALUE tsnok.
M096 KONSTSETS::KONSTSETS I EMPTY.
M097 KONSTANTS::KONST I KONST KONST.
M098 KONSTORSETS::KONSTORSETS.
M099 LABEL::label KONSTANTS end lab.
M100 LEFTETY::in left side I EMPTY.
M101 LEOROEllless or equal to greater or equal to.
M102 LETTER::letter ALDIG.
M103 LETTERSET::LETTERSET EMPTY.
M104 LETTERS::LETTER LETTER LETTERS.
M105 LOC::loc NAME be AMOUNT end.
M106 LOCSET::LOCSET EMPTY.
M107 LOCS::LOC LOC LOCS.
M108 LORD::less than greATER than.
M109 MAXINT::one one one one. \{m\_times\}
M110 MAXLENID::LETTER LETTER LETTER LETTER.
M111 MAXOUT::one one one. \{of\_times\}
M112 MAXSCALNO::one one one one. \{ms\_times\}
M113 MAXTEGRAL::one one one one. \{max\_times\}
M114 MINTTEGRAL::one one one one. \{min\_times\}
M115 MULOP::times intdiv intdiv %mod mod and.
M116 NAME::letter ALPHA LETTERSET.
M117 NAMETY::NAMEIEMPTY.
M118 NBXESET::NBOXES EMPTY.
M119 NBOXES::NBOX INBOX NBOXES.
M120 NBOXET::NBOXEMPTY.
M121 NBOX::the ONES st BOX ht.
M122 NEW::letter n letter e letter w.
M123 NUMBER::UOP UNUMBER.
M124 OBJECT::constant type variable FUNPROC formal parameter.
M125 ODD::letter o letter d letter d.
M126 OFFSET::offset UINT.
M127 ONESET::ONES EMPTY.
M128 ONES::one one ONES.
M129 OR::letter o letter r letter d.
M130 OULINLEN::one one one one. \{ol\_times\}
M131 OUTFILE::KCHARSET with ONESET chars in a line and ONESET total chars.
M132 PARNUMETY::PARNUM EMPTY.
M133 PARNUM::ONES parameters.
M134 PM::plus minus.
M135 PMTDD::PMT intdiv.
M136 PMT::PMT realdiv.
M137 PMT::plus minus times.
M138 POINTER::IDEN.
M139 PRED::letter p letter r letter e letter d.
M140 PROCDFDEC::PARNUMETY type TYPE and ONES as its local table USAGEETY.
M141 RADIX::one one one one. \{r\_times\}
M142 READ::letter r letter e letter a letter d.
M143 READLN::letter r letter e letter a letter d letter l letter n.
M144 REALFORMAT::one one one \{real\_times\}
M145 REAL::letter r letter e letter a letter l.
M146 REALRANGE::real from minus MTEGRAL e minus DECIMAL to uplus MTEGRAL e minus DECIMAL.
M147 REALRANGETY::REALRANGE EMPTY.
M148 REALTYPEDEF::scalar defined by REALRANGE
M149 REALTYPE::REALTYPEDEF needs one boxes.
M150 RECORD::rekoard FIELDLIST end.
M151 REFVAR::reference VARIABLE arrow.
M152 RELOP::NOTEQ GE OR EQ in.
M153 RESULTETY::RESULTIEMPTY.
M154 RESULT::TYPADRES ICONSTANT.
M155 REAL:: UOP UREAL.
M156 BOOLTYPE:: subrange defined by BOOLPAIR needs one boxes.
M157 CHARTYPE:: subrange defined by KCHARS needs one boxes.
M158 INTTYPE:: subrange defined by INTERVAL needs one boxes.
M159 SIDE:: TYPADRES:: CONSTANT.
M160 SCALAR:: NAMES: KCHARS: REAL RANGE: INTERVAL.
M161 SCALAR:: IDENT: KCHARS: NUMBER.
M162 BOOLTYPE:: scalar defined by BOOLPAIR needs one boxes.
M163 CHARTYPE:: scalar defined by KARSET needs one boxes.
M164 INTTYPE:: scalar defined by integer from minus MAXINT to plus ONT needs one boxes.
M165 SCGR:: scalar subrange.
M166 SECTIONS:: SELECTIONS:: EMPTY.
M167 SECTIONS:: SELECTIONS:: SECTIONS.
M168 SELECTIONS:: SELECTIONS:: EMPTY.
M169 SELECTION:: selection NAME of TYPE with OFFSET tagged end sel.
M170 SELECTIONS:: SELECTIONS:: EMPTY.
M171 SELECTIONS:: SELECTIONS:: SELECTIONS.
M172 SET:: subset ELEMENTSET:: elems.
M173 TYPE:: elements of TYPE.
M174 EXP:: TERM:: UOP TERM:: SEXP ADDOP TERM.
M175 SHAPE:: value:: variable:: FUNPROC.
M176 SIGN:: + -.
M177 SIGNETY:: SIGN:: EMPTY.
M178 STACK:: FRAMES.
M179 STANDFUN:: left ORD right left right left PRED right left CHR right left EOLN right left EOF right.
M180 STANDPROC:: left READLN right left WRITE right left WRITELN right.
M181 STKHP:: stack:: heap.
M182 STMTA:: NAME in its ONES table:
  exit:
  assign VARIABLE becomes EXP stmt
  procedure SUBCALL call:
  if EXP then STMTETY ELSEETY fi:
  case EXP of CASEELEMSET:: esac:
  while EXP do STMTETY od:
  repeat STMTETY until EXP taper fi:
  for NAME becomes EXP through DOTO EXP do STMTETY od.
M183 STMTETY:: STMT:: EMPTY.
M184 STMTETY:: STMT:: EMPTY.
M185 STMTS:: STMTETY:: STMTETY STMTS.
M186 STMT:: STMTS:: STMTA.
M187 STRINGSET:: STRING:: KONST SET.
M188 STRING:: string KCHARS end.
M189 SUBCALL:: DNAME with actuals Commaset.
M190 SUCC:: letter s letter u letter c letter c.
M191 SYMBOL:: SIGN:: *
  / = ! & ; % & ;; ~ _ ; a ; t ; space.
M192 TABLEETY:: TABLES:: EMPTY.
M193 TABLES:: TABLE:: TABLE TABLES.
M194 TABLE:: the ONESt LOCETY link to ONESETY table.
M195 TAGETY:: tag field:: EMPTY.
M196 TAGFIELD:: SELECTIONETY EXTR.
M197 TAPI:: SCALAR ARRAY:: RECORD:: SETYPE:: POINTER.
M198 TERM:: FACTOR:: TERM MULOP FACTOR.
M199 TINT::SIGNETY UTINT.
M200 TNUMBER::SIGNETY UNUMBER.
M201 TRUE::letter t letter r letter u letter e.
M202 TYPADRES::ADDRESS of TYPE.
M203 TYPEDEF::FORM defined by TAIP.
M204 TYPE::TYPEDEF needs UINT boxes?IDEN?void.
M205 UINT::zeros?ONES.
M206 UNUMBER::UINT?UTREAL.
M207 UOP::plus?uminus.
M208 UREAL::UINT e INT.
M209 USAGEETY::body is ahead?EMPTY.
M210 UTINT::DIGITS.
M211 UNUMBER::UTINT?UTREAL.
M212 UREAL::UTINT dot UINT EXPONETY?UTINT EXPON.
M213 VALUE::SCALVALini1.
M214 VARIABLE::ENTVARI?COMPVARI?REFVARI.
M215 VARIANTETY::VARIANT?EMPTY.
M216 VARTYPE::VARIANTSET?EMPTY.
M217 VARY::VARIANTETY?VARIANTSETS VARTYPE.
M218 VARY::variant LABLE FIELDLIST endvar.
M219 VARPART::case TAGFIELD of VARY.
M220 VAR::TYPE.
M221 WORDDELMETERS::left letter a letter n letter d right...
       left letter a letter r letter r letter a letter y right.
M222 WRITE::letter w letter r letter i letter t letter e.
M223 WRITELN::letter w letter r letter i letter t letter e
       letter l letter n.
DE

Compile time semantics for declarations

H001 program:
program ,
elementary action for standard table TABLE ,@de2
TABLE standard functions or procedure give the oneth
LOCS slink to table TABLES and STMTS ,@de3
limitation on NAME identifier ,@sy1
i ,
the oneth LOCS slink to table TABLES find the last table
number ONES ,@de4
where LOC is loc be procedure of type void and ONES
one as its local table end ,
where TABLES1 is the oneth LOCS LOC slink to table
TABLES ,
TABLES1 the ONES oneth slink to one table from block
introduce TABLES2 BNAMESETY and STMTSETY with main
compound statement STMTSETY1 ,@de5
'
INFILE reset input file to give INFILE1.@fi1
TABLES2 elementary actions to transfer the standard
table the oneth LOCS LOC slink to table in STACK ,@fr1
TABLES2 create a new frame on STACK for the ONES one
table to produce STACK1 ,@fr10
TABLES2 with STMTS STMTSETY NAME in its one table
STMTSETY1 exit and INFILE1 with chars in a line and
total chars with STACK1 and to execute NAME in its one
table to give INFILE2 OUTFILE with STACK2 and
HEAPETY.@ex1

H002 elementary action for standard table the oneth loc
CHARACTER be type of SCCHARTYPE end loc BOOLEAN be type
of SCBOOLTYPE end loc INTEGER be type of SCINTTYPE end loc
REAL be type of REALTYPE end slink to table ,
true.

H003 the oneth LOCS slink to table standard functions or
procedure give TABLE1 the one oneth loc letter x be formal
parameter of value of SCINTTYPE end slink to one table and
ODD in its one table if left letter x right mod konst
uplus one one tsnok equal to konst uplus zero tsnok then
assign left ODD right becomes konst TRUE in one table
tsnok stmt else assign left ODD right becomes konst FALSE
in one table tsnok stmt fi exit ;
where TABLE1 is the oneth LOCS loc ODD be function of
one parameters type SCBOOLTYPE and one one as its local
table and slink to table ,

H004 TABLESETY the ONESt loc LOCSETY slink to ONESETY table find
the last table number ONES1 ;
where ONES1 is ONES.
H005 TABLESETV the ONESth LOCSETV slink to ONESETY table
BNAMESETV from block introduce TABLES BNAMESETY1 and
STMTSETY with main compound statement STMTSETY1:
TABLESETV the ONESth LOCSETY slink to ONESETY table
BNAMESETV constant part give TABLES2 BNAMESETY2 from
ONES table \(\alpha_{de6}\)
TABLES2 BNAMESETV2 type part give TABLES3 BNAMESETY3
from ONES table \(\alpha_{de63}\)
TABLES3 each pointer type is defined properly from ONES
table \(\alpha_{de60}\)
TABLES3 BNAMESETY3 variable part give TABLES4 BNAMESETY4
from ONES table \(\alpha_{de63}\)
TABLES4 BNAMESETY4 procedures or functions from ONES
table give STMTSETY and TABLES1 BNAMESETY1 \(\alpha_{de66}\)
TABLES1 from ONES table subroutine head and block is
declared \(\alpha_{de82}\)
begin,
TABLES1 stmt train STMTSETY1 from ONES table \(\alpha_{es1}\)
end.

H006 TABLESETV the ONESth LOCSETY slink to ONESETY table
TABLESETY1 BNAMESETV OBJECT part give TABLES BNAMESETY1
from ONES1 table:
where ONES is ONES1,
the syntax of OBJECT be checked \(\alpha_{sy96}\)
TABLESETV the ONESth LOCSETY slink to ONESETY table
BNAMESETY nonempty OBJECT location LOCs and BNAMESETY1
with TABLESETY2 \(\alpha_{de7}\)
where TABLES is TABLESETV the ONESth LOCSETY LOCs slink
to ONESETY table TABLESETY1 TABLESETY2;
where TABLES is TABLESETV the ONESth LOCSETY slink to
ONESETY table TABLESETY1;
where BNAMESETY1 is BNAMESETY.

H007 TABLES BNAMESETY nonempty OBJECT locations LOCs and
BNAMEs with TABLESETV:
OBJECT could only be constant or type \(\alpha_{de8}\)
TABLES BNAMESETY design OBJECT locations LOCs and
BNAMEs with TABLESETY \(\alpha_{de9}\)
where OBJECT is variable;
TABLES BNAMESETY consider groups of OBJECT locations
LOCs and BNAMEs with TABLESETY \(\alpha_{de67}\)

H008 OBJECT could only be constant or type:
where OBJECT is constant;
where OBJECT is type.
H009 TABLESETY the ONESth LOCSETY slink to ONESETY table BNAMESETY design OBJECT locations loc NAME be OBJECT1 of KIND end LOCSETY1 and BNAMEs with TABLESETY1 TABLESETY2 :
 limitation on NAME identifier ,asy1
 = ,
 where OBJECT1 is OBJECT ,
 where left NAME right is not in BNAMESETY ,
 TABLESETY the ONESth LOCSETY slink to ONESETY table BNAMESETY left NAME right obtain OBJECT of KIND and BNAMEs1 with TABLESETY1 ;ade10,25
 1 ,
 TABLESETY the ONESth LOCSETY loc NAME be OBJECT of KIND end slink to ONESETY table BNAMEs1 design OBJECT locations LOCSETY1 and BNAMEs with TABLESETY2 ;ade9
 where LOCSETY1 TABLESETY2 is EMPTY ,
 limitation on NAME identifier ,asy1
 = ,
 where OBJECT1 is OBJECT ,
 where left NAME right is not in BNAMESETY ,
 TABLESETY the ONESth LOCSETY slink to ONESETY table BNAMESETY left NAME right obtain OBJECT of KIND and BNAMEs with TABLESETY1.ade10,25

H010 TABLES BNAMEs obtain constant of CONST and BNAMEs1 with TABLESETY :
  where TABLESETY is EMPTY ,
  where BNAMEs1 is BNAMEs ,
 TABLES consider the constant of CONST.ade11

H011 TABLES consider the constant of CONST :
 TABLES consider the signed constant of CONST 13de12
 TABLES consider the unsigned constant of CONST.ade21

H012 TABLES consider the signed constant of konst UOP UNUMBER tsnok :
  UOP1 operation ;asy 97,98
  NAME identifier ;asy 2
  TABLES find the amount of old identifier NAME to be constant of konst UOP2 UNUMBER tsnok 13de13
  the number UOP UNUMBER is in the rigth range ,3co30,31
  the unary sign UOP1 and UOP2 make UOP 13de20
  plan konst UOP UNUMBER tsnok to be number might in integer from uminus MAXINT to uplus MAXINT.3co4

H013 TABLESETY the ONESth LOCSETY slink to ONESETY table find the amount of old identifier NAME to be AMOUNT INTABETY :
  where LOCSETY contains loc NAME be AMOUNT end ,
  INTABETY will be defined by ONES 13de14
  where loc NAME be is not in LOCSETY ,
  left NAME right should not be an scalar item of LOCSETY 13de16
  TABLESETY via ONESETY table reach TABLES 13de19
  TABLES find the amount of old identifier NAME to be AMOUNT INTABETY.3de13

H014 INTABETY will be defined by ONES :
 nonempty INTABETY will be defined by ONES 13de15
 where INTABETY is EMPTY.
H015 nonempty at ONES table will be defined by ONES1:
where ONES is ONES1.

H016 BNAME should not be an scalar item of LOCSETY:
nonempty LOCSETY does not involve scalar item
BNAME iade17
where LOCSETY is EMPTY.

H017 nonempty loc NAME be AMOUNT end LOCSETY does not involve
scalar item BNAME:
when AMOUNT is scalar then must not have BNAME iade18
BNAME should not be an scalar item of LOCSETY iade16
BNAME should not be an scalar item of LOCSETY.ade16

H018 when scalar defined by BNAMES needs one boxes is scalar
then must not have BNAME:
where BNAME is not in BNAMES.

H019 TABLESETY the ONESth LOCSETY slink to ONESETY table
TABLESETY1 via ONES1 table reach TABLES:
where ONES is ONES1:
where TABLES is TABLESETY the ONESth LOCSETY slink to
ONESETY table.

H020 the unary sign UOP and UOP1 make UOP2:
where UOP UOP1 UOP2 is uminus uminus uplus i
where UOP UOP1 UOP2 is uminus uplus uminus i
where UOP UOP1 UOP2 is uplus uminus uminus i
where UOP UOP1 UOP2 is uplus uplus uplus.

H021 TABLES consider the unsigned constant of CONST:
NAME identifier iasy2
TABLES find the amount of old identifier NAME to be
constant of CONST iade13
NAME identifier iasy2
TABLES search for scalar item of NAME in ONES table of
BNAMES iade12
where CONST is konst NAME in ONES table tsnok i
nil i
where CONST is konst nil tsnok i
uplus unsigned .number UTNUMBER produce UNUMBER integer
from uminus MAXINT to uplus MAXINT iaco5
where CONST is konst uplus UNUMBER tsnok i
plan CONST to be character of KARSET iaco3
plan CONST to be string.acy1

H022 TABLESETY the ONESth LOCSETY slink to ONESETY table search
for scalar item of NAME in ONES1 table of BNAMES:
NAME in BNAMES must be an scalar item of LOCSETY iade23
where ONES1 is ONES i
where loc NAME be is not in LOCSETY i
left NAME right should not be an scalar item of
LOCSETY iade16
TABLESETY via ONESETY table reach TABLES iade19
TABLES search for scalar item of NAME in ONES1 table of
BNAMES.ade22
H023 NAME in BNAMES must be an scalar item of LOC LOCSETY:
   BNAMES is an scalar definition of LOC ~ade24
   where BNAMES contains left NAME right:
   NAME in BNAMES must be an scalar item of LOCSETY.~ade23

H024 BNAMES is an scalar definition of loc NAMETY be type of
   scalar defined by BNAMES1 needs one boxes end:
   where BNAMES is BNAMES1.

H025 TABLES BNAMESETY obtain type of TYPE and BNAMESETY1 with
   TABLESETY:
   TABLES BNAMESETY simple type of TYPE and
   BNAMESETY1 ~ade26
   where TABLESETY is EMPTY:
   TABLES structured type of TYPE and give
   TABLESETY ~ade32,39,59
   where BNAMESETY1 is BNAMESETY:
   TABLES pointer of TYPE ~ade59
   where TABLESETY is EMPTY:
   where BNAMESETY1 is BNAMESETY.

H026 TABLES BNAMESETY simple type of TYPE and BNAMESETY1:
   TABLES BNAMESETY scalar type of TYPE and
   BNAMESETY1 ~ade27
   TABLES subrange type of TYPE ~ade29
   where BNAMESETY1 is BNAMESETY:
   TABLES old type identifier of TYPE ~ade31
   where BNAMESETY1 is BNAMESETY.

H027 TABLESETY the ONESth LOCSETY slink to ONESETY table
   BNAMESETY scalar type of scalar defined by BNAMES needs
   one boxes and BNAMESETY1:
   {
     check BNAMES against BNAMESETY with the last
     BNAMESETY1 ~ade29
   }.

H028 check left NAME right BNAMESETY against BNAMESETY1 with
   the last BNAMESETY2:
   limitation on NAME identifier ~ade31
   where NAME is not in BNAMESETY1:
   i,
   check BNAMESETY against BNAMESETY1 left NAME right with
   the last BNAMESETY2 ~ade28
   where BNAMESETY is EMPTY:
   limitation on NAME identifier ~ade31
   where NAME is not in BNAMESETY1:
   where BNAMESETY2 is BNAMESETY1 left NAME right.
H029 TABLES subrange type of TYPE:
   TABLES describe the signed or unsigned CONST of
   TYPE1 'ade30
   TYPE1 of scalar except real 'aty8
   
   TABLES describe the signed or unsigned CONST1 of
   TYPE1 'ade30
   based on CONST and CONST1 the type TYPE is a part of
   the type TYPE1 'aty9-11

H030 TABLES describe the signed CONST of TYPE:
   TABLES consider the constant CONST 'ade11
   TABLES the TYPE is defined by the form of constant
   CONST 'aty1-6

H031 TABLES old type identifier of NAME in ONES table:
   NAME identifier 'asy2
   TABLES find the amount of old identifier NAME to be
   TYPE at ONES table 'ade13

H032 TABLES structured type of array defined by ONES
   dimensional INDEXES of TYPE needs ONES1 boxes and give
   TABLESETY TABLESETY1:
   array 'as
   TABLES the ONES index types INDEXES produce ONES2 as
   multiple of bounds and give TABLESETY 'ade33
   1 'as
   TABLES obtain type of TYPE and BNAMESETY with
   TABLESETY1 'ade25
   TABLES from the type TYPE get type definition TYPEDEF
   needs ONES3 boxes 'aty13
   the integer uplus ONES1 equal integer uplus ONES2 times
   integer uplus ONES3 'aop27

H033 TABLES the one ONESETY index types INDEX INDEXESETY
   produce ONES ONESETY1 as multiple of bounds and give
   TABLESEY TABLESETY1:
   TABLES single index type INDEX with ONES items
   TABLESEY 'ade34,38
   1 'as
   TABLES the ONESETY index types INDEXESETY produce
   ONESETY1 as multiple of bounds and give
   TABLESETY1 'ade33
   where ONESETY INDEXESETY ONESETY1 TABLESETY1 is EMPTY 'as
   TABLES single index type INDEX with ONES items and
   TABLESETY 'ade34,38
H034 TABLESET the ONESth LOCSETY slink to ONESETY table single
index type left in ONES1 table right with ONES2 items and
the ONES3th loc be type of scalar defined by SCALAR needs
one boxes and slink to ONES4 table :
  TABLESET the ONESth LOCSETY slink to ONESETY table
·simple type of SCRG defined by SCALAR needs one
  boxes .ade26
  where SCALAR differs from REALRANGE ,
  where ONES3 is ONES one ,
  where ONES1 is ONES3 ;
  where ONES4 is ONES ;
  there are ONES2 items of SCALAR.ade35-37

H035 there are one ONESETY items of BNAME BNAMESETY :
there are ONESETY items of BNAMESETY ade35
where ONESETY BNAMESETY is EMPTY .

H036 there are one ONESETY items of KCHAR KCHARSETY :
there are ONESETY items of KCHARSETY ade36
where ONESETY KCHARSETY is EMPTY .

H037 there are ONES items of integer from INT to INT1 :
  the integer INT2 equal to integer INT1 minus integer
  INT .ade27
  the integer uplus CNES equal to integer uplus one plus
  integer INT2 .ade27

H038 TABLES single index type left IDEN right with ONES items
and TABLESETY :
  TABLES old type identifier IDEN .ade31
  where TABLESETY is EMPTY ;
  TABLES the type identifier IDEN introduce SCRG defined
  by SCALAR needs one boxes .ade12
  where SCALAR differs from REALRANGE ;
  there are ONES items of SCALAR .ade35-37

H039 TABLES structured type of record defined by rekord
  FIELDLIST end needs UINT boxes and give TABLESETY :
  record ,
  TABLES field list FIELDLIST present offset zero last
  offset UINT and BNAMESETY with TABLESETY ;ade40
end .

H040 TABLES BNAMESETY field list SECTIONS present offset UINT
last offset UINT1 and BNAMESETY1 with TABLESETY :
  TABLES BNAMESETY only fixed part SECTIONS present offset
  UINT last offset UINT1 and BNAMESETY1 with
  TABLESETY ade41
  TABLES BNAMESETY fixed and variant part FIELDLIST
  present offset UINT last offset UINT1 and BNAMESETY1 with
  TABLESETY ade46
  TABLES BNAMESETY only variant part FIELDLIST present
  offset UINT last offset UINT1 and BNAMESETY1 with
  TABLESETY1 ade47
H041 TABLES BNAMESETY only fixed part SECTIONETY SECTIONETY present offset UINT last offset UINT1 and BNAMESETY1 with TABLESETY TABLESETY1:
  TABLES BNAMESETY might empty section SECTIONETY present offset UINT last offset UINT2 and BNAMESETY2 with TABLESETY @de42

H042 TABLES BNAMESETY might empty section SECTIONETY present offset UINT last offset UINT1 and BNAMESETY1 with TABLESETY:
  TABLES BNAMESETY selection group SECTIONETY present offset UINT last offset UINT1 and BNAMESETY1 with TABLESETY @de43
    where SECTIONETY TABLESETY1 is EMPTY,
    where BNAMESETY1 is BNAMESETY:
    where UINT1 is UINT.

H043 TABLES BNAMESETY selection group SELECTIONS present offset UINT last offset UINT1 and BNAMESETY1 with TABLESETY:
  BNAMESETY the group of names BNAMESETY1 present BNAMESETY1 @de44
  TABLES obtain type of TYPE and BNAMESETY1 with TABLESETY @de25
  TABLES SELECTIONS is built from BNAMESETY1 of TYPE present offset UINT last offset UINT1 @de45

H044 BNAMESETY the group of names left NAME right BNAMESETY1 present BNAMESETY1:
  limitation on NAME identifier @sy1
  where NAME is not in BNAMESETY:

  BNAMESETY left NAME right the group of names BNAMESETY1 present BNAMESETY1 @de44
  where BNAMESETY1 is EMPTY:
  limitation on NAME identifier @sy1
  where NAME is not in BNAMESETY:
  where BNAMESETY is BNAMESETY left NAME right.
H045 TABLES selection NAME of TYPE with offset UINT ends
SELECTIONSETY is built from left NAME1 right BNAMESETY of
TYPE1 present offset UINT1 last offset UINT2 :
where NAME1 is NAME1,
where TYPE is TYPE1,
where UINT is UINT1,
TABLES from the type TYPE1 get type definition TYPEDEF
needs UINT3 boxes 3ty13
the integer uplus UINT4 equal to integer UINT1 plus
integer uplus UINT3,9op27
TABLES SELECTIONSETY is built from BNAMESETY of TYPE1
present offset UINT4 last offset UINT2 3de45
where SELECTIONSETY BNAVGETY is EMPTY,
where NAME is NAME1,
where TYPE is TYPE1,
where UINT is UINT1,
TABLES from the type TYPE1 get type definition TYPEDEF
needs UINT3 boxes 3ty13
the integer uplus UINT2 equal to integer uplus UINT1
plus integer uplus UINT3,9op27

H046 TABLES BNAMESETY fixed and variant part SECTIONS VARPART
present offset UINT last offset UINT1 and BNAMESETY1 with
TABLESETY TABLESETY1 :
TABLES BNAMESETY only fixed part SECTIONS present offset
UINT last offset UINT2 and BNAMESETY2 with
TABLESETY 9de41
TABLES BNAMESETY2 only variant part VARPART present
offset UINT2 last offset UINT1 and BNAMESETY1 with
TABLESETY1.9de47

H047 TABLES BNAMESETY only variant part case TAGFIELD of
VARIANTS present offset UINT last offset UINT1 and
BNAMESETY1 with TABLESETY :
TABLES BNAMESETY tag field TAGFIELD present offset UINT
last offset UINT2 and BNAMESETY2 with tag type
IDEN 9de48
TABLES BNAMESETY2 variant group VARIANTS present offset
UINT2 label last offset UINT1 label KONSTANTSETY and
BNAMESETY1 with tag type IDEN produce TABLESETY.9de51

H048 TABLES BNAMESETY tag field SELECTIONETY EXTRA present
offset UINT last offset UINT1 and BNAMESETY1 with tag type
IDEN :
TABLES BNAMESETY discriminated union SELECTIONETY EXTRA
present offset UINT last offset UINT1 and BNAMESETY1
with tag type IDEN 9de49
where SELECTIONETY is EMPTY,
where BNAMESETY1 is BNAMESETY,
TABLES free union EXTRA present offset UINT last offset
UINT1 with tag type IDEN.9de50
H049 TABLES BNAMESETY discriminated union selection NAME of IDEN with offset UINT endsel EXTRA present offset UINT1 last offset ONES and BNAMESETY1 with tag type IDEN1:

- where NAME is not in BNAMESETY,
- where BNAMESETY1 is BNAMESETY left NAME right,
- TABLES old type identifier of IDEN, and
- TABLES the type identifier IDEN introduce SCRG defined by SCALAR needs one boxes, and
- where SCRG defined by SCALAR needs one boxes differs from REALTYPE,
- where IDEN1 is IDEN,
- where UINT is UINT1,
- the integer ONES1 equal to integer uplus UINT1 plus integer uplus one,
- where EXTRA is extra IDEN with offset ONES1 endext,
- where ONES is ONES1 one.

H050 TABLES free union extra IDEN with offset UINT endext present offset UINT1 last offset ONES with tag type IDEN1:

- TABLES old type identifier of IDEN, and
- TABLES the type identifier IDEN introduce SCRG defined by SCALAR needs one boxes, and
- where SCRG defined by SCALAR needs one boxes differs from REALTYPE,
- where IDEN1 is IDEN,
- where UINT is UINT1,
- the integer ONES1 equal to integer uplus UINT1 plus integer uplus one.

H051 TABLES BNAMESETY variant group VARIANTETY VARIANTSETY present offset UINT label KONSTANTSETY last offset UINT1 label KONSTANTSETY1 and BNAMESETY1 with tag type IDEN:

- TABLES BNAMESETY might empty single variant VARIANTETY present offset UINT label KONSTANTSETY last offset UINT2 label KONSTANTSETY2 and BNAMESETY2 with tag type IDEN produce TABLESETY, and
- TABLES BNAMESETY2 variant group VARIANTETY present offset UINT label KONSTANTSETY2 last offset UINT1 label KONSTANTSETY1 and BNAMESETY1 with tag type IDEN:

- TABLES BNAMESETY left UINT2 right produce TABLESETY, and
- TABLES BNAMESETY2 variant group VARIANTETY present offset UINT label KONSTANTSETY2 last offset UINT2 label KONSTANTSETY1 and BNAMESETY1 with tag type IDEN produce TABLESETY, and
- UINT1 is the maximum of BUINTSETY left UINT2 right.
H052 TABLES BNAMESETY might empty single variant VARIANTETY present offset UINT label KONSTANTSETY last offset UINT1 label KONSTANTSETY1 and BNAMESETY1 with tag type IDEN produce TABLESETY:
TABLES BNAMESETY nonempty single variant VARIANTETY present offset UINT label KONSTANTSETY last offset UINT1 label TABLES KONSTANTSETY1 and BNAMESETY1 with tag type IDEN produce TABLESETY.  
where VARIANTETY TABLESETY is EMPTY, where BNAMESETY1 is BNAMESETY, where UINT1 is UINT, where KONSTANTSETY1 is KONSTANTSETY.

H053 TABLES BNAMESETY nonempty single variant variant label KONSTANTS endlab FIELDLIST endvar present offset UINT label KONSTANTSETY last offset UINT1 label KONSTANTSL1 and BNAMESETY1 with tag type IDEN produce TABLESETY:
TABLES the type identifier IDEN introduce SCRG defined by SCALAR needs one boxes at ONES table.  
TABLES the labels KONSTANTS which is in SCALAR and distinct from KONSTANTSETY at ONES table.  
where KONSTANTSL1 is KONSTANTSETY KONSTANTS.  
TABLES BNAMESETY field list FIELDLIST present offset UINT last offset UINT1 and BNAMESETY1 with TABLESETY.  

H054 TABLES the labels KONSTANT KONSTANTSETY which is in SCALAR and distinct from KONSTANTSETY1 at ONES table:
TABLES look for the signed or unsigned constant CONST of SCRG defined by SCALAR needs one boxes at ONES table.  
where KONSTANT is not in KONSTANTSETY1, TABLES the labels KONSTANT which is in SCALAR and distinct from KONSTANTSETY1 at ONES table.  
where KONSTANTSETY is EMPTY, TABLES look for the signed or unsigned constant CONST of SCRG defined by SCALAR needs one boxes at ONES table.  
where KONSTANT is not in KONSTANTSETY1.

H055 TABLESETY the ONESTh LOCSETY slink to ONESETY table TABLESETY1 look for the signed or unsigned constant CONST of TYPE at ONES table:
where ONES is ONES1, TABLESETY the ONESTh LOCSETY slink to ONESETY table describe the signed or unsigned CONST of TYPE.  

H056 UINT is the maximum of BUINT BUINTSETY:
UINT is the maximum of integer numbers BUINT BUINTSETY.  
where BUINTSETY is EMPTY, where BUINT is left UINT right.
H057  UINT is the maximum of integer numbers left UINT1 right left UINT2 right BUINTSETY ;
    the integer uplus UINT1 greater or equal to integer uplus UINT2 ;@ap83
    UINT is the maximum of left UINT1 right BUINTSETY ;@ap83
    the integer UINT1 less than integer uplus UINT2 ;@ap83
    UINT is the maximum of left UINT2 right BUINTSETY. @de56

H058  TABLES structured type of set defined by elements of TYPE needs one boxes and give TABLESETY :
      set ;
      of ;
      where TABLESETY is EMPTY ;
      TABLES simple type of TYPE and BNAMESETY ;@de26
      TABLES from the type TYPE get type definition SCRG
      defined by SCALAR needs one boxes ;@ty13
      where SCRG defined by SCALAR needs one boxes differs
      from REALTYPE.

H059  TABLES pointer type of pointer defined by IDEN needs one boxes :
      ↑ ;
      TABLES old type identifier of IDEN. @de31

H060  TABLESETY the ONESt th LOCSETY slink to ONESETY table
      TABLESETY1 each pointer type is defined properly from ONES table !
      where ONES is ONE1 ;
      TABLESETY the ONESth LOCSETY slink to ONESETY table look
      for pointer type in LOCSETY i@de61
      where LOCSETY is EMPTY.

H061  TABLES look for pointer type in LOCSETY LOC ;
      TABLES the location LOC might be of pointer type ;@de62
      TABLES look for pointer type in LOCSETY i@de61
      where LOCSETY is EMPTY ;
      TABLES the location LOC might be of pointer type. @de62

H062  TABLES the location CHARSETY might be of pointer type ;
      where CHARSETY is CHARSETY1 pointer defined by IDEN
      CHARSETY2 ;
      TABLES the type identifier IDEN introduce TYPE ;@ty12
      TABLES the location CHARSETY1 CHARSETY2 might be of
      pointer type i@de62
      where pointer defined is not in CHARSETY.
H063 TABLES the ONESth LOCS link to ONESETY table BNAMESETY consider groups of variable locations LOCS LOCSY1 and BNAMEs with TABLESETY TABLESETY1:

TABLESETY the ONESth LOCS link to ONESETY table BNAMESETY a group of same type variables give LOCS and BNAMEs1 with TABLESETY i3de64

TABLESETY the ONESth LOCS link to ONESETY table BNAMESETY consider groups of variable locations LOCSY1 and BNAMEs with TABLESETY1 i3de63

where LOCSY1 TABLESETY1 is EMPTY.

TABLESETY the ONESth LOCS link to ONESETY table BNAMESETY a group of same type variables give LOCS and BNAMEs with TABLESETY.3de64

H064 TABLES BNAMESETY a group of same type variables give LOCS and BNAMEs with TABLESETY:

BNAMESETY the group of names BNAMEs1 present BNAMEs1 ade44

TABLES BNAMEs1 obtain type of TYPE and BNAMEs2 with TABLESETY.3de25

LOCS variable locations made from BNAMEs1 of TYPE.3de65

H065 loc NAME be variable of TYPE end LOCS link variable locations made from left NAME1 right BNAMESETY of TYPE1:

where NAME is NAME1,

where TYPE is TYPE1,

LOCS variable locations made from BNAMESETY of TYPE1 i3de65.

where LOCS BNAMESETY is EMPTY,

where NAME is NAME1,

where TYPE is TYPE1.

H066 TABLES BNAMESETY procedures or functions from ONEs table give STMTSETY and TABLES1 BNAMESETY1:

TABLES BNAMESETY nonempty procedure or functions from ONEs table give STMTSETY and TABLES1 BNAMESETY1 i3de67

where STMTSETY is EMPTY,

where TABLES1 is TABLES,

where BNAMESETY1 is BNAMESETY.

H067 TABLES BNAMESETY nonempty procedure or functions from ONEs table give STMTSETY STMTSETY1 and TABLES1 BNAMESETY1:

TABLES BNAMESETY single procedure or function from ONEs table give STMTSETY and TABLES2 BNAMESET2 3de68

TABLES2 BNAMESET2 nonempty procedure or functions from ONEs table give STMTSETY1 and TABLES1 BNAMESETY1 i3de67

where STMTSETY1 is EMPTY.

TABLES BNAMESETY single procedure or function from ONEs table give STMTSETY and TABLES1 BNAMESETY1.3de68

H068 TABLES BNAMESETY single procedure or function from ONEs table give STMTSETY and TABLES1 BNAMESETY1:

FUNPROC sort is of four types 3day103

TABLES BNAMESETY single FUNPROC from ONEs table give STMTSETY and TABLES1 BNAMESETY1.3de69
H069 TABLES BNAMESETY single FUNPROC from ONES table give
STMTSETY and TABLES1 BNAMESETY1:
TABLES BNAMESETY new complete FUNPROC from ONES table
give STMTSETY and TABLES1 BNAMESETY1 i3de70
where BNAMESETY1 is BNAMESETY,
TABLES already declared FUNPROC from ONES table give
STMTSETY and TABLES1 i3de79
where STMTSETY is EMPTY,
TABLES BNAMESETY only head FUNPROC from ONES table give
TABLES1 BNAMESETY1 and parameter identifiers BNAMESETY2
body is ahead i3de71
i:
forward.

H070 TABLES BNAMESETY new complete FUNPROC from ONES table
give STMTSETY NAME in its ONES1 table STMTSETY1 exit and
TABLES1 BNAMESETY1 left NAME1 right1:
where ONES1 is ONES,
TABLES BNAMESETY only head FUNPROC from ONES table give
TABLES2 BNAMESETY1 left NAME1 right and parameters
identifiers BNAMESETY2 i3de71
where NAME is NAME1;
1:
TABLES2 BNAMESETY2 from block introduce TABLES1
BNAMESETY3 and STMTSETY with main compound statement
STMTSETY1 i3de5
when FUNPROC is a function NAME1 should be in left hand
side of one of STMTSETY STMTSETY1 at least.3de78

H071 TABLES BNAMESETY only head FUNPROC from ONES table give
TABLES1 BNAMES and parameter identifiers BNAMESETY1
USAGETY:
limitation on NAME identifier i3sy1
where NAME is not in BNAMESETY,
where BNAMES is BNAMESETY left NAME right,
TABLES find last table number ONES1 i3de4
TABLES the ONES1 oneth slink to ONES table PARNUMETY
formal parameters give TABLES2 BNAMESETY1 i3de72
TABLES the type of FUNPROC formal parameter or
subroutine is TYPE i3ty14,15
TABLES2 put the location loc NAME be FUNPROC of
PARNUMETY type TYPE and ONES1 one as its local table
USAGETY end in the ONES table to give TABLES1,3de77

H072 TABLES the ONEStth slink to ONESETY table PARNUMETY formal
parameters give TABLES1 BNAMESETY:
(,
TABLES nonempty PARNUMETY which is formal produce LOCS
and BNAMESETY i3de73
),
where TABLES1 is TABLES the ONEStth LOCS slink to ONESETY
table;
where PARNUMETY BNAMESETY is EMPTY,
where TABLES1 is TABLES the ONEStth slink to ONESETY
table.
H073 TABLES BNAMESETY nonempty ONES ONESETY parameters which is formal produce LOCSETY and BNAMEs:

TABLES BNAMESETY same type ONES formal parameter group LOCS give BNAMEs1 ade74

TABLES BNAMEs1 nonempty ONESETY parameters which is formal produce LOCSETY and BNAMEs ade73

where ONESETY LOCSETY is EMPTY;

TABLES BNAMESETY same type ONES formal parameter group LOCS give BNAMEs ade74

H074 TABLES BNAMESETY same type ONES formal parameter group LOCS give BNAMEs:

SHAPE sort is of four types 10sy103

BNAMESETY the group of names BNAMEs1 present BNAMEs ade46

there are ONES number of BNAMEs1 formal parameters ade75

TABLES1 the type of SHAPE formal parameter or subroutine is TYPE 10ty14,15

LOCs formal SHAPE parameter locations made from BNAMEs1 of TYPE ade76

H075 there are one ONESETY number of BNAME BNAMESETY formal parameters:

there are ONESETY number of BNAMESETY formal parameters ade75

where ONESETY BNAMESETY is EMPTY.

H076 LOC LOCSETY formal SHAPE parameter locations made from left NAME right BNAMESETY of TYPE:

where LOC is loc NAME be formal parameter of SHAPE of TYPE end;

LOCSETY formal SHAPE parameter locations made from BNAMESETY of TYPE ade76

where LOCSETY BNAMESETY is EMPTY;

where LOC is loc NAME be formal parameter of SHAPE of TYPE end.

H077 TABLESETY the ONEStth LOCSETY link to ONESETY table TABLESETY1 put the location LOC in the ONESt1 table to give TABLES:

where ONES is ONESt1;

where TABLES is TABLESETY the ONEStth LOCSETY LOC link to ONESETY table TABLESETY1.

H078 when FUNPROC is a function NAME should be in left hand side of one of STMTSETY at least:

where FUNPROC is function;

where STMTSETY contains assign left NAME right becomes;

where FUNPROC is procedure.
H079 TABLESETY the ONESth LOCs slink to ONESETY table
TABLESETY1 already declared FUNPROC from ONES1 table give
STMTSETY NAME in its ONES2 table STMTSETY1 exit and
TABLES1:
  NAME identifier ;@sy2
  where ONES is ONES1,
  where ONES2 is ONES1,
where LOCs is LOCSETY loc NAME be FUNPROC of PARNUMETY
type TYPE and ONES3 as its local table body is ahead and
LOCSETY1,
TABLESETY the ONESth LOCs slink to ONESETY table
TABLESETY1 find all parameters identifier from ONES3
  table to be BNAMESETY ;@de80
TABLESETY the ONESth LOCSETY loc NAME be FUNPROC of
PARNUMETY type TYPE and ONES3 as its local table end
LOCSETY1 slink to ONESETY table TABLESETY1 BNAMESETY
from block introduce TABLES1 BNAMESETY1 and STMTSETY
with main compound statement STMTSETY1 ;@de5
when FUNPROC is function NAME should be in left hand
side of one of STMTSETY STMTSETY1 at least ;@de78

H080 TABLESETY the ONESth LOCSETY slink to ONESETY table
TABLESETY1 find all parameters identifier from ONES1 table
to be BNAMESETY :  
where ONES is ONES1,
these locations LOCSETY introduce identifiers
BNAMESETY ;@de81
  where ONES is ONES1,
  where LOCSETY is EMPTY,

H081 these locations LOC LOCSETY introduce identifiers left
NAME right BNAMESETY : 
where LOC contains loc NAME ,
these locations LOCSETY introduce identifiers
BNAMESETY ;@de81
  where LOCSETY BNAMESETY is EMPTY ,
  where LOC contains loc NAME be.

H082 TABLESETY the ONESth LOCSETY slink to ONESETY table
TABLESETY1 from ONES1 table subroutine head and block is
declared : 
where ONES is ONES1,
  where body is ahead is not in LOCSETY.
Compile time semantics for statements

H001 TABLES stmt train STMTETY STMTSETY from ONES table :
    TABLES might empty stmt STMTETY from ONES table @st2
    
    TABLES stmt train STMTETY from ONES table @st1
    where STMTETY is EMPTY.
    TABLES might empty stmt STMTETY from ONES table @st2

H002 TABLES might empty STMTETY from ONES table :
    TABLES single stmt STMTETY from ONES table @st3
    where STMTETY is EMPTY.

H003 TABLES single stmt assign VARIABLE becomes EXP stmt from ONES table :
    TABLES variable expression VARIABLE of TYPE from ONES
    table in left side @st30
    = ,
    TABLES single expression EXP of TYPE1 from ONES
    table @st22
    TABLES the type TYPE either is the wider form of TYPE1
    or the same @st17

H004 TABLES single stmt procedure SUBCALL call from ONES table :
    TABLES subroutine invocation procedure SUBCALL call of
    void from ONES table @st47

H005 TABLES single stmt STMTETY from ONES table :
    begin ,
    TABLES stmt train STMTETY from ONES table @st1
    end.

H006 TABLES single stmt if EXP then STMTETY ELSESTY fi from ONES table :
    if ,
    TABLES compound expression EXP of BOOLTYPE from ONES
    table @st29
    then ,
    TABLES might empty stmt STMTETY from ONES table @st2
    TABLES else part ELSESTY from ONES table @st7
    where ELSESTY is EMPTY ,
    if ,
    TABLES compound expression EXP of BOOLTYPE from ONES
    table @st29
    then ,
    TABLES might empty stmt STMTETY from ONES table @st2

H007 TABLES else part else STMTETY from ONES table :
    else ,
    TABLES might empty stmt STMTETY from ONES table @st2
H008 TABLES single stmt case EXP of CASEELEMSETY esac from ONES table :
  case
    TABLES single expression EXP of SCRG defined by SCALAR needs one boxes from ONES table ,@st22
         where SCALAR differs from REALRANGE ,
   of ,
    TABLES case stmt part CASEELEMSETY from ONES table and label type SCALAR present label last label
         KONSTANTSETY ,@st9
  end.

H009 TABLES case stmt part CASEELEMSETY CASEELEMSETY from ONES table and label type SCALAR present label KONSTANTSETY
        last label KONSTANTSETY1 :
    TABLES might empty case item CASEELEMETY from ONES table and label type SCALAR present label KONSTANTSETY last
         label KONSTANTSETY2 ,@st10
    TABLES case stmt part CASEELEMSETY from ONES table and label type SCALAR present label KONSTANTSETY2 last label
         KONSTANTSETY1 ,@st11
      where CASEELEMSETY is EMPTY ,
    TABLES might empty case item CASEELEMETY from ONES table and label type SCALAR present label KONSTANTSETY last
         label KONSTANTSETY1 ,@st10

H010 TABLES might empty case item CASEELEMETY from ONES table and label type SCALAR present label KONSTANTSETY last
        label KONSTANTSETY1 :
    TABLES nonempty case item CASEELEMETY from ONES table and label type SCALAR present label KONSTANTSETY last
         label KONSTANTSETY1 ,@st11
      where CASEELEMETY is EMPTY ,
    where KONSTANTSETY1 is KONSTANTSETY.

H011 TABLES nonempty case item label KONSTANTS end lab STMTETY
        from ONES table and label SCALAR present label KONSTANTSETY last label KONSTANTSETY1 :
    TABLES the labels KONSTANTS which is in SCALAR and distinct from KONSTANTSETY ,@st54
      where KONSTANTSETY1 is KONSTANTSETY KONSTANTS ,
    TABLES single stmt STMTETY from ONES table ,@st3-6,8,12-14

H012 TABLES single stmt while EXP do STMTETY od from ONES table :
    while
    TABLES compound expression EXP of BOOLTYPE from ONES table ,@st29
    do
      TABLES single stmt STMTETY from ONES table ,@st3-6,8,12-14
H013 TABLES single stmt repeat STMTETY until EXP taper from ONES table:
   repeat;
   TABLES stmt train STMTETY from ONES table \@st2
   until;
   TABLES compound expression EXP of BOOLTYPE from ONES table.\@st29

H014 TABLES single stmt for NAME becomes EXP through DOTO EXP1
do STMTETY od from ONES table:
   for;
   TABLES entire variable left NAME right of SCRG defined
   by SCALAR needs one boxes from ONES table \@st31
   where SCALAR differs from REALRANGE;
   =
   TABLES single expression EXP of SCRG defined by SCALAR
   needs one boxes from ONES table \@st22
   DOTO up or down control \@st15
   TABLES single expression EXP1 of SCRG defined by SCALAR
   needs one boxes from ONES table \@st22
   do;
   TABLES single stmt STMTETY from ONES
   table \@st3-6,8,12-14
   where assign left NAME right becomes is not in
   STMTETY \@st16
   TABLES NAME is not an actual variable parameter in a
   call in EXP EXP1 STMTETY from ONES table.\@st16

H015 DOTO up or down control:
   to;
   where DOTO is to ,
   downto ;
   where DOTO is downto.

H016 TABLES NAME is not an actual variable parameter in a call
   in CHARSETY from ONES table:
   where CHARSETY contains FUNPROC;
   CHARSETY delete a subroutine FUNPROC SUBCALL invocation
   and leaves CHARSETY1 \@st17
   TABLES the single call SUBCALL might have NAME as actual
   variable parameter from ONES table \@st18
   TABLES NAME is not an actual variable parameter in a
   call in CHARSETY1 from ONES table \@st16
   where FUNPROC is not in CHARSETY.

H017 CHARSETY FUNPROC SUBCALL call CHARSETY1 delete a
   subroutine FUNPROC1 SUBCALL1 invocation and leaves
   CHARSETY2:
   where FUNPROC is FUNPROC1;
   where CHARSETY2 is CHARSETY CHARSETY1.
H018 TABLESETY the ONESth LOCSETY slink to ONESETY table
TABLESETY1 the single call NAME with actuals COMMASETY might have NAME1 as actual variable parameter from ONES1 table:
    where ONES1 is ONES,
    TABLESETY the ONESth LOCSETY slink to ONESETY table find
the amount of old identifier NAME to be FUNPROC of
PARNUMETY type TYPE and ONES2 as its local table iade13
TABLESETY the ONESt LOCSETY slink to ONESETY table
TABLESETY1 left most formal parameter location of
COMMASETY should not be variable parameter NAME1 from
ONES2 table.@st19

H019 TABLESETY the ONESt LOCSETY slink to ONESETY table
TABLESETY1 left most formal parameter location COMMASETY
should not be variable parameter NAME from ONES1 table:
    where ONES is ONES1,
    LOCSETY the first location is not a variable parameter
    corresponded to nonempty COMMASETY of NAME i0st20
where COMMASETY PARNUMETY is EMPTY.

H020 LOC LOCSETY the first location is not a variable parameter
corresponded to nonempty COMMA COMMASETY of NAME:
LOC the single location is not a variable parameter to
the single COMMA of NAME iast21
LOCSETY the left most location is not a variable
parameter related to COMMASETY of NAME i0st20
where COMMASETY is EMPTY,
LOC the single location is not a variable parameter to
the single COMMA of NAME.@st21

H021 loc NAME be formal parameter of SHAPE of TYPE end the
single location is not a variable parameter to the single
comma left NAME1 right ammoc of NAME2:
    where SHAPE is variable,
    where NAME1 differs from NAME2 i
where SHAPE differs from variable.

H022 TABLES single expression EXP of TYPE from ONES table:
    TABLES simple expression EXP of TYPE from ONES
    table iast23
    TABLES compound expression EXP of TYPE from ONES
table.@st29

H023 TABLES simple expression SEXP of TYPE from ONES table:
    TABLES term expression SEXP of TYPE from ONES
    table iast24
    TABLES signed expression SEXP of TYPE from ONES
    table iast27
    TABLES adding expression SEXP of TYPE from ONES
table.@st28

H024 TABLES term expression TERM of TYPE from ONES table:
    TABLES factor expression TERM of TYPE from ONES
    table iast25
    TABLES multiply expression TERM of TYPE from ONES
table.@st26
H025 TABLES factor expression FACTOR of TYPE from ONES table:
  TABLES variable expression FACTOR of TYPE from ONES table f;ast30
  TABLES unsigned constant expression FACTOR of TYPE from ONES table f;ast40
  TABLES bracketed factor FACTOR of TYPE from ONES table f;ast41
  TABLES negative factor FACTOR of TYPE from ONES table f;ast42
  TABLES set expression FACTOR of TYPE from ONES table f;ast43
  TABLES subroutine invocation FACTOR of TYPE from ONES table @st47

H026 TABLES multiply expression TERM MULOP FACTOR of TYPE from ONES table:
  TABLES term expression TERM of TYPE1 from ONES table r;ast24
  MULOP binary multiplication operator ,@sy104-8
  TABLES factor expression FACTOR of TYPE2 from ONES table r;ast25
  TABLES operand TYPE1 multiply operation MULOP operand TYPE2 make TYPE.@ty46

H027 TABLES signed expression UOP TERM of TYPE from ONES table:
  UOP operation ,@sy97-8
  TABLES term expression TERM of INTTYPE from ONES table r;ast24
  UOP operation ,@sy97-8
  TABLES term expression TERM of REALTYPE from ONES table @;ast24

H028 TABLES adding expression SEXP ADDOP TERM of TYPE from ONES table:
  TABLES simple expression SEXP of TYPE1 from ONES table r;ast23
  ADDOP binary adding operator ,@sy109-111
  TABLES term expression TERM of TYPE2 from ONES table r;ast24
  TABLES operand TYPE1 adding operation ADDOP operand TYPE2 make TYPE.@ty56

H029 TABLES compound expression SEXP1 RELOP SEXP2 of SCBOOLTYPE from ONES table:
  TABLES simple expression SEXP1 of TYPE1 from ONES table @;st23
  RELOP binary relational operation ,@sy112-118
  TABLES simple expression SEXP2 of TYPE2 from ONES table @;st23
  TABLES operand TYPE1 relational operation RELOP operand TYPE2 are compatible.@ty59
H030 TABLES variable expression VARIABLE of TYPE from ONES table LEFTETY :
   TABLES entire variable VARIABLE of TYPE from ONES table LEFTETY 18st31
   TABLES compound variable VARIABLE of TYPE from ONES table i@st34
   TABLES referenced variable VARIABLE of TYPE from ONES table @st39

H031 TABLES entire variable left NAME right of TYPE from ONES table LEFTETY :
   NAME identifier i@sy2
   TABLES the NAME of TYPE1 is either variable function or parameter LEFTETY from ONES table @@st32
   TABLES from the type TYPE1 get type definition TYPE.@ty13

H032 TABLES the NAME of TYPE is either variable function or parameter LEFTETY from ONES table :
   TABLES look for the old identifier NAME to be variable of TYPE from ONES table i@st33
   where LEFTETY is in left side,
   TABLES look for the old identifier NAME to be function of PARNUMETY type TYPE and ONES1 as its local table from ONES table i@st33
   TABLES look for the old identifier NAME to be formal parameter of value of TYPE from ONES table i@st33
   TABLES look for the old identifier NAME to be formal parameter of variable of TYPE from ONES table.@st33

H033 TABLESETY the ONESth LOCSETY slink to ONESETY table TABLESETY1 look for the old identifier NAME to be AMOUNT from ONES1 table :
   where ONES is ONES1 ,
   TABLESETY the ONESth LOCSETY slink to ONESETY table find the amount of old identifier NAME to be AMOUNT.@de13

H034 TABLES compound variable VARIABLE of TYPE from ONES table :
   TABLES indexed variable VARIABLE of TYPE from ONES table i@st35
   TABLES field designator VARIABLE of TYPE from ONES table.@st38

H035 TABLES indexed variable matrix VARIABLE indices COMMAS array of TYPE from ONES table :
   TABLES variable expression VARIABLE of array defined by ONES1 dimensional INDEXES of TYPE1 needs ONES2 boxes from ONES table i@st30
   TABLES from the type TYPE1 get type definition TYPE ,@ty13
   [ TABLES the ONES1 expressions COMMAS of types INDEXES from ONES table ,@st36 ].
H036 TABLES the one ONESETY expressions comma EXP ammoc
COMMASETY of types left IDEN right INDEXESETY from ONES
table:
TABLES the type identifier IDEN introduce TYPE .@ty12
TABLES single expression EXP of TYPE1 from ONES
table .@st22
TABLES the TYPE1 give TYPE2 from ONES table .@st37
the identity of scalar or subranges TYPE and
TYPE2 .@ty22

TABLES the ONESETY expressions COMMASETY of type
INDEXESETY from ONES table .@st36
where ONESETY COMMASETY INDEXESETY is EMPTY ,
TABLES the type identifier IDEN introduce TYPE .@ty12
TABLES single expression EXP of TYPE1 from ONES
table .@st22
TABLES the TYPE1 give TYPE2 from ONES table .@st22
the identity of scalar or subranges TYPE and TYPE2 .@ty22

H037 TABLESETY the ONESth LOCSETY slink to ONESEITY table
TABLESETY1 the TYPE give TYPE1 from ONES1 table :
where ONES is ONESt ,
TABLESETY the ONESth LOCSETY slink to ONESEITY table
from the type TYPE get type definition TYPE1 .@ty13

H038 TABLES field designator field VARIABLE point NAME select
of TYPE from ONES table :
TABLES variable expression VARIABLE of record defined by
rekkor FIELDLIST end needs UINT boxes from ONES
table .@st30

NAME identifier .@sy2
where FIELDLIST contains selection NAME of TYPE1 with
OFFSET TAGETY endsel ,
TABLES from the type TYPE1 get type definition
TYPE .@ty13

H039 TABLES referenced variable reference VARIABLE arrow of
TYPE from ONES table :
TABLES variable expression VARIABLE of pointer defined
by IDEN needs one boxes .@st30
TABLES type identifier IDEN introduce TYPE .@ty12

H040 TABLESETY the ONESth LOCSETY slink to ONESEITY table
TABLESETY1 unsigned constant expression CONST of TYPE from
ONES1 table :
where ONES1 is ONES ,
TABLESETY the ONESth LOCSETY slink to ONESEITY table
consider the unsigned constant of CONST .@de21
TABLESETY the ONESth LOCSETY slink to ONESEITY table
the TYPE is defined by the form of constant CONST .@ty1-6
H041 TABLES bracketed factor open bracket EXP close bracket of TYPE from ONES table:
\[
\begin{align*}
&\text{TABLES single expression EXP of TYPE from ONES table.} \@st22 \\
&\end{align*}
\]

H042 TABLES negative factor not FACTOR of BOOLTYPE from ONES table:
\[
\begin{align*}
&\text{not, TABLES factor expression FACTOR of BOOLTYPE from ONES table.} \@st25 \\
&\end{align*}
\]

H043 TABLES set expression subset ELEMENTSETY t e s b u s of set defined by elements of TYPE needs one boxes from ONES table:
\[
\begin{align*}
&\text{TABLES nonempty subset ELEMENTSETY of TYPE from ONES table.} \@st44 \\
&\text{where ELEMENTSETY is EMPTY,} \\
&\end{align*}
\]

H044 TABLES nonempty subset elem EXP mele ELEMENTSETY of TYPE from ONES table:
\[
\begin{align*}
&\text{TABLES single set member ELEMENT of TYPE from ONES table.} \@st45 \@st46 \\
&\text{TABLES nonempty subset ELEMENTSETY of TYPE from ONES table.} \@st44 \\
&\text{where ELEMENTSETY is EMPTY,} \\
&\text{TABLES single set member ELEMENT of TYPE from ONES table.} \@st45 \@st46 \\
&\end{align*}
\]

H045 TABLES single set member elem EXP mele SCRG defined by SCALAR needs one boxes from ONES table:
\[
\begin{align*}
&\text{TABLES single expression EXP of SCRG defined by SCALAR needs one boxes from ONES table.} \@st22 \\
&\text{where SCALAR differs from REALRANGE.} \\
&\end{align*}
\]

H046 TABLES single set member elem EXP dot dot EXPI mele of SCRG defined by SCALAR needs one boxes from ONES table:
\[
\begin{align*}
&\text{TABLES single expression EXPI of SCRG defined by SCALAR needs one boxes from ONES table.} \@st22 \\
&\text{where SCALAR differs from REALRANGE,} \\
&\end{align*}
\]
H047 TABLES subroutine invocation FUNPROC SUBCALL call of TYPE from ONES table:
   TABLES FUNPROC call SUBCALL of TYPE is ordinary subroutine call from ONES table $\alpha_{st48}$
   TABLES FUNPROC call SUBCALL of TYPE is a formal parameter from ONES table $\alpha_{st64}$
   TABLES FUNPROC call SUBCALL of TYPE is standard subroutine call from ONES table $\alpha_{st66,68,69,71,72,75,76,78,79}$

H048 TABLESETY the ONESt LOCSETY slink to ONESETY table
   TABLESETY1 FUNPROC call NAME with actuals COMMASETY of TYPE is ordinary subroutine call from ONES1 table:
      NAME identifier $\alpha_{sy2}$
      where ONES is ONES1,
   TABLESETY the ONESt LOCSETY slink to ONESETY table find the amount of old identifier NAME to be FUNPROC of ONES2 parameters type TYPE1 and ONES3 as its local table $\alpha_{de13}$
   TABLESETY the ONESt LOCSETY slink to ONESETY table
   TABLESETY1 base on FUNPROC from TYPE1 find the ultimate type TYPE $\alpha_{st49}$
   {
   TABLESETY the ONESt LOCSETY slink to ONESETY table
   TABLESETY1 there are ONES2 parameters formal location in ONESES table to be LOCS $\alpha_{st50}$
   TABLESETY the ONESt LOCSETY slink to ONESETY table
   TABLESETY1 formal parameter locations LOCS related to actual parameters COMMASETY from ONES table $\alpha_{st52}$
   }
   where COMMASETY is EMPTY,
      NAME identifier $\alpha_{sy2}$
      where ONES is ONES1,
   TABLESETY the ONESt LOCSETY slink to ONESETY table find the amount of old identifier NAME to be FUNPROC of type TYPE1 and ONES2 as its local table $\alpha_{de13}$
   TABLESETY the ONESt LOCSETY slink to ONESETY table
   TABLESETY1 base on FUNPROC from TYPE1 find the ultimate type TYPE $\alpha_{st49}$

H049 TABLES base on FUNPROC from TYPE find the ultimate type TYPE1:
   where FUNPROC is function,
   TABLES the type identifier TYPE introduce TYPE1 $\alpha_{ty12}$
   where FUNPROC is procedure,
   where TYPE is void,
   where TYPE1 is void.

H050 TABLESETY the ONESt LOCS LOCSETY slink to ONESETY table
   TABLESETY1 there are ONES1 parameters formal location in ONES2 table to be LOCS1:
   where ONES is ONES2,
   there exist ONES1 number of LOCS $\alpha_{st51}$
   where LOCS1 is LOCS.

H051 there exist one ONESETY number of LOC LOCSETY:
   there exist ONESETY number of LOCSETY $\alpha_{st51}$
   where ONESETY LOCSETY is EMPTY.
H052 TABLES formal parameter locations LOC LOCSETY related to actual parameters COMMA COMMASETY from ONES table:

1. TABLES formal parameter location LOC corresponded to actual parameter COMMA from ONES table 18st53-55

2. TABLES formal parameter locations LOCSETY related to actual parameters COMMASETY from ONES table 18st52 where LOCSETY COMMASETY is EMPTY.

TABLES formal parameter location LOC corresponded to actual parameter COMMA from ONES table 18st53-55

H053 TABLES formal parameter location loc NAME be formal parameter of value of IDEN end corresponded to actual parameter comma EXP ammoc from ONES table:

TABLES single expression EXP of TYPE1 from ONES table 18st22
TABLES the type identifier IDEN introduce TYPE2 18ty12
TABLES the type TYPE2 either is the wider form of TYPE1 or the same.18ty17

H054 TABLES formal parameter location loc NAME be formal parameter of variable of IDEN end corresponded to actual parameter comma VARIABLE ammoc from ONES table:

TABLES variable expression VARIABLE of TYPE1 from ONES table 18st30
TABLES the type identifier IDEN introduce TYPE2 18ty12
TABLES the TYPE2 and TYPE1 are identical.18ty20

H055 TABLES formal parameter location loc NAME be formal parameter of FUNPROC of TYPE end corresponded to actual parameter COMMA from ONES table:

TABLES the formal FUNPROC of NAME of TYPE is related to ordinary subroutine COMMA from ONES table 18st56
TABLES the formal FUNPROC of NAME of TYPE is related to formal subroutine COMMA from ONES table 18st62
TABLES the standard FUNPROC subroutine COMMA of TYPE.18st63

H056 TABLES the formal FUNPROC of NAME of TYPE is related to ordinary subroutine comma NAME1 ammoc from ONES table:

NAME1 identifier 18sy2
TABLES inspect NAME1 from ONES table to be FUNPROC of PARNUMETY type TYPE1 and ONES1 as its local table 18st57
TABLES with condition.FUNPROC the formal TYPE and TYPE1 are compatible 18st58
TABLES there could be PARNUMETY formal parameter locations in ONES1 table to be LOCSETY 18st59
if any formal parameters LOCSETY then they are of value type.18st60

H057 TABLESETY the ONEStth LOCSETY slink to ONESETY table
TABLESETY1 inspect NAME from ONES1 table to be AMOUNT
where ONES is ONESt.
TABLESETY the ONEStth LOCSETY slink to ONESETY table look for the old identifier NAME to be AMOUNT 18st33
H058 TABLES with condition FUNPROC the formal TYPE and actual TYPE1 are compatible:
   TABLES base on FUNPROC from TYPE find the ultimate type TYPE2, i\^{\text{ast}}49
   TABLES base on FUNPROC from TYPE1 find the ultimate type TYPE3, i\^{\text{ast}}49
   TABLES the subroutine types TYPE2 and TYPE3 are either void or usual, i\^{\text{ty66}}

H059 TABLES there could be PARNUMETY formal parameter locations in ONES table to be LOCSETY:
   TABLES there are PARNUMETY formal location in ONES table to be LOCSETY, i\^{\text{ast50}}
   where PARNUMETY LOCSETY is EMPTY.

H060 if any formal parameter LOCSETY then they are of value type:
   all formal parameters of LOCSETY are of value type i\^{\text{ast61}}
   where LOCSETY is EMPTY.

H061 all formal parameters of local NAME be formal parameter of value of IDEN end LOCSETY are of value type:
   all formal parameters of LOCSETY are of value type i\^{\text{ast61}}
   where LOCSETY is EMPTY.

H062 TABLES the formal FUNPROC of NAME of TYPE is related to formal subroutine comma NAME1 ammoc from ONES table:
   NAME1 identifier, i\^{\text{sy2}}
   TABLES inspect NAME1 from ONES table to be formal parameter of FUNPROC of TYPE1, i\^{\text{ast57}}
   TABLES with condition FUNPROC the formal TYPE and actual TYPE1 are compatible, i\^{\text{ty58}}

H063 TABLES the standard FUNPROC subroutine comma #NAME ammoc of TYPE:
   NAME identifier, i\^{\text{sy2}}
   where FUNPROC is procedure,
   where STNDPROC contains left NAME right,
   where TYPE is void, i\^{\text{ty67}}
   NAME identifier,
   where FUNPROC is function,
   where STANDFUNC contains left NAME right,
   TABLES the type identifier TYPE introduce TYPE1, i\^{\text{ty12}}
   the standard function NAME is of type TYPE1, i\^{\text{ty67}}
TABLESETV the ONESth LOCSETV slink to ONESETV table
TABLESETV1 FUNPROC call NAME with actuals COMMASETV of TYPE is a formal parameter from ONES1 table :
   NAME identifier,asy2
   where ONES is ONES1,
   TABLESETV the ONESth LOCSETV slink to ONESETV table look for the old identifier NAME to be formal parameter of FUNPROC of TYPE1,ast33
TABLESETV the ONESth LOCSETV slink to ONESETV table
TABLESETV1 base on FUNPROC from TYPE1 find the ultimate type TYPE,ast49[

TABLESETV the ONESth LOCSETV slink to ONESETV table
TABLESETV1 value actual parameters COMMASETV of ONES1 table ,ast65
]
where COMMASETV is EMPTY,
   NAME identifier,asy2
   where ONES is ONES1,
   TABLESETV the ONESth LOCSETV slink to ONESETV table look for the old identifier NAME to be formal parameter of FUNPROC of TYPE1,ast33
TABLESETV the ONESth LOCSETV slink to ONESETV table
TABLESETV1 base on FUNPROC from TYPE1 find the ultimate type TYPE,ast49

TABLES value actual parameters comma EXP ammoc COMMASETV from ONES table :
   TABLES single expression EXP of TYPE from ONES table ,ast22
   TABLES value actual parameters COMMASETV from ONES table ,ast65
   where COMMASETV is EMPTY,
   TABLES single expression EXP of TYPE from ONES table ,ast22

TABLES procedure call #READ with actuals COMMAS of void is standard subroutine call form ONES table :
   READ identifier,asy2
   TABLES all variable parameters COMMAS from ONES table ,ast67
]
H067 TABLES all variable parameters comma VARIABLE ammoc
COMMASETY from ONES table:
TABLES variable expression VARIABLE of TYPE from ONES
  table \ast30
there is an special TYPE for read subroutine
  parameter \asty68

\{ TABLES all variable parameters COMMASETY from ONES
  table \astt67
  where COMMASETY is EMPTY
  TABLES variable expression VARIABLE of TYPE from ONES
  table \ast30
  there is an special TYPE for read subroutine
  parameter \asty68
\}

H068 TABLES procedure call #READLN with actuals COMMASETY of
  void is standard subroutine call from ONES table:
  READLN identifier \asty2
  \{
  TABLES all variable parameters COMMASETY from ONES
  table \astt67
  \}
  where COMMASETY is EMPTY
  READLN identifier \asty2

H069 TABLES procedure call #WRITE with actuals COMMAS of void
  standard subroutine call from ONES table:
  WRITE identifier \asty2
  \{
  TABLES all expression parameters COMMAS from ONES
  table \astt70
  \}

H070 TABLES all expression parameters comma EXP ammoc COMMASETY
  from ONES table:
  TABLES single expression EXP of TYPE from ONES
  table \astt22
  TABLES there is an special TYPE for print
  subroutine \asty69
  \{
  TABLES all expression parameters COMMASETY from ONES
  table \astt70
  where COMMASETY is EMPTY
  TABLES single expression EXP of TYPE from ONES
  table \astt22
  TABLES there is an special TYPE for print
  subroutine \asty69
  \}

H071 TABLES procedure call #WRITELN with actuals COMMASETY of
  void is standard call from ONES table:
  WRITELN identifier \asty2
  \{
  TABLES all expression parameters COMMASETY from ONES
  table \astt70
  \}
  where COMMASETY is EMPTY
  WRITELN identifier \asty2
H072 TABLES procedure call #NAME with actuals commas of void is
standard subroutine call from ONES table:
NAME identifier ;@sy2
the NAME either is new or dispose ,@st73
,  
TABLES all pointer variables commas from ONES
table ,@st74
).

H073 the NAME either is new or dispose :  
where NAME is 'NEW ;  
where NAME is 'DISPOSE.

H074 TABLES all pointer variables comma VARIABLE ammoc
COMMASETY from ONES table:  
TABLES variable expression VARIABLE of pointer defined
by IDEN needs one boxes from ONES table ,@st30  
TABLES all pointer variable COMMASETY from ONES
table ,@st74
where COMMASETY is EMPTY ;  
TABLES variable expression VARIABLE of pointer defined
by IDEN needs one boxes from ONES table ,@st30

H075 TABLES function call #ORD with actuals comma EXP ammoc of
SCINTTYPE is standard subroutine call from ONES table:
ORD identifier ;@sy2
,  
TABLES single expression EXP of SCRG defined by SCALAR
needs one boxes from ONES table ,@st22
where SCALAR differs from REALRANGE ;  
).

H076 TABLES function call #NAME with actuals comma EXP ammoc of
SCRG defined by SCALAR needs one boxes is standard
subroutine call from ONES table:
NAME identifier ;@sy2
NAME either is successor or predecessor ,@st77
,  
TABLES single expression EXP of SCRG defined by SCALAR
needs one boxes from ONES table ,@st22
where SCALAR differs from REALRANGE ;  
).

H077 NAME either is successor or predecessor :  
where NAME is 'SUCC ;  
where NAME is 'PRED.

H078 TABLES function call #CHR with actuals comma EXP ammoc of
SCCHARTYPE :  
CHR identifier ;@sy2
,  
TABLES single expression EXP of INTTYPE from ONES
table ,@st22
).

H079 TABLES function call #NAME with actuals of SCBOOLTYPE
from ONES table:
NAME identifier ;@sy2
NAME could be either eoln or eof, @st80

H080 NAME could be either eoln or eof :  
where NAME is 'EOLN ;  
where NAME is 'EOF.
Execution semantics for statements

H001 TABLES with STMTS and CONDITION to execute NAME in its ONES table to give CONDITION1:
where STMTS is STMTSETY NAME in its ONES table STMTSETY1 
exit STMTSETY2,
where exit is not in STMTSETY,
TABLES with STMTS and CONDITION execute might empty STMTSETY1 to give CONDITION1.@ex2

H002 TABLES with STMTS and CONDITION execute might empty STMTSETY to give CONDITION1:
TABLES with STMTS and CONDITION execute nonempty STMTSETY to give CONDITION 1@ex3
where STMTSETY is EMPTY,
where CONDITION1 is CONDITION.

H003 TABLES with STMTS and CONDITION execute nonempty stmt train STMTA STMTSETY to give CONDITION:
TABLES with STMTS and CONDITION execute single STMTA to give CONDITION2 .@ex4-6,8,11-14
TABLES with STMTS and CONDITION2 execute stmt train STMTSETY to give CONDITION1 1@ex3
where STMTSETY is EMPTY,
TABLES with STMTS and CONDITION execute single STMTA to give CONDITION1.@ex4-6,8,11-14

H004 TABLES with STMTS and CONDITION execute single assign VARIABLE becomes EXP stmt to give INFILE OUTFILE with STACK and HEAPETY:
TABLES with STMTS and CONDITION execute might function left variable VARIABLE to give CONDITION1 and produce ADDRESS of TYPE .@ex23
TABLES with STMTS and CONDITION1 execute expression EXP to give INFILE OUTFILE with STACK1 and HEAPETY1 and produce RSIDE .@ex15
TABLES from the type TYPE get type definition TYPE1 .@ty13
TABLES with STACK1 and HEAPETY1 the ADDRESS of TYPE1 must become RSIDE with new STACK and HEAPETY.@op1

H005 TABLES with STMTS and CONDITION execute single procedure SUBCALL call to give CONDITION1:
TABLES with STMTS and CONDITION execute subroutine invocation procedure SUBCALL call to give CONDITION1 and produce EMPTY.@ru1
TABLES with STMTS and CONDITION execute single if EXP
then STMTSETY ELSESETY fi to give CONDITION1:
TABLES with STMTS and CONDITION perform expression EXP
to give CONDITION2 and eventually prepare konst TRUE in
one table tsnok ex41
TABLES with STMTS and CONDITION2 execute might empty
STMTSETY to give CONDITION1 @ex2
TABLES with STMTS and CONDITION perform expression EXP
to give CONDITION2 and eventually prepare konst FALSE
in one table tsnok @ex41
TABLES with STMTS and CONDITION2 execute else part of
ELSESETY to give CONDITION1.@ex7

TABLES with STMTS and CONDITION execute else part of
ELSESETY to give CONDITION1:
where ELSESETY is else STMTSETY ,
TABLES with STMTS and CONDITION execute might empty
STMTSETY to give CONDITION1 @ex2
where ELSESETY is EMPTY ,
where CONDITION1 is CONDITION.

TABLES with STMTS and CONDITION execute single case EXP
of CASEELEMS pan to give CONDITION1:
TABLES with STMTS and CONDITION perform expression EXP
to give CONDITION2 and eventually prepare
KONSTANT @ex41
TABLES with STMTS and CONDITION2 execute atmost one of
the items of case statement CASEELEMSETY due to KONSTANT
to CONDITION1 @ex9
where CASEELEMSETY is EMPTY ,
TABLES with STMTS and CONDITION perform expression EXP
to give CONDITION1 and eventually prepare KONSTANT.@ex41

TABLES with STMTS and CONDITION execute atmost one of the
items of case statement CASEELEM CASEELEMSETY due to
KONSTANT to give CONDITION:
TABLES with STMTS and CONDITION execute single case
element CASEELEM due to KONSTANT to give
CONDITION1 @ex10
TABLES with STMTS and CONDITION execute atmost one of
items of case statement CASEELEMSETY due to KONSTANT
to give CONDITION1.@ex9

TABLES with STMTS and CONDITION execute single case
element label KONSTANTS end label STMTSETY due to KONSTANT
 to give CONDITION1:
where KONSTANTS contains KONSTANT ,
TABLES with STMTS and CONDITION execute might empty
STMTSETY to give CONDITION1,@ex2
H011 TABLES with STMTS and CONDITION execute single while EXP do STMTSETY od to give CONDITION1:

TABLES with STMTS and CONDITION perform expression EXP to give CONDITION2 and eventually prepare konst TRUE in one table tsnok. @ex41

TABLES with STMTS and CONDITION2 execute might empty STMTSETY to give CONDITION3. @ex2

TABLES with STMTS and CONDITION3 execute single while EXP do STMTSETY od to give CONDITION1. @ex11

TABLES with STMTS and CONDITION perform expression EXP to give CONDITION1 and eventually prepare konst FALSE in one table tsnok. @ex41

H012 TABLES with STMTS and CONDITION execute single repeat STMTSETY until EXP tasep to give CONDITION1:

TABLES with STMTS and CONDITION execute might empty STMTSETY to give CONDITION2. @ex2

TABLES with STMTS and CONDITION2 execute single while not EXP do STMTSETY od to give CONDITION1. @ex11

H013 TABLES with STMTS and CONDITION execute single for NAME becomes EXP through to EXP1 do STMTSETY od to give INFILE OUTFILE with STACK and HEAPETY:

TABLES with STMTS and CONDITION execute single assign left NAME right becomes EXP stmt to give CONDITION1. @ex4

TABLES with STMTS and CONDITION1 execute single while left NAME right less or equal to EXP1 do STMTSETY assign left NAME right becomes function #SUCC with actuals comma left NAME right call stmt od to give INFILE OUTFILE with STACK1 and HEAPETY1. @ex11

TABLES and STACK1 the left hand side entire left NAME right produce ADDRESS of TYPE. @ex24

STACK1 and HEAPETY put the contents of undefined in ADDRESS to have STACK and HEAPETY. @op7

H014 TABLES with STMTS and CONDITION execute single for NAME becomes EXP through down to EXP1 do STMTSETY od to give INFILE OUTFILE with STACK and HEAPETY:

TABLES with STMTS and CONDITION execute single assign left NAME right becomes EXP stmt to give CONDITION1. @ex4

TABLES with STMTS and CONDITION1 execute single while left NAME right greater or equal to EXP1 do STMTSETY assign left NAME right becomes function #PRED with actuals comma left NAME right call stmt od to give INFILE OUTFILE with STACK1 and HEAPETY. @ex11

TABLES and STACK1 the left hand side entire left NAME right produce ADDRESS of TYPE. @ex24

STACK1 and HEAPETY put the contents of undefined in ADDRESS to have STACK and HEAPETY. @op7
H015 TABLES with STMTS and CONDITION execute expression EXP to give CONDITION1 and produce RSIDE:
  TABLES with STMTS and CONDITION execute simple expression EXP to give CONDITION1 and produce RSIDE i\text{ex}16
  TABLES with STMTS and CONDITION execute compound expression EXP to give CONDITION1 and produce RSIDE i\text{ex}22

H016 TABLES with STMTS and CONDITION execute simple expression SEXP to give CONDITION1 and produce RSIDE:
  TABLES with STMTS and CONDITION execute term SEXP to give CONDITION1 and produce RSIDE i\text{ex}17
  TABLES with STMTS and CONDITION execute signed expression SEXP to give CONDITION1 and produce RSIDE i\text{ex}20
  TABLES with STMTS and CONDITION execute adding expression SEXP to give CONDITION1 and produce RSIDE i\text{ex}21

H017 TABLES with STMTS and CONDITION execute term TERM to give CONDITION1 and produce RSIDE:
  TABLES with STMTS and CONDITION execute factor TERM to give CONDITION1 and produce RSIDE i\text{ex}18
  TABLES with STMTS and CONDITION execute multiply expression TERM to give CONDITION1 and produce RSIDE i\text{ex}19

H018 TABLES with STMTS and CONDITION execute factor FACTOR to give CONDITION1 and produce RSIDE:
  TABLES with STMTS and CONDITION execute variable FACTOR to give CONDITION1 and produce RSIDE i\text{ex}38
  TABLES with STMTS and CONDITION execute unsigned constant FACTOR to give CONDITION1 and produce RSIDE i\text{ex}57
  TABLES with STMTS and CONDITION execute bracketed FACTOR to give CONDITION1 and produce RSIDE i\text{ex}59
  TABLES with STMTS and CONDITION execute set FACTOR to give CONDITION1 and produce RSIDE i\text{ex}59
  TABLES with STMTS and CONDITION execute negative FACTOR to give CONDITION1 and produce RSIDE i\text{ex}70
  TABLES with STMTS and CONDITION execute subroutine invocation FACTOR to give CONDITION1 and produce RSIDE i\text{ru}1
H019 TABLES with STMTS and CONDITION execute multiply expression TERM MULOP FACTOR to give INFILE OUTFILE with STACK and HEAPETY and produce DBOX:

TABLES with STMTS and CONDITION execute term expression TERM to give INFILE1 OUTFILE1 with STACK1 and HEAPETY1 and produce RSIDE1 @ex17
STACK1 and HEAPETY1 the RSIDE1 eventually introduces DBOX1 @op14.

TABLES with STMTS and INFILE1 OUTFILE1 with STACK1 and HEAPETY1 execute factor expression FACTOR to give INFILE OUTFILE with STACK and HEAPETY and produce RSIDE2 @ex19
STACK and HEAPETY the RSIDE2 eventually introduces DBOX2 @op14
scalar value or set operation DBOX1 MULOP DBOX2 give DBOX.@op25,57-59

H020 TABLES with STMTS and CONDITION execute signed expression UOP TERM to give INFILE OUTFILE with STACK and HEAPETY and produce konst UOP1 UNUMBER tsnok:

TABLES with STMTS and CONDITION execute term expression TERM to give INFILE OUTFILE with STACK and HEAPETY and produce RSIDE @ex17
STACK and HEAPETY the RSIDE eventually introduces konst UOP2 UNUMBER tsnok @op14
the unary sign UOP and UOP2 make UOP1.@de20

H021 TABLES with STMTS and CONDITION execute adding expression SEXP ADDOP TERM to give INFILE OUTFILE with STACK and HEAPETY and produce DBOX:

TABLES with STMTS and CONDITION execute simple expression SEXP to give INFILE1 OUTFILE1 with STACK1 and HEAPETY1 and produce RSIDE1 @ex16
STACK1 and HEAPETY1 the RSIDE1 eventually introduces DBOX1 @op14
TABLES with STMTS and INFILE1 OUTFILE1 with STACK1 and HEAPETY1 execute term TERM to give INFILE OUTFILE with STACK and HEAPETY and produce RSIDE2 @ex17
STACK and HEAPETY the RSIDE2 eventually introduces DBOX2 @op14
scalar value or set operation DBOX1 ADDOP DBOX2 give DBOX.@op25,65.
H022 TABLES with STMTS and CONDITION execute compound expression SEXP1 RELOP SEXP2 to give INFILE OUTFILE with STACK and HEAPETY and produce konst IDEN tsnok:
TABLES with STMTS and CONDITION execute simple expression SEXP1 to give INFILE1 OUTFILE1 with STACK1 and HEAPETY1 and produce RSIDE1 @ex16
STACK1 and HEAPETY1 the RSIDE1 will give value or set or pointer or sequence DBOXORSTRG1 @Op68
TABLES with STMTS and INFILE1 OUTFILE1 with STACK1 and HEAPETY1 execute simple expression SEXP2 to give INFILE OUTFILE with STACK and HEAPETY and produce RSIDE2 @ex16
STACK and HEAPETY the RSIDE2 will give value or set or pointer or sequence DBOXORSTRG2 @Op68
TABLES the four types of relational operation DBOXORSTRG1 RELOP DBOXORSTRG2 introduce the boolean IDEN.@ex71

H023 TABLES with STMTS and INFILE OUTFILE with STACK and HEAPETY execute might function left variable VARIABLE to give CONDITION and produce TYPADRES:
TABLES and STACK the left hand side entire VARIABLE produce TYPADRES @ex24
where CONDITION is INFILE OUTFILE with STACK and HEAPETY:
TABLES with STMTS and INFILE OUTFILE with STACK and HEAPETY execute component variable VARIABLE to give CONDITION and produce TYPADRES in left side @ex36
TABLES with STMTS and INFILE OUTFILE with STACK and HEAPETY execute referenced variable VARIABLE to give CONDITION and produce TYPADRES in left side.@ex56

H024 TABLES and FRAMESETY FRAME the left hand side left NAME right produce TYPADRES:
TABLES and FRAMESETY FRAME have NAME as variable or parameter to give TYPADRES @ex25
TABLES and FRAMESETY FRAME obtain NAME from top head to give TYPADRES @ex32
where loc NAME refers is not in in FRAME,
where block NAME be is not in FRAME,
FRAME top frame points to ONES table @ex27
TABLES investigate for NAME in ONES table to be not a scalar identifier @ex35
FRAMES find the top frame linkage to be ONES1 @ex33
back to ONES1 frame FRAMESETY to be FRAMES @ex34
TABLES and FRAMES the left hand side entire left NAME right produce TYPADRES.@ex24

H025 TABLES and STACK have NAME as variable or parameter to give TYPADRES:
TABLES and STACK the NAME is variable so give TYPADRES @ex26
TABLES and STACK the NAME is value formal parameter so give TYPADRES @ex29
TABLES and STACK the NAME is variable formal parameter so give TYPADRES.@ex30
H026 TABLES and FRAMESETY FRAME the NAME is variable so give stack ONESth box of TYPE :

where FRAME contains loc NAME refers the ONESth BOX ht ,
FRAME top frame points to ONES1 table \texttt{@ex27}
TABLES involve loc NAME be variable of TYPE1 end exactly
in ONES1 table \texttt{@ex28}
TABLES from type TYPE1 get type definition TYPE.@ty13

H027 FRAMESETY the ONESth frame from ONES1 table \texttt{FRAME} link to
ONESETY frame top frame points to ONES2 table :
where ONES2 is ONES1,

H028 TABLESETY the ONESth LOCSETY slink to ONESETY table
TABLESETY1 involve LOC exactly in ONES1 table :
where ONES is ONES1 ,
where LOCSETY contains LOC.

H029 TABLES and FRAMESETY FRAME the NAME is value formal
parameter so give stack ONESth box of TYPE :

where FRAME contains loc NAME refers the ONESth BOX ht ,
FRAME top frame points to ONES1 table \texttt{@ex27}
TABLES involve loc NAME be formal parameter of value of
IDEN end exactly in ONES1 table \texttt{@ex28}
TABLES the type identifier IDEN introduce TYPE.@ty12

H030 TABLES and FRAMESETY FRAME the NAME is variable formal
parameter so give TYPADRES :
where FRAME contains loc NAME refers the ONESth ADDRESS ht ,
FRAMESETY FRAME by way of stack address ADDRESS find the
NAME1 and new STACK \texttt{@ex31}
TABLES and STACK have NAME1 as variable or parameter to
give TYPADRES.@ex25

H031 FRAMESETY the ONESth frame from ONES1 table HEADETY and
FLOCSETY loc NAME refers the ONES2th BOX ht NBOXESERTY
end FLOCSETY1 link to ONESETY frame FRAMESETY1 by way of
stack address stack ONESth box find the NAME1 and new
STACK :
where ONES2 is ONES3 ,
where NAME1 is NAME ,
where STACK is FRAMESETY the ONESth frame from ONES1
table HEADETY and FLOCSETY loc NAME refers the ONES2th
BOX ht NBOXESERTY end FLOCSETY1 link to ONESETY frame.

H032 TABLES and FRAMESETY FRAME obtain NAME from top head to
give stack ONESth box of TYPE :
where FRAME contains the block NAME be function refers
the ONESth BOX ht ,
FRAME find the top frame linkage to be ONES1 \texttt{@ex33}
back to ONES1 frame FRAMESETY to be FRAMES \texttt{@ex34}
FRAMES top frame points to ONES2 table \texttt{@ex27}
TABLES involve loc NAME be function of PARNUMETY type
IDEN and ONE53 as its local table end exactly in ONES2
table \texttt{@ex28}
TABLES the type identifier IDEN introduce TYPE.@ty12.
H033 the ONESth frame from ONES1 table FRAIM link to ONES2 frame find the top frame linkage to be ONES3 where ONES3 is ONES2.

H034 back to ONES frame FRAMESETY the ONES1th frame from ONES2 table FRAIM link to ONESETY frame FRAMESETY1 to be FRAMES where ONES1 is ONES where FRAMES is FRAMESETY the ONES1th frame from ONES2 table FRAIM link to ONESETY frame.

H035 TABLESETY the ONESth LOCSETY slink to ONESETY table TABLESETY1 investigate for NAME in ONES1 table to be not a scalar identifier where ONES is ONES1 left NAME right should not be an scalar item of LOCSETY.

H036 TABLES with STMTS and CONDITION execute component variable COMOPVAR to give CONDITION1 and produce TYPADRES LEFTETY:
- TABLES with STMTS and CONDITION execute indexed variable COMPVAR to give CONDITION1 and produce TYPADRES LEFTETY @ex37
- TABLES with STMTS and CONDITION execute field designator variable COMPVAR to give CONDITION1 and produce TYPADRES LEFTETY.@ex46

H037 TABLES with STMTS and CONDITION execute indexed variable matrix VARIABLE to give CONDITION2 and produce STKHP ONESth box of TYPEDEF needs UINT boxes LEFTETY:
- TABLES with STMTS and CONDITION execute indexed VARIABLE to give CONDITION2 and produce STKHP ONES2th box of array defined by ONES3 dimensional INDEXES of TYPE needs UINT1 boxes LEFTETY @ex38
- TABLES from the type TYPE get type definition TYPEDEF needs UINT boxes @ty13
- TABLES with STMTS and CONDITION2 the group of expression COMMAS result KONSTANTS and CONDITION @ex40
- TABLES with KONSTANTS by the help of INDEXES present sum uplus zero next INT @ex42
the integer INT1 equal to integer INT times integer uplus UINT @op27
the integer uplus ONES equal to integer ONES2 plus integer INT1.@op27
TABLES with STMTS and INFILE OUTFILE with STACK and HEAPETY execute variable VARIABLE to give CONDITION and produce TYPADRES LEFTETY:

TABLES with STMTS and STACK the right hand side entire VARIABLE produce TYPADRES @ex39

where CONDITION is INFILE OUTFILE with STACK and HEAPETY

TABLES with STMTS and INFILE OUTFILE with STACK and HEAPETY execute component variable VARIABLE to give CONDITION and produce TYPADRES LEFTETY @ex36

TABLES with STMTS and INFILE OUTFILE with STACK and HEAPETY execute referenced VARIABLE to give CONDITION and produce TYPADRES LEFTETY @ex56

TABLES and FRAMESET FRAME the right hand side entire left NAME right produce TYPADRES:

TABLES and FRAMESET FRAME have NAME as variable or parameter to give TYPADRES @ex25

where loc NAME refers is not in FRAME,

FRAME top frame points to ONES table @ex27

TABLES investigate foe NAME in ONES table to be not a scalar identifier @ex35

FRAMES find the top frame linkage to be ONES1 @ex33

back to ONES1 frame FRAMESET to be FRAMES @ex34

TABLES and FRAMES the right hand side entire left NAME right produce TYPADRES @ex39

TABLES with STMTS and CONDITION the group of expressions comma EXP amoc COMMASETY result KONSTANT KONSTANTSETY and INFILE OUTFILE with STACK and HEAPETY:

TABLES with STMTS and CONDITION perform expression EXP to give INFILE1 OUTFILE1 with STACK1 and HEAPETY1 and eventually prepare KONSTANT @ex41

TABLES with STMTS and INFILE1 OUTFILE1 with STACK1 and HEAPETY1 the group of expressions COMMASETY result KONSTANTSETY and INFILE OUTFILE with STACK and HEAPETY @ex40

where COMMASETY KONSTANTSETY is EMPTY,

TABLES with STMTS and CONDITION perform expression EXP to give INFILE OUTFILE with STACK and HEAPETY and eventually prepare KONSTANT @ex41

TABLES with STMTS and CONDITION perform expression EXP to give INFILE OUTFILE with STACK and HEAPETY and eventually prepare KONSTANT:

TABLES with STMTS and CONDITION execute expression EXP to give INFILE OUTFILE with STACK and HEAPETY and produce RSIDE @ex15

STACK and HEAPETY the RSIDE eventually introduces KONSTANT @op14
H042 TABLES with KONSTANT KONSTANTSETY by the help of INDEX INDEXESETY present sum INT next INT1:
TABLES with KONSTANT from INDEX will build indice INT2 and upper bound INT3 @ex43-45
the integer INT4 equal to integer INT times integer INT3 @op27
the integer INT5 equal to integer INT4 plus integer INT2 @op27
the integer INT6 equal to integer INT5 minus integer INT2 @op27
TABLES with KONSTANTSETY by the help of INDEXESETY present sum INT6 next INT1 @ex42
where KONSTANTSETY INDEXESETY is EMPTY,
TABLES with KONSTANT from INDEX will build indice INT2 and upper bound INT3 @ex43-45
the integer INT4 equal to integer INT times integer INT3 @op27
the integer INT5 equal to integer INT4 plus integer INT2 @op27
the integer INT1 equal to integer INT5 minus integer INT2 @op27

H043 TABLES with konst NAME in ONES table tsnok from index of left NAMETY in ONES1 table right will build indice INT and upper bound INT1:
where ONES is ONES1,
TABLES from type NAMETY in ONES table get type definition SCRG defined by BNAMESETY BNAME needs one boxes @ty13
find the order of identifier left NAME right in BNAMESETY BNAME to be INT @op89
find the order of identifier BNAME in BNAMESETY BNAME to be INT1 @op89

H044 TABLES with konst KCHAR tsnok from index of left NAMETY in ONES table right will build indice INT and upper bound INT1:
TABLES from type NAMETY in ONES table get type definition SCRG defined by KCHARSETY KCHAR1 needs one boxes @ty13
find the order of character KCHAR in KCHARSETY KCHAR1 to be INT @op74
find the order of character KCHAR1 in KCHARSETY KCHAR1 to be INT1 @op74

H045 TABLES with konst INT tsnok from index of left TYPE right will build indice INT and upper bound INT1:
TABLES from type TYPE get type definition SCRG defined by integer from INT3 to INT1 needs one boxes @ty12
the integer INT is in the interval integer from INT3 to INT1 @op31
H046 TABLES with STMTS and CONDITION execute field designator variable field VARIABLE point NAME select to give INFILE OUTFILE with STACK and HEAPETY and produce ADDRESS of TYPE LEFTETY:

TABLES with STMTS and CONDITION execute variable VARIABLE to give INFILE OUTFILE with STACK1 and HEAPETY1 ADDRESS1 of record defined by rekord FIELDLIST end needs UINT boxes LEFTETY1 @ex23

TABLES with STACK1 and HEAPETY1 the selector NAME of TYPE with the base ADDRESS1 in term of FIELDLIST give ADDRESS LEFTETY and STACK and HEAPETY. @ex47

H047 TABLES with STACK and HEAPETY the selector NAME of TYPE with the base ADDRESS in term of FIELDLIST give ADDRESS1 LEFTETY and STACK1 and HEAPETY1:

TABLES find the fixed part FIELDLIST which involve the selector NAME of TYPE with base ADDRESS give ADDRESS1 @ex48

where STACK1 and HEAPETY1 is STACK and HEAPETY1:

TABLES with STACK and HEAPETY the fixed and variant part FIELDLIST involving NAME of TYPE with base ADDRESS give ADDRESS1 LEFTETY and STACK1 and HEAPETY1 @ex49

TABLES with STACK and HEAPETY the variant part FIELDLIST including NAME of TYPE with base ADDRESS give ADDRESS1 LEFTETY and STACK1 and HEAPETY1. @ex50

H048 TABLES with STACK and HEAPETY the selector NAME of TYPE with offset UINT endsel SELECTIONSETY1 which involve the selector NAME1 of TYPE1 with base STKHP ONES1th box give STKHP1 ONES1th box:

where NAME is NAME1,

TABLES from the type TYPE get type definition TYPE1 @ty13

where STKHP is STKHP1,

the integer uplus ONES1 equals to integer uplus ONES plus integer uplus UINT @op27

H049 TABLES with STACK and HEAPETY the fixed and variant part SELECTIONS VARPART involving NAME of TYPE with base ADDRESS give ADDRESS1 LEFTETY and STACK1 and HEAPETY1:

TABLES the fixed part SELECTIONS which involve the selector NAME of TYPE with base ADDRESS give ADDRESS1 @ex48

where STACK1 and HEAPETY1 is STACK and HEAPETY1:

TABLES with STACK and HEAPETY the variant part VARPART including NAME of TYPE with base ADDRESS give ADDRESS1 LEFTETY and STACK1 and HEAPETY1. @ex50
TABLES with STACK and HEAPETY the variant part case

SELECTIONETY EXTRA of VARIANTS including NAME of TYPE with base ADDRESS give STKHP ONESTh TAGETY box LEFTETY and STACK1 and HEAPETY1:

TABLES the fixed part SELECTIONETY which involve the selector NAME of TYPE with base ADDRESS give STKHP ONESTh box @ex48

TAGETY will be deriven from LEFTETY @ex51
where STACK1 and HEAPETY1 is STACK and HEAPETY
where TAGETY is EMPTY.

TABLES with STACK and HEAPETY the variants VARIANTS containing NAME of TYPE with base ADDRESS give STKHP ONESTh box with EXTRA LEFTETY and STACK1 and HEAPETY1 @ex52

TABLES with STACK and HEAPETY the variant part case
SELECTIONETY EXTRA of VARIANTS including NAME of TYPE with base ADDRESS give STKHP ONESTh TAGETY box LEFTETY and STACK1 and HEAPETY1:

TABLES the fixed part SELECTIONETY which involve the selector NAME of TYPE with base ADDRESS give STKHP ONESTh box @ex48

TAGETY will be deriven from LEFTETY @ex51
where STACK1 and HEAPETY1 is STACK and HEAPETY
where TAGETY is EMPTY.

TABLES with STACK and HEAPETY the variants VARIANTS containing NAME of TYPE with base ADDRESS give STKHP ONESTh box with EXTRA LEFTETY and STACK1 and HEAPETY1 @ex52

TABLES with STACK and HEAPETY the variants VARIANTS containing NAME of TYPE with base ADDRESS give STKHP ONESTh box with EXTRA LEFTETY and STACK1 and HEAPETY1 @ex52

TABLES with STACK and HEAPETY the selector NAME of TYPE with the base ADDRESS in term of FIELDLIST give ADDRESS1 LEFTETY @ex47

TABLES with STACK and HEAPETY execution process for EXTRA with base ADDRESS and label KONSTANTS in term of LEFTETY and STACK1 and HEAPETY1 @ex54
TABLES with STACK and HEAPETY execution process for extra
TYPE with offset UINT endext with base STKHP ONESith box
and label KONSTANTS in term of LEFTETY and STACK1 and
HEAPETY1:

where LEFTETY is in left side,
the integer uplus ONES1 equal to integer uplus ONES plus
integer uplus UINT @op27
STACK and HEAPETY the STKHP ONESith box contains of the
amount BOX @op16
STACK and HEAPETY unless BOX is already one of the
KONSTANTS insert it in STKHP ONESith box to have STACK1
and HEAPETY1 @ex55
where LEFTETY is EMPTY,
the integer uplus ONES1 equal to integer uplus ONES plus
integer uplus UINT @op27
STACK and HEAPETY the STKHP ONESith box contains of the
amount KONSTANT @op16
where KONSTANTS contains KONSTANT,
where STACK1 and HEAPETY1 is STACK and HEAPETY.

TABLES with STACK and HEAPETY unless BOX is already one of the
KONSTANT KONSTANTSETY insert it in ADDRESS to have STACK1
and HEAPETY1:

where KONSTANT KONSTANTSETY contains BOX,
where STACK1 and HEAPETY1 is STACK and HEAPETY
STACK and HEAPETY put the quantity KONSTANT in ADDRESS
to have STACK1 and HEAPETY1 @op6

TABLES with STMTS and CONDITION execute referenced
variable reference VARIABLE arrow to give INFILEOUTFILE
with STACK and HEAPETY and produce heap ONESith box of
TYPE:

TABLES with STMTS and CONDITION execute variable
VARIABLE to give INFILEOUTFILE with STACK and HEAPETY
and produce ADDRESS of pointer defined by IDEN needs one
boxes LEFTETY @ex23
STACK and HEAPETY the ADDRESS contains of the amount
heap ONESith box @op16
TABLES the type identifier IDEN introduce TYPE.@ty12

TABLES with STMTS and CONDITION execute unsigned constant
CONST to give CONDITION1 and produce CONST1:

where CONST1 is CONST,
where CONDITION1 is CONDITION.

TABLES with STMTS and CONDITION execute bracketed open
bracket EXP close bracket to give CONDITION1 and produce
RSIDE:

TABLES with STMTS and CONDITION execute expression EXP
to give CONDITION1 and produce RSIDE.@ex15

TABLES with STMTS and CONDITION execute set subset
ELEMENTSETY tesbus to give CONDITION1 and produce subset
ELEMENTSETY1 tesbus:

TABLES with STMTS and CONDITION calculate set elements
ELEMENTSETY to give CONDITION1 and produce
ELEMENTSETY1 @ex60
where ELEMENTSETY ELEMENTSETY1 is EMPTY.
TABLES with STMTS and CONDITION calculate set elements
ELEMENT ELEMENTSErTy to give CONDITION1 and produce
ELEMENTSErTy1 ELEMENTSErTy2

TABLES with STMTS and CONDITION calculate single set
member ELEMENT to give CONDITION2 and produce
ELEMENTSErTy1 ,@ex61,62

TABLES with STMTS and CONDITION2 calculate set elements
ELEMENTSErTy to give CONDITION1 and produce
ELEMENTSErTy2 @ex60

where ELEMENTSErTy ELEMENTSErTy2 is EMPTY ,

TABLES with STMTS and CONDITION calculate single set
member ELEMENT to give CONDITION1 and produce
ELEMENTSErTy1 ,@ex61,62

TABLES with STMTS and CONDITION calculate single set
member ELEMENT to give INFILE OUTFILE with STACK and
HEAPETY and produce elem KONSTANT mele

TABLES with STMTS and CONDITION perform expression EXP
to give INFILE OUTFILE with STACK and HEAPETY and
eventually prepare KONSTANT.@ex41

TABLES with STMTS and CONDITION calculate single set
member elem EXP mele to give INFILE OUTFILE with STACK and
HEAPETY and produce elem KONSTANT mele

TABLES with STMTS and CONDITION perform expression EXP1
to give CONDITION1 eventually and prepare
KONSTANT1 ,@ex41

TABLES with STMTS and CONDITION perform expression EXP2
to give INFILE OUTFILE with STACK and HEAPETY and
eventually prepare KONSTANT2 ,@ex41

TABLES fill the gap between KONSTANT1 and KONSTANT2 to
build ELEMENTSErTy. @ex63

TABLES fill the gap between KONSTANT and KONSTANT1 to
build ELEMENTSErTy :

fill the space between integers KONSTANT and KONSTANT1
to build ELEMENTSErTy. @ex64

fill the space between characters KONSTANT and KONSTANT1
to build ELEMENTSErTy. @ex66

TABLES fill the space between scalar identifiers
KONSTANT and KONSTANT1 to build ELEMENTSErTy. @ex68

TABLES fill the space between integers konst INT tsnok and konst
INT1 tsnok to build ELEMENTSErTy :
the integer INT less than integer INT1 ,@op83
design all integer members ELEMENTSErTy from INT through
INT. @ex65

where INT is INT1 ,

where ELEMENTSErTy is elem konst INT tsnok mele ;
the integer INT greater than integer INT1 ,@op83
where ELEMENTSErTy is EMPTY.
H065 design all integer members elem konst INT tsnok mele
ELEMENTSETY from INT1 through INT2:
the integer INT3 equal to integer INT1 plus integer
uplus one $\Theta$op27
the integer INT3 less or equal1 to integer INT2 $\Theta$op83
where INT is INT1,
design all integer members ELEMENTSETY from INT3 through
INT2 $\Theta$ax65
the integer INT3 equal to integer INT1 plus integer uplus
one $\Theta$op27
the integer INT3 grater than integer INT2 $\Theta$op83
where INT is INT1.

H066 fill the space between characters konst KCHAR tsnok and
konst KCHAR1 tsnok to build ELEMENTSETY:
where KARSET contains KCHAR KCHARSETY KCHAR1,
built all character members ELEMENTSETY from KCHAR
KCHARSETY KCHAR1 $\Theta$ex67
where KCHAR is KCHAR1,
where ELEMENTSETY is elem konst KCHAR tsnok mele if
where KARSET contains KCHAR1 KCHARSETY KCHAR,
where ELEMENTSETY is EMPTY.

H067 build all character members elem konst KCHAR tsnok
ELEMENTSETY from KCHAR1 KCHARSETY:
where KCHAR is KCHAR1,
built all character members ELEMENTSETY from
KCHARSETY $\Theta$ex67
where ELEMENTSETY KCHARSETY is EMPTY,
where KCHAR is KCHAR1.

H068 TABLESETY the ONESth LOCSETY loc NAMETY be type of scalar
defined by BNAMES needs one boxes end LOCSETY1 slink to
ONESETY table TABLESETY1 fill the space between scalar
identifiers konst NAME1 in ONES1 table tsnok and konst
NAME2 in ONES2 table tsnok:
where ONES is ONES1,
where ONES is ONES2,
where BNAMES contains left NAME1 right BNAMESETY left
NAME2 right,
construct all identifiers members ELEMENSETY from left
NAME1 right BNAMESETY left NAME2 right at ONES
table $\Theta$ex69
where NAME1 is NAME2,
where ELEMENTSETY is elem konst NAME1 in ONES table
tsnok mele;
where BNAMES contains left NAME2 right BNAMESETY left
NAME1 right,
where ELEMENTSETY is EMPTY.

H069 construct all identifiers elem konst IDEN tsnok mele
ELEMENTSETY from left NAME right BNAMESETY at ONES table:
where IDEN is NAME in ONES table;
construct all identifiers ELEMENTSETY from BNAMESETY at
ONES table $\Theta$ex69
where ELEMENTSETY BNAMESETY is EMPTY,
where IDEN is NAME in ONES table.
H070 TABLES with STMTS and CONDITION execute negative not FACTOR to give INFILE OUTFILE with STACK and HEAPETY and produce konst IDEN tsnok:

TABLES with STMTS and CONDITION execute factor FACTOR to give INFILE OUTFILE with STACK and HEAPETY and produce RSIDE, 3ex18 STACK and HEAPETY the RSIDE eventually introduces konst IDEN1 tsnok, @op14.

the boolean value IDEN is opposite of the boolean value IDEN1.@op99,100
Runtime semantics for subroutine invocations

H001 TABLES with STMTS and CONDITION execute subroutine invocation FUNPROC SUBCALL call to give CONDITION1 and produce RESULTETY:
   TABLES with STMTS and CONDITION evaluate ordinary FUNPROC SUBCALL call to give CONDITION1 and RESULTETY @ru2
   TABLES with STMTS and CONDITION evaluate parameter FUNPROC SUBCALL call to give CONDITION1 and RESULTETY @ru15
   TABLES with STMTS and CONDITION evaluate standard FUNPROC SUBCALL call to give CONDITION1 and RESULTETY.@ru20

H002 TABLES with STMTS and INFILE OUTFILE with STACK and HEAPETY evaluate ordinary FUNPROC NAME with actuals COMMASETY call to give CONDITION and RESULTETY:
   TABLES and STACK looking for memory location of subroutine NAME in ONES frame from ONES1 table @ru3
   TABLES involve loc NAME be FUNPROC of PARNUMETY type TYPE and ONES2 as its local table end exactly in ONES1 table @ex28
   TABLES with STMTS and INFILE OUTFILE with STACK and HEAPETY actions for ordinary FUNPROC NAME with actuals COMMASETY in ONES frame in ONES1 table and ONES2 as local table to give CONDITON and RESULTETY.@ru4

H003 TABLES and FRAMESETY the ONESth frame from ONES1 table FRAIM link to ONESETY frame looking for memory location of subroutine NAME in ONES2 frame from ONES3 table:
   where FRAIM contains loc NAME refers ,
   where ONES2 is ONES ,
   where ONES3 is ONES1 ;
   where loc NAME refers is not in FRAIM ,
   TABLES investigate for NAME in ONES1 table to be not a scalar identifier @ex35
   back to,ONESETYth frame FRAMESETY to be FRAMES @ex34
   TABLES and FRAMES looking for memory location of subroutine NAME in ONES2 frame from ONES3 table.@ru3
H004 TABLES with STMTS and INFILE OUTFILE with STACK and
HEAPETY actions for ordinary FUNPROC NAME with actuals
COMMASETY in ONES frame in ONES1 table and ONES2 as local
table to give INFILE1 OUTFILE1 with STACK1 and HEPETY1
and RESULTETY :
TABLES open new frame on STACK from ONES2 table to
introduce STACK2 with FUNPROC head NAME and linkage
ONES .@fr18 TABLES with STMTS and INFILE OUTFILE with STACK2 and
HEAPETY do the COMMASETY parameter mechanism to give
last INFILE2 OUTFILE2 with STACK3 and HEPETY2 .@ru5
TABLES with STMTS and INFILE2 OUTFILE2 with STACK3 and
HEAPETY2 to execute NAME in its ONES1 table to give
INFILE1 OUTFILE1 with STACK1 the ONES3th frame from
ONES2 table the block NAME be FUNPROC refers NBOXETY and
FLOCSETY link to ONES frame .@ex1
RESULTETY is precisely defined by FUNPROC and TYPE and
NBOXETY .@ru13.14

H005 TABLES with STMTS and CONDITION do the COMMASETY parameter
mechanism to give last CONDITION1 :
TABLES with STMTS and CONDITION arrange the oneth
parameter mechanism COMMASETY to give CONDITION1 .@ru6
where COMMASETY is EMPTY ,
where CONDITION1 is CONDITION.

H006 TABLES with STMTS and CONDITION arrange the ONESth
parameter mechanism COMMA COMMASETY to give CONDITION1 :
TABLES with STMTS and CONDITION do single ONESth
parameter device COMMA to give CONDITION2 .@ru7
TABLES with STMTS and CONDITION2 arrange ONES oneth
parameter mechanism COMMASETY to give CONDITION1 .@ru6
where COMMASETY is EMPTY ,
TABLES with STMTS and CONDITION do single ONESth
parameter device COMMA to give CONDITION1 .@ru7.

H007 TABLES with STMTS and CONDITION do single ONESth parameter
device COMMA to give CONDITION1 :
TABLES with STMTS and CONDITION value ONESth parameter
process COMMA will give CONDITION1 .@ru8
TABLES with STMTS and CONDITION variable ONESth
parameter process COMMA will give CONDITION1 .@ru11.
TABLES with STMTS and CONDITION subroutine ONESth
parameter process COMMA will give CONDITION1 .@ru12
H008 TABLES with STMTS and INFILE OUTFILE with FRAMESETY the ONESt frame from ONESt table HEADETY and FLOCSETY loc NAME refers the ONESt BOX ht NOBOXESETY end FLOCSETY1 link to ONESETY frame and HEAPETY value ONESETY one parameter comma EXP ammoc will give INFILE1 OUTFILE1 with STACK and HEAPETY1:

ONESETY1 would be balanced with FLOCSETY @ru9 TABLES involve loc NAME be formal parameter of value of IDEN end exactly in ONESt table @ex28 TABLES the type identifier IDEN introduce TYPE @ty12 TABLES with STMTS and INFILE OUTFILE with FRAMESETY and HEAPETY execute expression EXP to give INFILE1 OUTFILE1 with FRAMESETY1 and HEAPETY1 and produce RSIDE @ex15 TABLES with FRAMESETY1 the ONESt frame from ONESt table HEADETY and FLOCSETY1 loc NAME refers the ONESt BOX ht end FLOCSETY1 link to ONESETY frame and HEAPETY1 the stack ONESt box of TYPE must become RSIDE with new STACK1 and HEAPETY1.3op1

H009 ONESETY would be balanced with FLOCSETY:

the nonempty ONESETY enumerate FLOCSETY @ru10 where ONESETY FLOCSETY is EMPTY.

H010 the nonempty one ONESETY enumerate FLOC FLOCSETY:

ONESETY would be balanced with FLOCSETY.@ru9

H011 TABLES with STMTS and INFILE OUTFILE with FRAMESETY the ONESt frame from ONESt table HEADETY and FLOCSETY loc NAME refers the ONESt BOX ht end FLOCSETY1 link to ONESETY frame and HEAPETY variable ONESETY1 oneth parameter process comma VARIABLE ammoc will give INFILE1 OUTFILE with STACK and HEAPETY1:

ONESETY1 would be balanced with FLOCSETY @ru9 TABLES involve loc NAME be formal parameter of value of IDEN end exactly in ONESt table @ex28 TABLES with STMTS and INFILE OUTFILE with FRAMESETY and HEAPETY execute variable VARIABLE to give INFILE1 OUTFILE1 with FRAMESETY1 and HEAPETY1 and produce ADDRESS of TYPE @ex38 where STACK is FRAMESETY the ONESt frame from ONESt table HEADETY and FLOCSETY loc NAME refers the ONESt ADDRESS ht end FLOCSETY1 link to ONESETY1 frame.

H012 TABLES with STMTS and INFILE OUTFILE with FRAMESETY the ONESt frame from ONESt table HEADETY and FLOCSETY loc NAME refers the ONESt BOX ht end FLOCSETY1 link to ONESETY frame and HEAPETY subroutine ONESETY1 oneth parameter process comma DNAME ammoc will give INFILE1 OUTFILE with STACK and HEAPETY1:

ONESETY1 would be balanced with FLOCSETY @ru9 TABLES involve loc NAME be formal parameter of FUNPROC of TYPE end exactly in ONESt table @ex28 where STACK is FRAMESETY the ONESt frame from ONESt table HEADETY and FLOCSETY loc NAME refers the ONESt DNAME ht end FLOCSETY1 link ONESETY frame , where INFILE1 OUTFILE1 and HEAPETY1 is INFILE OUTFILE and HEAPETY.
H013 stack ONESth box of TYPE is precisely defined by function and TYPE1 and the ONESth BOX ht:
where ONES is ONESl,
where TYPE is TYPE1.

H014 is precisely defined by procedure and void and EMPTY:
true.

H015 TABLES with STMTS and INFINE OUTFILE with STACK and HEAPETY evaluate parameter FUNPROC NAME with actuals
COMMASETY call to give CONDITION and RESULTETY:
TABLES and STACK looking for memory location of subroutine NAME in ONESt frame from ONESth table.
TABLES involve loc NAME be formal parameter of FUNPROC of TYPE end exactly in ONESt table.
STACK from NAME in ONESt frame take NAME1 and present new STACK1.
TABLES and STACK1 through path of FUNPROC NAME1 find
subname NAME2 in ONESt table with ONES3 as its local table.
TABLES with STMTS and INFINE OUTFILE with STACK and HEAPETY actions for ordinary FUNPROC NAME2 with actuals
COMMASETY in ONESt1 table and ONES3 as local table to give CONDITION and RESULTETY.
TABLES and STACK looking for memory location of subroutine NAME in ONESt frame from ONESt table.
TABLES involve loc NAME be formal parameter of FUNPROC of TYPE end exactly in ONESt table.
STACK from NAME in ONESt frame take DNAME and present new STACK1.
TABLES and STACK1 from the trace of FUNPROC DNAME give
#NAME1.
TABLES with STMTS and INFINE OUTFILE with STACK and HEAPETY evaluate standard FUNPROC #NAME1 with actuals
COMMASETY call formal type TYPE to give CONDITION and RESULTETY.

H016 FRAMESETY the ONESt frame from ONESt1 table FRAIM link to ONESETY Frame FRAMESETY1 from NAME in ONESt2 frame take
DNAME and present new STACK:
where ONES1 is ONESt2,
where FRAIM contains loc NAME refers the ONESt3th DNAME ht end,
where STACK is FRAMESETY the ONESt frame from ONES1 table FRAIM
link to ONESETY frame.

\* "in ONESt2" is changed to "in ONESt4 frame in ONESt2".
H017 TABLES and STACK through path of FUNPROC NAME find subroutine NAME1 in ONES table with ONES1 as its local table. TABLES and STACK looking for memory location of subroutine NAME in ONES2 frame from ONES table. TABLES involve loc NAME be FUNPROC of PARNUMETY type. TABLES and ONES1 as its local table end exactly in ONES table.

where NAME1 is NAME:

TABLES and STACK looking for memory location of subroutine NAME in ONES2 frame from ONES3 table. TABLES involve loc NAME be formal parameter of FUNPROC of TYPE end exactly in ONES3 table.

STACK from NAME in ONES2 frame take NAME2 and present new STACK1.

TABLES and STACK1 through path of FUNPROC NAME2 find subroutine NAME1 in ONES table with ONES1 as its local table in ONES2 frame.

H018 TABLES and STACK from the trace of FUNPROC DNAME give

#NAME:

standard subroutine DNAME is exactly the same as #NAME.

TABLES and STACK looking for memory location of subroutine DNAME in ONES frame from ONES1 table.

TABLES involve loc DNAME be formal parameter of FUNPROC of TYPE end exactly in ONES1 table.

STACK from DNAME in ONES frame take DNAME1 and present new STACK1.

TABLES and STACK1 from the trace of FUNPROC DNAME1 give

#NAME.

H019 standard subroutine #NAME is exactly the same as #NAME1:

where NAME1 is NAME.

H020 TABLES with STMTS and CONDITION evaluate standard FUNPROC SUBCALL call FORMALTYPE to give CONDITION1 and RESULTETY:

where FUNPROC is function;

TABLES with STMTS and CONDITION the SUBCALL standard function FORMALTYPE find CONDITION1 and

RESULTETY.

where RESULTETY is EMPTY;

where FUNPROC is procedure;

TABLES with STMTS and CONDITION the SUBCALL standard procedure FORMALTYPE find CONDITION1.

H021 TABLES with STMTS and INFILE OUTFILE with STACK and HEAPETY the #NAME with actuals COMMASETY standard function FORMALTYPE find CONDITION and KONSTANT:

TABLES with STMTS and INFILE OUTFILE with STACK and HEAPETY obtain the value of parameterised standard function NAME with actual COMMASETY to be KONSTANT.

where COMMASETY is EMPTY;

INFILE execute standard function end of line or end of file NAME to give KONSTANT.

where CONDITION is INFILE OUTFILE with STACK and HEAPETY.
H022 TABLES with STMTS and CONDITION obtain the value of 
parameterised standard function NAME with FORMALTYPETY
actual comma EXP ammaco to be KONSTANT ;
TABLES with STMTS and CONDITION perform expression EXP
to give CONDITION and eventually prepare
KONSTANT1 ,@ex41
TABLES and FORMALTYPETY due to standard name NAME and
result of its actual parameter KONSTANT1 are
compatible ,@ty71
TABLES the standard function NAME on constant KONSTANT1
give the correct answer KONSTANT,.@ru23-26

H023 TABLES the standard function ORD on constant konst VALUE
tsnok give the correct answer konst VALUE1 tsnok : 
TABLES obtain the ordinal value VALUE to be INT.@op101

H024 TABLES the standard function SUCC on constant konst VALUE
tsnok give the correct answer konst VALUE1 tsnok :
the integer successor of VALUE due to integer from
uminus MAXINT to uplus MAXINT should be VALUE1 !@op104
the character successor of VALUE due to KARSET should be
VALUE1 !@op105
TABLES the successor of scalar identifier VALUE must be
VALUE1.@op106

H025 TABLES the standard function PRED on constant KONSTANT
give the correct answer KONSTANT1 : 
TABLES the standard function SUCC on constant KONSTANT1
give the correct answer KONSTANT..@ru24

H026 TABLES the standard function CHR on konst INT tsnok give
the correct answer konst KCHAR tsnok : 
find the order of character KCHAR in KARSET to be
INT.@op74

H027 TABLES with STMTS and CONDITION the SUBCALL standard
procedure FORMALTYPETY find CONDITION1 : 
TABLES with STMTS and CONDITION execute input
FORMALTYPETY procedure SUBCALL to give
CONDITION1 !@ru23,30
TABLES with STMTS and CONDITION execute output
FORMALTYPETY procedure SUBCALL to give
CONDITION1 !@ru31,35
TABLES with STMTS and CONDITION execute new or dispose
procedure SUBCALL to give CONDITION1.@ru36

H028 TABLES with STMTS and CONDITION execute input procedure
#READ with actuals COMMA COMMASETY to give CONDITION1 : 
TABLES with STMTS and CONDITION read single COMMA to
give CONDITION2 ,@ru29
TABLES with STMTS and CONDITION2 execute input procedure
#READ with actuals COMMASETY to give CONDITION1 i@ru28
where COMMASETY is EMPTY ,
TABLES with STMTS and CONDITION read single COMMA to
give CONDITION1.@ru29
H029 TABLES with STMTS and CONDITION read single comma VARIABLE
ammoc to give INFILE OUTFILE with STACK and HEAPETY:
TABLES with STMTS and CONDITION execute variable
VARIABLE to give INFILE1 OUTFILE with STACK1 and
HEAPETY1 and produce TYPADRES @ex38
INFILE1 with STACK1 and HEAPETY1 input the address of
TYPADRES to make INFILE OUTFILE with STACK and
HEAPETY.@fi15.

H030 TABLES with STMTS and INFILE OUTFILE with STACK and
HEAPETY execute input FORMALTYPETY procedure #READLN with
actuals COMMASETY to give INFILE1 OUTFILE1 with STACK1 and
HEAPETY:
where FORMALTYPETY is EMPTV,
TABLES with STMTS and INFILE OUTFILE with STACK and
HEAPETY execute input procedure #READ with actuals
COMMASETY to give INFILE2 OUTFILE1 with STACK1 and
HEAPETY1.@ru28
INFILE2 execute input newline procedure to give
INFILE1 @fi29
where COMMASETY is EMPTV,
INFILE execute input newline procedure to give
INFILE1.@fi29
where OUTFILE1 with STACK1 and HEAPETY1 is OUTFILE with
STACK and HEAPETY.

H031 TABLES with STMTS and CONDITION execute output
FORMALTYPETY procedure #WRITE with actuals COMMA COMMASETY
to give CONDITION1:
TABLES with STMTS and CONDITION write single COMMA to
give CONDITION2 and FORMALTYPETY.@ru32
TABLES with STMTS and CONDITION2 execute output
procedure #WRITE with actuals COMMASETY to give
CONDITION1.@ru31
where COMMASETY is EMPTV,
TABLES with STMTS and CONDITION write single COMMA to
give CONDITION1.@ru32

H032 TABLES with STMTS and CONDITION write single comma EXP
ammoc to give INFILE OUTFILE with STACK and HEAPETY
FORMALTYPETY:
TABLES with STMTS and CONDITION execute expression EXP
to give INFILE OUTFILE1 with STACK and HEAPETY and
produce RSIDE @ex15
TABLES with FORMALTYPETY write procedure have suitable
type due to RSIDE @ty74
STACK and HEAPETY from RSIDE take the string or constant
CONST.@ru33
OUTFILE1 write in output file either an string or
constant value CONST to have OUTFILE.@fi31

H033 STACK and HEAPETY from RSIDE take the string or constant
CONST:
STACK and HEAPETY from address RSIDE produce either
string or constant CONST @ru34
where CONST is RSIDE.
H034 STACK and HEAPETY from address ADDRESS of TYPEDEF needs ONES boxes produce either string or constant CONST:
   where ONES is one,
   STACK and HEAPETY the ADDRESS contains the amount of
   CONST 12op16
   where ONES contains one one,
   STACK and HEAPETY from ADDRESS access to a group of ONES
   characters KCHARS 12op70
   where CONST is string KCHARS end.

H035 TABLES with STMTS and INFILE OUTFILE with STACK and
   HEAPETY execute output procedure #WRITELN with actuals
   COMMASETY to give INFILE1 OUTFILE1 with STACK1 and
   HEAPETY1:
   OUTFILE execute output newline procedure to give
   OUTFILE2 ;@fi49
   TABLES with STMTS and INFILE OUTFILE2 with STACK and
   HEAPETY execute output procedure #WRITE with actuals
   COMMASETY to give INFILE1 OUTFILE1 with STACK1 and
   HEAPETY1 ;@ru31
   where COMMASETY is EMPTY,
   OUTFILE execute output newline procedure to give
   OUTFILE1 ;@fi49
   where INFILE1 with STACK1 and HEAPETY1 is INFILE with
   STACK and HEAPETY.

H036 TABLES with STMTS and CONDITION execute new or dispose
   procedure #NAME with actuals COMMA COMMASETY to give
   CONDITION1:
   TABLES with STMTS and CONDITION evaluate the new or
   dispose procedure NAME with actual COMMA to give
   CONDITION2 ;@ru37,39
   TABLES with STMTS and CONDITION2 execute new or dispose
   procedure #NAME with actuals COMMASETY to give
   CONDITION1 ;@ru36
   where COMMASETY is EMPTY,
   TABLES with STMTS and CONDITION evaluate the new or
   dispose procedure NAME with actual COMMA to give
   CONDITION1.@ru37,39

H037 TABLES with STMTS and CONDITION evaluate the new or
   dispose procedure NEW with actual comma VARIABLE @moc to
   give INFILE OUTFILE with STACK and HEAP:
   TABLES with STMTS and CONDITION execute variable
   VARIABLE to give INFILE OUTFILE with STACK1 and HEAPETY
   and produce ADDRESS of TYPEDEF needs ONES boxes @ex38
   the last heap number in HEAPETY would be a possibly
   empty ONESETY @ru38
   there are ONES boxes of NBOXES present box ONESETY one
   last ONES @af7
   STACK1 and HEAPETY put the contents of heap ONESETY
   oneth box in ADDRESS to have STACK and HEAPETY1 @op7
   where HEAP is HEAPETY1 NBOXES.

H038 the last heap number in NBOXESETY would be a possibly
   empty ONESETY:
   ONESETY could be the top box number of NBOXESETY @af23
   where ONESETY NBOXESETY is EMPTY.
H039 TABLES with STMTS and CONDITION evaluate the new or
dispose procedure DISPOSE with actual comma VARIABLE ammoc
to give INFILE OUTFILE with STACK and HEAP:
TABLES with STMTS and CONDITION execute variable
VARIABLE to give INFILE OUTFILE with STACK1 and HEAP1
and produce ADDRESS of TYPE ;exp36
STACK1 and HEAP1 put the contents of undefined in
ADDRESS to have STACK and HEAP.aop7
Definitions related to the type checking

H001 TABLES the SCCHARTYPE is defined by the form of the constant konst KCHAR tsnok:
  .true.

H002 TABLES the SCINTTYPE is defined by the form of constant konst uplus UINT tsnok:
   true.

H003 TABLES the REALTYPE is defined by the form of constant konst uplus UREAL tsnok:
   true.

H004 TABLES the pointer defined by IDEN needs one boxes is defined by the form of constant konst nil tsnok:
   true.

H005 TABLESETY the ONESth LOCSETY slink to ONESETY table TABLESETY1 the scalar defined by BNAMEs needs one boxes is defined by the form of constant konst NAME1 in ONESt table tsnok:
   where ONEs is ONEs1,
   NAME in BNAMEs must be an scalar item of LOCSETY ade23

H006 TABLES the array defined by one dimensional left subrange defined by integer from uplus one to uplus ONEs needs one boxes right of SCCHARTYPE needs ONEs1 boxes is defined by the form of constant string KCHARS end:
   there are one ONEs characters in a string KCHARS 3ty7
   where ONEs1 is one ONEs.

H007 there are one ONESETY character in string KCHAR KCHARSETY:
   there are ONESETY characters in a string KCHARSETY 3ty7
   where ONESETY KCHARSETY is EMPTY.

H008 scalar defined by SCALAR needs one boxes of scalar except real:
   where SCALAR differs from REALRANGE.

H009 based on konst NAME in ONEs table tsnok and konst NAME1 in ONEs1 table tsnok the type subrange defined by BNAMEs needs one boxes is part of the type scalar defined by BNAMEs1 needs one boxes:
   where BNAMEs1 contains left NAME right BNAMESETY left NAME1 right,
   where BNAMEs is left NAME BNAMESETY left NAME1 right.

H010 based on konst KCHAR tsnok and konst KCHAR1 tsnok the type subrange defined by KCHARS needs one boxes is a part of the type scalar defined by KARSET needs one boxes:
   where KARSET contains KCHAR KCHARSETY KCHAR1,
   where KCHARS is KCHAR KCHARSETY KCHAR1.
H011 based on konst INT tsnok and konst INT1 tsnok the type subrange defined by integer from INT2 to INT3 needs one boxes is a part of the type SCINTTTYPE:
the integer INT less than integer INT1 \( \theta \)op83
where INT2 is INT,
where INT3 is INT1.

H012 TABLESETY the ONESth LOCS slink to ONESETY table
TABLESETY1 the type identifier NAMETY in ONES1 table
introduce TYPEDEF needs UINT boxes INTABETY:
where ONES is ONES1,
where LOCS contains loc NAMETY be type of TYPEDEF needs
UINT box=8s end,
INTABETY will be defined by ONES \{deo14
where ONES is ONES1,
whenever the NAMETY be not the empty string EMPTY, \{apr10
where LOCS contains loc NAMETY be type of IDEN end,
TABLES the type identifier IDEN introduce TYPEDEF needs
UINT boxes.@ty12

H013 TABLES from the type TYPE get type definition TYPEDEF
needs UINT boxes:
TABLES the type identifier TYPE introduce TYPEDEF needs
UINT boxes ipty12
where TYPEDEF needs UINT boxes is TYPE.

H014 TABLES the type of procedure formal parameter or
subroutine is void:
true.

H015 TABLES the type of SHAPE formal parameter or subroutine is
IDEN:
\( \_ \),
where SHAPE is value,
TABLES old type identifier of IDEN \{deo31
\( \_ \),
where SHAPE is variable,
TABLES old type identifier of IDEN \{deo31
\( \_ \),
where SHAPE is function,
TABLES old type identifier of IDEN \{deo31
TABLES the type identifier IDEN introduce TYPEDEF needs
one boxes, \{aty12
TYPEDEF of function only could be scalar subrange or
pointer.\aty16

H016 TYPEDEF of function only could be scalar subrange or
pointer:
where TYPEDEF is SCRG defined by SCALAR;
where TYPEDEF is pointer defined by IDEN.

H017 TABLES the type TYPE either is the wider form of TYPE1 or
the same:
TABLES the implicit TYPE and TYPE1 give the explicit
TYPE2 and TYPE3, \{aty18
TABLES the explicit TYPE either is the wider form of
explicit TYPE1 or the same.\aty19
H018 TABLES the implicit TYPE and TYPE1 give the explicit TYPE2 and TYPE3:
  TABLES from the type TYPE get type definition
  TYPE2 @ty13  
  TABLES from the type TYPE1 get type definition
  TYPE3 @ty13

H019 TABLES the explicit TYPE either is the wider form of explicit TYPE1 or the same:
  TABLES the TYPE and TYPE1 are identical @ty20
  the first real TYPE and the second integer TYPE1 @ty42
  TYPE be a subrange of TYPE1 or vice versa @ty43

H020 TABLES the TYPE and TYPE1 are identical:
  TABLES the implicit TYPE and TYPE1 give the explicit TYPE2 and TYPE3 @ty18
  TABLES the explicit TYPE2 and TYPE3 are identical @ty21

H021 TABLES the explicit TYPE and TYPE1 are identical:
  the identity of scalars or subranges TYPE and
  TYPE1 @ty22
  TABLES the structured TYPE and TYPE1 are alike in all details @ty23, 25, 40
  TABLES the sameness of pointer type TYPE and TYPE1 @ty41

H022 the identity of scalars or subranges SCRG defined by
SCALAR needs one boxes and SCRG1 defined by SCALAR1 needs
one boxes:
  where SCALAR is SCALAR1.

H023 TABLES the structured array defined by ONES dimensional
INDEXES of TYPE needs ONES1 boxes and array defined by
ONES2 dimensional INDEXES1 of TYPE1 needs ONES3 boxes are
alike in all details:
  where ONES is ONES2
  TABLES the cluster of INDEXES and INDEXES1 are all the
  same @ty24
  TABLES the TYPE and TYPE1 are identical @ty20
  where ONES1 is ONES3.

H024 TABLES the cluster of left TYPE right INDEXESETY and left
TYPE1 right INDEXESEKEY1 are all the same:
  TABLES from the type TYPE get type definition SCRG
defined by SCALAR needs one boxes @ty13
  TABLES from the type TYPE1 get type definition SCRG1
defined by SCALAR needs one boxes @ty13
  TABLES the cluster of INDEXESETY and INDEXESEKEY1 are all
  the same @ty24
  where INDEXESETY INDEXESEKEY1 is EMPTY
  TABLES from the type TYPE get type definition SCRG
defined by SCALAR needs one boxes @ty13
  TABLES from the type TYPE1 get type definition SCRG1
defined by SCALAR needs one boxes @ty13
H025 TABLES the structured record defined by rekord FIELDLIST
end need UINT boxes and record defined by rekord
FIELDLIST1 end needs UINT1 boxes are alike in all
details:
TABLES the possibly empty field lists FIELDLIST and
FIELDLIST1 are all the same i@ty26
where UINT is UINT1.

H026 TABLES the possibly empty field lists FIELDLIST and
FIELDLIST1 are all the same:
TABLES the unity of field lists FIELDLIST and
FIELDLIST1 i@ty27
the empty field list FIELDLIST and FIELDLIST1 are
unique.@ty39

H027 TABLES the unity of field lists FIELDLIST and FIELDLIST1:
TABLES the uniqueness of fixed part FIELDLIST and
FIELDLIST1 i@ty28
TABLES the uniqueness of fixed and variant part
FIELDLIST and FIELDLIST1 i@ty30
TABLES the uniqueness of variant part FIELDLIST and
FIELDLIST1.@ty31

H028 TABLES the uniqueness of fixed part SELECTION
SELECTIONSETY and SELECTION SELECTIONSETY1:
TABLES the pair SELECTION and SELECTION1 are
unique i@ty29
TABLES the uniqueness of fixed part SELECTIONSETY and
SELECTIONSETY1 i@ty28
where SELECTIONSETY SELECTIONSETY1 is EMPTY,
TABLES the pair SELECTION and SELECTION1 are
unique.@ty29

H029 TABLES the pair selection NAME of TYPE with OFFSET TAGETY
end sel and selection NAME1 of TYPE1 with OFFSET1 TAGETY
end sel are unique:
where NAME is NAME1,
TABLES the TYPE and TYPE1 are identical i@ty20
where OFFSET is OFFSET1,
where TAGETY is TAGETY1.

H030 TABLES the uniqueness of fixed and variant part
SELECTIONS VARPART and SELECTIONS1 VARPART1:
TABLES the uniqueness of fixed part SELECTIONS and
SELECTIONS1 i@ty28
TABLES the uniqueness of variant part VARPART and
VARPART1.@ty31

H031 TABLES the uniqueness of variant part VARPART and
VARPART1:
TABLES the identity of nonempty variants VARPART and
VARPART1 i@ty32
the identity of empty variants VARPART and
VARPART1.@ty38
H032 TABLES the identity of nonempty variants case TAGFIELD of VARIANTS and case TAGFIELD1 of VARIANTS1:
TABLES the sameness of tag field TAGFIELD and TAGFIELD1 @ty33
TABLES the sameness of variants VARIANTS and VARIANTS1.@ty33

H033 TABLES the sameness of tag field SELECTIONETY EXTRA and SELECTIONETY1 EXTRAV1:
TABLES the possibly empty field lists SELECTIONETY and SELECTIONETY1 are all the same @ty26
TABLES the identity of extras EXTRA and EXTRAV1.@ty34

H034 TABLES the identity of extras extra TYPE with OFFSET endext and extra TYPE1 with OFFSET1 endext:
TABLES the TYPE and TYPE1 are identical @ty20
where OFFSET is OFFSET1.

H035 TABLES the sameness of variants VARIANT VARIANSETY and VARIANT1 VARIANSETY1:
TABLES the single variant VARIANT and single variant VARIANT1 are the same @ty36
TABLES the sameness of variants VARIANSETY and VARIANSETY1 @ty35
where VARIANSETY VARIANSETY1 is EMPTY,
TABLES the single variant VARIANT and single variant VARIANT1 are the same.@ty36

H036 TABLES the single variant variant label KONSTANTS endlab FIELDDLIST endvar and single variant variant label KONSTANTS1 endlab FIELDDLIST1 endvar are the same:
each of constant KONSTANTS is a member of KONSTANTS1 @ty37
each of constant KONSTANTS1 is a member of KONSTANTS @ty37
TABLES the possibly empty field lists FIELDDLIST and FIELDDLIST1 are all the same.@ty26

H037 each of constant KONSTANT KONSTANSETY is a member of KONSTANTS:
where KONSTANTS contains KONSTANT,
each of constant KONSTANSETY is a member of KONSTANTS @ty37
where KONSTANSETY is EMPTY,
where KONSTANTS contains KONSTANT.

H038 the identity of empty variants and : true.

H039 the empty field list and empty field list are unique : true.

H040 TABLES the structured set defined by elements of TYPE needs one boxes and set defined by elements of TYPE1 needs one boxes are alike in all details:
TABLES the TYPE and TYPE1 are identical.@ty20
H041 TABLES the sameness of pointer types pointer defined by IDEN needs one boxes and pointer defined by IDEN1 needs one boxes:
where IDEN is IDEN1; TABLES the IDEN and IDEN1 are identical.

H042 the first real REALTYPE and the second integer INTTYPE: true.

H043 TYPE be subrange of TYPE1 or vice versa:
both integer TYPE and TYPE1: 
nonnumber scalar TYPE and TYPE1 be subrange of each other.

H044 both integer INTTYPE and INTTYPE1: true.

H045 nonnumber scalar SCRG defined by SCALAR needs one boxes and SCRG1 defined by SCALAR1 needs one boxes be subrange of each other:
where SCALAR contains SCALAR1; where SCALAR1 contains SCALAR.

H046 TABLES operand TYPE multiply operation MULOP operand TYPE1 make TYPE2:
TABLES the implicit TYPE and TYPE1 give the explicit TYPE3 and TYPE4:
TABLES explicit TYPE3 multiply operation MULOP explicit TYPE4 design TYPE2.

H047 TABLES explicit operand TYPE multiply operation types:
explicit TYPE1 design TYPE2:
both integer TYPE and TYPE1:
where TYPE2 is SCINTTYPE:
the first real TYPE and the second integer TYPE1:
where TYPE2 is REALTYPE:
the first real TYPE1 and the second integer TYPE:
where TYPE2 is REALTYPE:
both real TYPE and TYPE1:
where TYPE2 is REALTYPE:
TABLES a set type TYPE with set type TYPE1 make a set type TYPE2.

H048 both real REALTYPE and REALTYPE: true.

H049 TABLES a set type set defined by elements of TYPE needs one boxes with set type set defined by elements of TYPE1 needs one boxes make a set type set defined by elements of TYPE2 needs one boxes:
TABLES the implicit TYPE and TYPE1 give the explicit TYPE3 and TYPE4:
the identity of scalars or subranges TYPE3 and TYPE4:
where TYPE2 is TYPE3.

H050 TABLES explicit operand TYPE multiply operation realdiv explicit TYPE1 design TYPE2:
both integer or real TYPE and TYPE1:
where TYPE2 is REALTYPE.
H051 both integer or real \( \text{TYPE} \) and \( \text{TYPE1} \):
both integer \( \text{TYPE} \) and \( \text{TYPE1} \); 
the first real \( \text{TYPE} \) and the second integer \( \text{TYPE1} \); 
the first real \( \text{TYPE} \) and the second integer \( \text{TYPE1} \); 
both real \( \text{TYPE} \) and \( \text{TYPE1} \).

H052 TABLES explicit operand \( \text{TYPE} \) multiply operation intdiv
explicit \( \text{TYPE1} \) design \( \text{TYPE2} \):
both integer \( \text{TYPE} \) and \( \text{TYPE1} \); 
where \( \text{TYPE2} \) is \( \text{SCINTTYPE} \).

H053 TABLES explicit operand \( \text{TYPE} \) multiply operation mod
explicit \( \text{TYPE1} \) design \( \text{TYPE2} \):
both integer \( \text{TYPE} \) and \( \text{TYPE1} \); 
where \( \text{TYPE2} \) is \( \text{SCINTTYPE} \).

H054 TABLES explicit operand \( \text{TYPE} \) multiply operation and
explicit \( \text{TYPE1} \) design \( \text{TYPE2} \):
a boolean \( \text{TYPE} \) and a boolean \( \text{TYPE1} \) produce a boolean \( \text{TYPE2} \).

H055 a boolean \( \text{BOOLTYPE} \) and a boolean \( \text{BOOLTYPE} \) produce a boolean \( \text{SCBOOLTYPE} \):
true.

H056 TABLES operand \( \text{TYPE} \) adding operation \( \text{ADDOP} \) operand \( \text{TYPE1} \)
make \( \text{TYPE2} \):
TABLES implicit \( \text{TYPE} \) and \( \text{TYPE1} \) give the explicit \( \text{TYPE3} \) and \( \text{TYPE4} \); 
TABLES explicit \( \text{TYPE3} \) adding operation \( \text{ADDOP} \) explicit \( \text{TYPE4} \) make \( \text{TYPE2} \).

H057 TABLES explicit operand \( \text{TYPE} \) adding operation \( \text{ADDOP} \)
explicit \( \text{TYPE1} \) design \( \text{TYPE2} \):
where \( \text{ADDOP} \) is uplus,
TABLES explicit operand \( \text{TYPE} \) multiply operation times
explicit \( \text{TYPE1} \) design \( \text{TYPE2} \); 
where \( \text{ADDOP} \) is minus,
TABLES explicit operand \( \text{TYPE} \) multiply operation time
explicit \( \text{TYPE1} \) design \( \text{TYPE2} \); 
where \( \text{ADDOP} \) is or,
TABLES explicit operand \( \text{TYPE} \) multiply operation and
explicit \( \text{TYPE1} \) make \( \text{TYPE2} \).

H058 TABLES operand \( \text{TYPE} \) relational operation \( \text{RELOP} \) operand \( \text{TYPE1} \)
are compatible:
TABLES the implicit \( \text{TYPE} \) and \( \text{TYPE1} \) give the explicit \( \text{TYPE3} \) and \( \text{TYPE4} \); 
TABLES explicit \( \text{TYPE} \) relational operation \( \text{RELOP} \) explicit \( \text{TYPE1} \) are compatible.

H059 TABLES explicit \( \text{TYPE} \) relational operation in explicit set
defined by elements of \( \text{TYPE1} \) needs one boxes are
compatible:
TABLES from the type \( \text{TYPE1} \) get type definition \( \text{TYPE2} \);
the identity of scalars or subranges \( \text{TYPE2} \) and \( \text{TYPE1} \).
H060 TABLES explicit TYPE relational operation EQNOTEQ explicit TYPE1 are compatible:
   TABLES the sameness of pointer types TYPE and TYPE1 @ty41
   TABLES explicit TYPE relational operation less or equal to explicit TYPE1 are compatible.@ty61

H061 TABLES explicit TYPE relational operation LEORGE explicit TYPE1 are compatible:
   TABLES both set types TYPE and TYPE1 @ty62
   TABLES explicit TYPE relational operation less than explicit TYPE1 are compatible.@ty63

H062 TABLES both set types set defined by elements of TYPE needs one boxes and set defined by elements of TYPE1 needs one boxes:
   TABLES the TYPE and TYPE1 are identical.@ty20

H063 TABLES explicit TYPE relational operation LORG explicit TYPE1 are compatible:
   TABLES both string TYPE and TYPE1 @ty64
   both integer or real TYPE and TYPE1 @ty51
   both non number scalar TYPE and TYPE1.@ty65

H064 TABLES both string array defined by one dimensional left TYPE right of CHARTYPE needs ONES boxes and array defined by one dimensional left TYPE1 right of CHARTYPE1 needs ONES1 boxes:
   TABLES from the type TYPE get type definition SCRG defined by integer from uplus one to uplus one ONES2 needs one boxes @ty13
   TABLES from the type TYPE1 get type definition SCRG1 defined by integer from uplus one to uplus one ONES3 needs one boxes @ty13
   where ONES3 is ONES2 ,
   where CHARTYPE is CHARTYPE1 ,
   where ONES is ONES1 .

H065 both non number scalar SCRG defined by SCALAR needs one boxes SCRG1 defined by SCALAR1 needs one boxes:
   where SCALAR is SCALAR1 .

H066 TABLES the subroutine types TYPE and TYPE1 both are either void or usual:
   where TYPE TYPE1 is void void ,
   TABLES the type TYPE either is the wider form of TYPE1 or the same.@ty17

H067 the standard function NAME is of type TYPE:
   where NAME is ORD ,
   where TYPE is INTTYPE ,
   NAME either is successor or predecessor ,@st76
   where TYPE is SCRG defined by SCALAR needs one boxes ,
   where TYPE differs from REALTYPE ,
   where NAME is CHR ,
   where TYPE is CHARTYPE ,
   NAME could be either eoln or eof ,@st79
   where TYPE is BOOLTYPE .
H068 there is an special TYPE for read subroutine parameter:
where TYPE is INTTYPE;
where TYPE is REALTYPE.

H069 TABLES there is an special TYPE for print subroutine:
where TYPE is CHARTYPE;
where TYPE is INTTYPE;
where TYPE is REALTYPE;
TABLES the TYPE should be of mode of string.@ty70

H070 TABLES the array defined by one dimensional left TYPE
right of TYPE1 needs one ONES boxes should be of mode of
string:
TABLES from the type TYPE get type definition SCRG
defined by integer from uplus one to uplus one ONES1
needs one boxes ;@ty13
TABLES from the type TYPE1 get type definition
CHARTYPE.@ty13

H071 TABLES and FORMALTYPE latter due to standard name NAME and the
result of actual parameter KONSTANT are compatible:
TABLES and nonempty FORMALTYPE latter with standard name NAME
with actual constant KONSTANT are compatible ;@ty72
where FORMALTYPE is EMPTY.

H072 TABLES and nonempty formal type IDEN with standard name
NAME with actual constant KONSTANT are compatible:
TABLES the type identifier IDEN introduce TYPE ;@ty12
TABLES the TYPE1 is defined by the form of constant
KONSTANT ;@ty1-6
formal TYPE of actual standard NAME is compatible with
its actual parameter TYPE1.@ty73

H073 formal TYPE of actual standard NAME is compatible with its
actual parameter SCRG defined by SCALAR needs one boxes:
where NAME is ORD,
where SCALAR differs from REALRANGE
where NAME is CHR,
where SCALAR is INTERVAL
NAME either is successor or predecessor ;@st77
where SCALAR differs from REALRANGE,
where TYPE is SCRG defined by SCALAR needs one boxes.

H074 TABLES with FORMALTYPE write procedure have suitable
type due to RSIDE:
where FORMALTYPE is formal type void,
TABLES check the type of the actual parameter of write
by RSIDE ;@ty75
where FORMALTYPE is EMPTY.

H075 TABLES check the type of actual parameter of write by
RSIDE:
TABLES test the RSIDE for write parameter ;@ty76
TABLES the TYPE is defined by the form of constant
RSIDE ;@ty1-6
TABLES there is an special TYPE for print
subroutine.@69

H076 TABLES test the ADDRESS of TYPE for write parameter:
TABLES there is an special TYPE for print
subroutine.@69
Definition of constants

H001 plan string char CHAR KCHARS end to be string :
  CHAR symbol @sy3-94
  where KARSET contains char CHAR ,
  design sequence of characters KCHARS @co2
 |

H002 design sequence of characters char CHAR KCHARSETY :
  CHAR symbol ,
  where KARSET contains char CHAR ,
  design sequence of characters KCHARSETY @co2
  where KCHARSETY is EMPTY ,
  CHAR symbol ,@sy3-94
  where KARSET contains char CHAR .
 |

H003 plan konst char CHAR tsnok to be character of KCHARS :
  CHAR symbol ,@sy3-94
  where KCHARS contains char CHAR ,
 |

H004 plan konst UDP UNNUMBER tsnok to be number might in INTERVALETY :
  UDP operation ,sy97-98
  UDP unsigned number UTNUMBER produce UNNUMBER INTERVALITY @co5
  uplus unsigned number UTNUMBER produce UNNUMBER INTERVALITY @co5
 |

H005 UDP unsigned number UTNUMBER produce UNNUMBER INTERVALITY :
  syntax of unsigned integer UTNUMBER is correct @sy99
  UDP unsigned decimal integer number UTNUMBER produce unary integer UNNUMBER INTERVALITY @co6
  syntax of unsigned real UTNUMBER is correct @sy100-101
  UDP unsigned decimal real number UTNUMBER produce unary real UNNUMBER @co7
 |

H006 UDP unsigned decimal integer number DIGITS produce unary integer UINT INTERVALETY :
  remove left nulls from DIGITS to produce @co7
  where UINT is zero :
  remove left nulls from DIGITS to produce DIGITS1 @co7
  DIGITS1 unsigned integer is the same as UINT INTERVALETY @co8
remove left nulls from \text{DIGIT} \text{DIGITSETY} to produce \\
\text{DIGITSETY1} : \\
\text{where DIGIT is 0 ,} \\
\text{remove left nulls from \text{DIGITSETY} to produce} \\
\text{DIGITSETY1} \; \@\text{co7} \\
\text{where \text{DIGITSETY DIGITSETY1 is EMPTY},} \\
\text{where DIGIT is 0 ;} \\
\text{where DIGIT differs from 0 ,} \\
\text{where \text{DIGITSETY1 is DIGIT DIGITSETY.} } \\

\text{DIGITSETY DIGIT unsigned integer is the same as \text{UINT.} } \\
\text{INTERVALEY :} \\
\text{decimal DIGIT is equivalent to \text{UINT} \; \@\text{co9-17}} \\
\text{DIGITSETY unsigned integer is the same as \text{UINT1}} \\
\text{INTERVALEY \; \@\text{co8}} \\
\text{decimal DIGIT is equivalent to \text{UINT2} \; \@\text{co9-17}} \\
\text{the integer uplus \text{UINT3 equal to integer uplus \text{UINT1}} times integer uplus \text{RADIX INTERVALEY \; \@\text{op27}} \\
\text{the integer uplus \text{UINT equal to integer uplus \text{UINT3 plus integer uplus \text{UINT2 INTERVALEY \; \@\text{op27}}}} \\
\text{where \text{DIGITSETY is EMPTY.} } \\

\text{decimal 1 is equivalent to one : true.} \\
\text{decimal 2 is equivalent to one one : true.} \\
\text{decimal 3 is equivalent to one one one : true.} \\
\text{decimal 4 is equivalent to one one one one : true.} \\
\text{decimal 5 is equivalent to one one one one one : true.} \\
\text{decimal 6 is equivalent to one one one one one : true.} \\
\text{decimal 7 is equivalent to one one one one one one : true.} \\
\text{decimal 8 is equivalent to one one one one one one one : true.} \\
\text{decimal 9 is equivalent to one one one one one one one one : } \\
\text{true.} \\

\text{UOP unsigned real number UTREAL produce UREAL1 :} \\
\text{the two forms unsigned real UTREAL produce \text{UINT1 e uplus ONES \; \@\text{co19,24}} } \\
\text{the exponent UOP \text{UINT1 e uplus ONES is positive so change it to null to give UOP UREAL1 \; \@\text{co25}} \\
\text{the two forms unsigned real UTREAL produce} \\
\text{UINT1 e INT \; \@\text{co19,24}} \\
\text{where uplus one is not in INT} \\
\text{the two cases for real UOP \text{UINT1 e INT will give UOP}} \\
\text{UREAL1 \; \@\text{co26,27}}
H019 the two forms unsigned real DIGITS dot DIGITS1 EXPONENT produce UINT e INT:
remove left nulls from DIGITS to produce DIGITS2,
there are ONES decimal DIGITS1, where DIGITS2 DIGITS1 unsigned integer is the same as UINT.
exponent part EXPONENT produce INT1, where the integer INT equal to integer uminus ONES plus integer INT1.
remove left nulls from DIGITS to produce DIGITS2,
there are ONES decimal DIGITS1, where DIGITS2 unsigned integer is the same as UINT.
exponent part EXPONENT produce INT1, where the integer INT equal to integer uplus ONES plus integer INT1.
remove left nulls from DIGITS to produce DIGITS2,
remove left nulls from DIGITS1 to produce DIGITS1, where UINT e INT is zero e uplus zero.

H020 there are one ONESETY decimal DIGIT DIGITSETY:
DIGIT symbol,
there are ONESETY decimal DIGITSETY, where ONESETY DIGITSETY is EMPTY,
DIGIT symbol.

H021 exponent part EXPONENT produce INT:
nonempty exponent part EXPONENT produce INT, where EXPONENT is EMPTY, where INT is uplus zero.

H022 nonempty exponent part e SIGNETY DIGITS produce UOP UINT:
SIGNETY will specially clear UOP, where UOP unsigned decimal integer number DIGITS produce UINT.

H023 SIGNETY will specially clear UOP:
where SIGNETY UOP is uplus, where SIGNETY UOP is uminus, where SIGNETY is EMPTY, where UOP is uplus.

H024 the two forms unsigned real DIGITS e EXPON produce UINT e INT:
where UINT contains one,
uplus unsigned decimal integer number DIGITS produce UINT,
nonempty exponent part EXPON produce INT, where UINT is zero,
uplus unsigned decimal integer number DIGITS produce zero,
where INT is e uplus zero.
The exponent INT uplus one ONESETY is positive so change it to null to give REYAL 1:
the integer INT1 equal to integer INT times integer uplus RADIX youp27
the exponent INT1 uplus ONESETY is positive so change it to negative to give REYAL 10co25
where ONESETY is EMPTY 1.
the integer INT1 equal to integer INT times integer uplus RADIX youp27
the two cases for real INT1 uplus zero will give REYAL 10co26,27

The two cases for real INT uplus UOP zero will give REYAL 1:
the integer INT is in the interval integer from uminus INTEGRAL to uplus INTEGRAL youp31
where REYAL is INT uplus zero.

The two cases for real INT uminus ONES will give REYAL 1:
the integer uminus ONES less than integer uminus DECIMAL youp83
decimal addition INT uminus ONES and integral division give REYAL1 10co28
the two cases for real REYAL1 will give REYAL 10co26,27
the integer uminus ONES greater or equal to integer uminus DECIMAL 10op83
right or wrong integral INT uminus ONES give REYAL 10co29

decimal addition INT uminus ONES and integral division give INT1 uplus INT2 1:
the integer INT2 equal to integer uplus one plus integer uminus ONES 10op27
scalar value or set operation konst INT tsnok intdiv konst uplus RADIX tsnok give konst INT1 tsnok 10op57

Right or wrong integral UOP UINT uminus ONES give REYAL 1:
the integer UOP UINT is in the interval integer from uminus INTEGRAL to uplus INTEGRAL 10op31
where REYAL is UOP UINT uminus ONES 1.
the integer uplus UINT greater than integer uplus INTEGRAL 10op83
decimal addition UOP UINT uminus ONES and integral division give REYAL1 10co28
the two cases for real REYAL1 will give REYAL 10co26,27

The number INT is in the right range 1 true.

The number INT uminus INT1 is in the right range 1:
the integer INT is in the interval integer from uminus MTEGRAL to uplus MAXTEGRAL 10op31
the integer INT1 greater or equal to integer uminus DECIMAL 10op83
Definitions for operations

H001 TABLES with STACK and HEAPETY the TYPEADRES must become RSIDE with new STACK1 and HEAPETY1:
- TABLES with STACK and HEAPETY the TYPEADRES should be assigned by sequence value RSIDE to give new STACK1 and HEAPETY1 @op2
- STACK and HEAPETY; the RSIDE eventually introduces DBOX @op4
- STACK and HEAPETY the TYPEADRES which is simple or pointer assigned by DBOX to give STACK1 and HEAPETY1 @op4
- STACK and HEAPETY the structured type TYPEADRES becomes RSIDE with new STACK1 and HEAPETY1.@op19,24

H002 TABLES with STACK and HEAPETY the ADDRESS of array defined by one dimensional left TYPE right of TYPE1 needs ONES box should be assigned by sequence value string KCHARS and to give new STACK1 and HEAPETY1:
- TABLES from the type TYPE get type definition SCRG defined by integer from uplus one to uplus one ONES1 @aty13
- TABLES from the type TYPE1 get type definition CHARTYPE @aty13
- STACK and HEAPETY from ADDRESS assign each KCHARS in ONES times each of CHARTYPE to have STACK1 and HEAPETY1.@op3

H003 STACK and HEAPETY from STKHP ONESth box assigned each KCHAR KCHARSETY in one ONESETY times each of CHARTYPE to have STACK1 and HEAPETY1:
- STACK and HEAPETY the STKHP ONESth box of CHARTYPE which is simple or pointer is assigned by konst KCHAR tsnok to give STACK2 and HEAPETY2 @op4
- STACK2 and HEAPETY2 from STKHP ONES oneth box assigned each KCHARSETY in ONESETY times each of CHARTYPE to have STACK1 and HEAPETY1 @op3
- where KCHARSETY ONESETY is EMPTY,
- STACK and HEAPETY the STKHP ONESth box of CHARTYPE which is simple or pointer is assigned by konst KCHAR tsnok to give STACK1 and HEAPETY1 @op4

H004 STACK and HEAPETY the ADDRESS of TYPE which is simple or pointer is assigned by DBOX to give STACK1 and HEAPETY1:
- STACK and HEAPETY the real ADDRESS of TYPE becomes DBOX to give STACK1 and HEAPETY1 @op5
- STACK and HEAPETY the integer or character or scalar identifier ADDRESS of TYPE becomes DBOX to give STACK1 and HEAPETY1 @op10
- STACK and HEAPETY put the quantity DBOX in ADDRESS to have STACK1 and HEAPETY1.@op6
H005 STACK and HEAPETY the real ADDRESS of REALTYPE becomes konst NUMBER tsnok to give STACK1 and HEAPETY1:
  where NUMBER is INT,
  STACK and HEAPETY put the quantity konst INT e plus zero tsnok in ADDRESS to have STACK1 and HEAPETY1 \( \oplus \) op6
  .STACK and HEAPETY put the quantity konst NUMBER tsnok in ADDRESS to have STACK1 and HEAPETY1 \( \oplus \) op6

H006 STACK and HEAPETY put the quantity BOX in STKHP ONESth TAGETY box to have STACK1 and HEAPETY1:
  where TAGETY is tag field,
  STACK and HEAPETY put the contents of BOX in STKHP ONESth box to have STACK2 and HEAPETY2 \( \oplus \) op7
  STACK2 and HEAPETY2 put the contents of BOX in STKHP ONES oneth box to have STACK1 and HEAPETY1 \( \oplus \) op7
  where TAGETY is EMPTY,
  STACK and HEAPETY put the contents of BOX in STKHP ONESth box to have STACK1 and HEAPETY1 \( \oplus \) op7

H007 STACK and HEAPETY put the contents of BOX in ADDRESS to have STACK1 and HEAPETY1:
  STACK stack insertion BOX at ADDRESS give STACK1 \( \oplus \) op8
  where HEAPETY1 is HEAPETY
  HEAPETY heap insertion BOX at ADDRESS give
  HEAPETY1 \( \oplus \) op9
  where STACK1 is STACK.

H008 CHARSETY the ONESth BOX ht CHARSETY1 stack insertion BOX1 at stack ONES1th box give STACK:
  where ONES is ONES1,
  where STACK is CHARSETY the ONESth BOX1 ht CHARSETY1.

H009 CHARSETY the ONESth BOX ht CHARSETY1 heap insertion BOX1 at heap ONES1th box give HEAP:
  where ONES is ONES1,
  where HEAP is CHARSETY the ONESth BOX1 ht CHARSETY1.

H010 STACK and HEAPETY the integer or character or scalar identifier ADDRESS of SCRG defined by SCALAR needs one boxes becomes konst VALUE tsnok to give STACK1 and HEAPETY1:
  the value VALUE must be in the range of SCALAR \( \oplus \) op11-13
  STACK and HEAPETY put the quantity konst VALUE tsnok in ADDRESS to have STACK1 and HEAPETY1 \( \oplus \) op6

H011 the value INT must be in the range of INTERVAL:
  the integer INT is in the interval INTERVAL.

H012 the value KCHAR must be in the range of KCHARS:
  where KCHARS contains KCHAR.

H013 the value NAME in ONES table must be in the range of BNAMES:
  where BNAMES contains left NAME right.
H014 STACK and HEAPETY the RSIDE eventually introduces DBOXORSTG:
STACK and HEAPETY the address RSIDE reach to the quantity DBOXORSTG 13op15
where RSIDE is DBOXORSTG.

H015 STACK and HEAPETY the address ADDRESS of TYPE reach to the quantity DBOX:
STACK and HEAPETY the ADDRESS contains the amount DBOX 2op16

H016 STACK and HEAPETY the ADDRESS contains the amount BOX:
STACK the stack destination ADDRESS produce BOX 13op17
HEAPETY the heap destination ADDRESS produce BOX 2op18

H017 STACK the stack destination stack ONESth TAGETY box produce BOX:
where STACK contains the ONESth BOX ht.

H018 HEAP the heap destination heap ONESth TAGETY box produce BOX:
where HEAP contains the ONESth BOX ht.

H019 STACK and HEAPETY the structured type ADDRESS of array defined by ONESt dimensional INDEXES of TYPE needs UINT boxes becomes ADDRESS1 with new STACK1 and HEAPETY1:
STACK and HEAPETY determine a collection of UINT of BOXESETY from ADDRESS1 13op20
STACK and HEAPETY copy BOXESETY at ADDRESS to have new STACK1 and HEAPETY1 2op22

H020 STACK and HEAPETY determine a collection of UINT of BOXESETY from ADDRESS:
STACK and HEAPETY obtain a collection of UINT of BOXESETY from ADDRESS 13op21
where UINT is zero,
where BOXESETY is EMPTY.

H021 STACK and HEAPETY obtain a collection of one ONESETY of BOX BOXESETY from STKHP ONESth box:
STACK and HEAPETY the STKHP ONESth box contains of the amount BOX 13op16
STACK and HEAPETY obtain a collection of ONESETY of BOXESETY from STKHP ONESt box 13op21
where ONESETY BOXESETY is EMPTY,
STACK and HEAPETY the STKHP ONESth box contains of the amount BOX 2op16

H022 STACK and HEAPETY copy BOXESETY at ADDRESS to have new STACK1 and HEAPETY1:
STACK and HEAPETY copy nonempty BOXESETY at ADDRESS to give new STACK1 and HEAPETY1 13op23
where BOXESETY is EMPTY.
H023 STACK and HEAPETY copy nonempty BOX BOXSETY at STKHP ONESth box to give new STACK1 and HEAPETY1 .
STACK and HEAPETY put the quantity BOX in STKHP ONESth box to have STACK2 and HEAPETY2 .
STACK2 and HEAPETY2 copy nonempty BOX BOXSETY at STKHP ONES oneth box to give new STACK1 and HEAPETY1 .

where BOXSETY is EMPTY ,
STACK and HEAPETY put the quantity BOX in STKHP ONESth box to have STACK1 and HEAPETY1 .

H024 STACK and HEAPETY the structured type ADDRESS of record defined by rekord FIELDLIST end needs UINT boxes becomes ADDRESS1 with new STACK1 and HEAPETY1 .
STACK and HEAPETY determine a collection of UINT of BOXSETY from ADDRESS1 .
STACK and HEAPETY copy BOXSETY at ADDRESS to have new STACK1 and HEAPETY1 .

H025 scalar value or set operation DBOX PMT DBOX1 give DBOX :
the result of number operation DBOX PMT DBOX1 is DBOX .
the result of set operation DBOX PMTD DBOX1 is DBOX .

H026 the result of number operation konst NUMBER tsnok PMTD konst NUMBER1 tsnok is konst NUMBER2 tsnok :
the integer NUMBER2 equal to integer NUMBER PMTD integer NUMBER1 integer from uminus MAXINT to uplus MAXINT .
the real NUMBER2 equal to integer NUMBER PMTD real NUMBER1 .
the real NUMBER2 equal to real NUMBER PMTD integer NUMBER1 .
the real NUMBER2 equal to real NUMBER PMTD real NUMBER1 .

H027 the integer INT equal to integer INT1 PMT integer INT2 INTERVALETY :
both non null integer INT1 PMT integer INT2 give integer INT due to INTERVALETY .
at least a null integer INT1 PMT integer INT2 give INT .

H028 both non null integer UOP ONES plus integer UOP1 ONES1 give UOP2 ONES2 due to INTERVALETY :
where UOP UOP1 UOP2 is uplus uplus uplus ,
the sumation of ONES and ONES1 will be UOP2 ONES2 due to INTERVALETY .
where UOP UOP1 is uplus uminus ,
both non null integer UOP ONES minus integer uplus ONES1 give integer UOP2 ONES2 due to uplus .
where UOP UOP1 is uminus uplus ,
both non null integer UOP1 ONES1 minus integer uplus ONES give integer UOP2 ONES2 due to uplus .
where UOP UOP1 UOP2 is uminus uminus uminus ,
the sumation of ONES ana ONES1 will be UOP2 ONES2 due to INTERVALETY .
H029 the summation of ONES and ONESETY one will be UOP ONES1 due to INTERVALETY.
check overflow situation for UOP ONES one with INTERVALETY @op30
the summation of ONES one and ONESETY will be UOP ONES1
due to INTERVALETY @op29
where ONESETY is EMPTY.
check overflow situation for UOP ONES one with INTERVALETY @op30
where ONES1 is ONES one.

H030 check overflow situation for INT with INTERVALETY:
the integer INT is in the interval INTERVALETY @op31
where INTERVALETY is EMPTY.

H031 the integer INT is in the interval integer from INT1 to INT2:
the integer INT1 less or equal to integer INT @op83
the integer INT less or equal to integer INT2 @op83

H032 both non null integer UOP ONES minus UOP1 ONES1 give
integer UOP2 UINT due to INTERVALETY:
where UOP UOP1 is uolus uplus,
the subtraction of ONES minus ONES1 or vice versa make
UOP2 UINT @op33
where UOP UOP1 UOP2 is uplus uminus uplus,
the summation of ONES and ONES1 will be UOP2 UINT due to
INTERVALETY @op29
where UOP UOP1 UOP2 is uminus uminus uplus,
the summation of ONES and ONES1 will be UOP2 UINT due to
INTERVALETY @op29
where UOP UOP1 is uminus uminus,
the subtraction of ONESETY minus ONES or vice versa make
UOP2 UINT @op33

H033 the subtraction of ONES minus ONESETY or vice versa make
INT:
where ONESETY is ONESETY
where INT is uolus zero
where ONESETY is ONES1 ONESETY
where INT is uplus ONESETY
where ONESETY is ONES ONESETY
where INT is uminus ONESETY

H034 both non null integer UOP ONES times integer UOP1 ONES1
give integer UOP2 ONESETY due to INTERVALETY:
the result of sign UOP and UOP1 in multiplication or
division operation is UOP2 @op35
the production of ONES and ONESETY is UOP2 ONESETY due to
INTERVALETY @op36

H035 the result of sign UOP and UOP1 in multiplication or
division operation is UOP2:
where UOP UOP1 UOP2 is uolus uolus uolus
where UOP UOP1 UOP2 is uplus uminus
where UOP UOP1 UOP2 is uminus uplus
where UOP UOP1 UOP2 is uminus uminus uplus.
H036 the production of ONES and ONESETY one is UOP ONES1 due to INTERVALEY:
more production for ONES and the factor counted by ONESETY will give ONES1 due to INTERVALEY @op37
where ONESETY is EMPTY , where ONES1 is ONES.
H037 more production for ONES and the factor ONES1 counted by one ONESETY will give UOP ONES2 due to INTERVALEY:
the sumation of ONES and ONES1 will be UOP ONES3 due to INTERVALEY @op29
more production for ONES3 and the factor ONES1 counted by ONESETY will give UOP ONES2 due to INTERVALEY @op37
where ONESETY is EMPTY , the sumation of ONES and ONES1 will be UOP ONES2 due to INTERVALEY @op29
H038 at least a null integer INT plus integer INT1 give integer INT2:
where INT is UOP zero ,
where INT2 is INT1 ,
where INT1 is UOP zero ,
where INT2 is INT.
H039 at least a null integer INT minus integer INT1 give integer INT2:
where INT1 is UOP zero ,
where INT2 is INT ,
where INT is UOP zero ,
where INT is UOP zero ,
both non null integer INT1 times integer minus one give integer INT2 due to @op34
H040 at least a null integer INT times integer INT1 give uplus zero:
where INT is UOP zero ,
where INT1 is UOP zero.
H041 the integer REYAL equal to integer INT realdiv integer INT1 INTERVALEY:
the real REYAL equal to real INT e uplus zero realdiv real INT1 e uplus zero @op48
H042 the real REYAL equal to integer INT PMTD real REYAL1:
the real REYAL equal to real INT e uplus zero PMTD real REYAL1 @op44,47,48
H043 the real REYAL equal to real REYAL1 PMTD integer INT:
the real REYAL equal to real REYAL1 PMTD real INT e uplus zero @op44,47,48
H044 the real INT e INT1 equal to real INT2 e INT3 PM real INT4 e INT5 :
the integer INT5 greater or equal to integer INT3 ,@op83
the integer INT6 equal to integer INT5 minus integer INT3 ,@op27
the product of INT4 and ten to the INT6 is INT7 due to integer from uminus INTEGRAL to uplus INTEGRAL ,@op45
the integer INT equal to integer INT2 PM integer INT7
integer from uminus INTEGRAL to uplus INTEGRAL ,@op27
where INT1 is INT3 ;
the integer INT3 greater than integer INT5 ,@op83
the integer INT6 equal to integer INT3 minus integer INT5 ,@op27
the product of INT2 and ten to the INT6 is INT7 due to integer from uminus INTEGRAL to uplus INTEGRAL ,@op45
the integer INT equal to integer INT7 PM integer INT4
integer from INTEGRAL to uplus INTEGRAL ,@op27
where INT1 is INT5.

H045 the product of INT and ten to the uplus UINT is INT1 due to INTERVALEY :
the production of INT and ten to non null UINT is INT1 due to INTERVALEY ,@op46
where UINT is zero ,
where INT1 is INT.

H046 the production of INT and ten to non null one ONESETY is INT1 due to INTERVALEY :
the integer INT2 equal to integer INT times integer uplus RADIX INTERVALEY ,@op27
the production of INT2 and ten to non null ONESETY is INT due to INTERVALEY ,@op46
where ONESETY is EMPTY ,
the integer INT1 equal to integer INT times integer uplus RADIX INTERVALEY .@op27

H047 the real INT e INT1 equal to real INT2 e INT3 times INT4 e INT5 :
the integer INT equal to integer INT2 times integer INT4
integer from uminus INTEGRAL to uplus INTEGRAL ,@op27
the integer INT1 equal to integer INT3 plus integer INT5
integer from uminus DECIMAL to uplus DECIMAL ,@op27
H048 the real UOP UINT e INT equal to real UOP1 UINT1 realdiv UOP2 UINT2 e INT2:
the result of sign UOP1 and UOP2 in multiplication or division operation is UOP1 @op35 the integer INT3 equal to integer INT1 minus integer INT2 @op27 the integer INT3 greater than integer UINT1 and ten to the INT3 is UINT3 due to @op45 the result of unsigned integer UINT3 divided by unsigned integer UINT4 @op50 the remainder actions for UINT1 and UINT2 and quotient UINT4 with exponent pat UINT1 equals to unsigned UINT3 e INT with division part UINT4 @op35 the result of sign UOP1 and UOP2 in multiplication or division operation is UOP1 @op35 the integer INT3 equal to integer INT1 minus integer INT2 @op27 the integer INT3 less or equal to integer UINT1 @op83 the real division of unsigned integer UINT1 by unsigned integer UINT2 and quotient zero with exponent part INT3 results unsigned integer UINT e INT.@op49

H049 the real division of unsigned integer UINT by unsigned integer UINT1 and quotient UINT2 with exponent part INT results unsigned integer UINT3 e INT1:
the result of unsigned integer UINT divided by unsigned integer UINT1 is unsigned integer UINT4 @op50 the integer UINT5 equal to integer UINT1 plus integer UINT4 @op27 the remainder actions for UINT1 and UINT2 and quotient UINT5 with exponent part results unsigned UINT3 e INT1 with division part UINT4 @op53 the result of unsigned integer UINT divided by unsigned integer UINT1 is unsigned integer UINT4 @op50 the integer UINT5 equal to integer UINT1 plus integer UINT4 @op27 the integer UINT5 greater than integer UINT1 plus integer UINT4 @op53 by the quotient INTEGER and exponent INT on the base of null or non null quotient make UINT3 e INT1.@op54

H050 the result of unsigned integer UINT divided by unsigned integer ONES is unsigned integer UINT1:
unsigned non null integer division UINT by ONES give UINT @op51 where UINT UINT1 is zero zero.

H051 unsigned non null integer division ONES by ONES1 give UINT:
greater dividend ONES than divisor ONES1 give the non null UINT @op52 where ONES1 is ONES ONES2.
where UINT is zero.
greater dividend ONES ONESETY than divisor ONES1 give the
non null one ONESETY1:
where ONES is ONES1,
greater dividend ONESETY than ONES1 give the non null
ONESETY1;@op52
where ONESETY1 is EMPTY,
where ONES1 is ONES ONESETY ONESETY2.

the remainder actions for UINT and UINT1 and quotient
UINT2 with exponent part INT results unsigned UINT3 e INT1
with division UINT4:
the integer uplus UINT5 equal to integer uplus UINT4
times integer uplus UINT1;@op27
the integer uplus zero equal to integer uplus UINT minus
integer uplus UINT5;@op27
by the quotient UINT2 and exponent INT on the base of
null or non null quotient make UINT3 e INT1;@op54
the integer uplus UINT5 equal to integer uplus UINT4
times integer uplus UINT1;@op27
the integer uplus ONES equal to integer uplus UINT minus
integer uplus UINT5;@op27
continued actions for non null remainder ONES with
divisor UINT1 and quotient UINT2 and exponent INT to
give UINT3 e INT1;@op55

by the quotient UINT and exponent INT on the base of null
or non null quotient make UINT1 e INT1:
where UINT is zero,
where UINT1 e INT1 is zero e uplus zero;
where UINT is ONES,
where UINT1 is UINT,
where INT1 is INT.

continued actions for non null remainder ONES with divisor
ONES1 and quotient UINT and exponent INT to give UINT1 e
INT1:
the integer uplus UINT2 equal to integer uplus UINT
times integer uplus RADIX integer from uminus INTEGRAL
to uplus INTEGRAL;@op27
the integer uplus UINT3 equal to integer uplus ONES
times integer uplus RADIX;@op27
it might underflow occur in exponent with dividend UINT3
and divisor ONES1 quotient UINT2 exponent part INT give
UINT1 e INT1 with the old quotient UINT;@op56
the integer uplus UINT2 equal to integer UINT times
integer uplus RADIX;@op27
the integer uplus UINT2 greater than uplus
INTEGRAL;@op83
by the quotient UINT and exponent INT on the base of
null or non null quotient make UINT1.e INT1;@op54
H056 it might underflow occur in exponent with dividend UINT and divisor ONES and quotient UINT1 exponent part INT give UINT2 e INT1 with the old quotient UINT3:
The integer INT3 equal to integer INT minus integer
uplus one integer from uminus DECIMAL to uplus
DECIMAL @op27
the real division of unsigned integer UINT by unsined integer
ONES and quotient UINT1 with exponent part INT3
results unsigned integer UINT2 e INT1 @op49
the integer INT2 equal to integer INT minus integer
uplus one @op27
the integer INT2 less than integer uminus DECIMAL @op27
by the quotient UINT3 and exponent INT on the base of
null or non null quotient make UINT2 e INT1 @op54

H057 scalar value or set operation konst UOP UINT tsnok intdiv
konst UOP1 UINT1 tsnok give konst UOP2 UINT2 tsnok:
the result of sign UOP and UOP1 in multiplication and
division operation is UOP @op34
the result of unsigned integer UINT divided by unsigned
integer UINT1 is unsigned integer UINT2 @op50

H058 scalar value or set operation konst INT tsnok mod
konst INT1 tsnok give konst INT2 tsnok:
scalar value or set operation konst INT tsnok intdiv
konst INT1 tsnok give konst INT3 tsnok @op57
the integer INT4 equal to integer INT3 times integer
INT1 @op27
the integer INT2 equal to integer INT minus integer
INT4 @op27

H059 scalar value or set operation konst NAME in one table
tsnok and konst NAME1 in one table tsnok give konst NAME2
in one table tsnok:
where NAME NAME1 NAME2 is TRUE TRUE TRUE ;
where NAME NAME1 NAME2 is TRUE FALSE FALSE ;
where NAME NAME1 NAME2 is FALSE TRUE FALSE ;
where NAME NAME1 NAME2 is FALSE FALSE FALSE .

H060 the result of set operation subset ELEMENTSETY tesbus plus
subset ELEMENTSETY1 tesbus is subset ELEMENTSETY2 tesbus:
nonempty members ELEMENTSETY union with nonempty members
ELEMENTSETY1 give nonempty members ELEMENTSETY2 @op60
where ELEMENTSETY is EMPTY ,
where ELEMENTSETY2 is ELEMENTSETY1 ;
where ELEMENTSETY1 is EMPTY ,
where ELEMENTSETY2 is ELEMENTSETY .

H061 nonempty members ELEMENT ELEMENTSETY union with nonempty
members ELEMENTS give nonempty members ELEMENTS1:
a single element ELEMENT may or might not be in ELEMENTS
so produce ELEMENTS2 @op62
nonempty members ELEMENTSETY union with nonempty members
ELEMENTSETY2 give nonempty members ELEMENTS1 @op61
where ELEMENTSETY is EMPTY ,
a single element ELEMENT may or might not be in ELEMENTS
so produce ELEMENTS1 @op62
H062 a single element ELEMENT may or might not be in ELEMENTS so produce ELEMENTS1: where ELEMENT is not in ELEMENTS, where ELEMENTS1 is ELEMENT ELEMENTS; where ELEMENTS contains ELEMENT, where ELEMENTS1 is ELEMENTS.

H063 the result of set operation subset ELEMENTSETY tesbus minus subset ELEMENTSETY1 tesbus is subset ELEMENTSETY1 tesbus is subset ELEMENTSETY2 tesbus; nonempty elements ELEMENTSETY set different nonempty element ELEMENTSETY1 will give elements ELEMENTSETY2 if op64 where ELEMENTSETY ELEMENTSETY2 is EMPTY; where ELEMENTSETY1 is EMPTY; where ELEMENTSETY2 is ELEMENTSETY.

H064 nonempty elements ELEMENT ELEMENTSETY set different nonempty element ELEMENTS will give elements ELEMENTSETY1: where ELEMENTS is ELEMENTSETY2 ELEMENT ELEMENTSETY3; the result of set operation subset ELEMENTSETY tesbus minus subset ELEMENTSETY2 ELEMENTSETY3 tesbus is subset ELEMENTSETY1 tesbus if op63 where ELEMENT is not in ELEMENTS; where ELEMENTSETY1 is ELEMENT ELEMENTSETY2; the result of set operation subset ELEMENTSETY tesbus minus subset ELEMENTS tesbus is subset ELEMENTSETY2 tesbus; @op63

H065 scalar value or set operation konst NAME in one table konst or konst NAME1 in one table tsnok give konst NAME2 in one table tsnok: where NAME NAME1 NAME2 is TRUE TRUE TRUE; where NAME NAME1 NAME2 is TRUE FALSE TRUE; where NAME NAME1 NAME2 is FALSE TRUE TRUE; where NAME NAME1 NAME2 is FALSE FALSE FALSE.

H066 the result of set operation subset ELEMENTSETY tesbus times subset ELEMENTSETY1 tesbus is subset ELEMENTSETY2 tesbus: the intersection of nonempty ELEMENTSETY and nonempty ELEMENTSETY1 must be ELEMENTSETY2 @op67 where ELEMENTSETY is EMPTY; where ELEMENTSETY2 is EMPTY; where ELEMENTSETY1 is EMPTY; where ELEMENTSETY2 is EMPTY.
The intersection of nonempty ELEMENT ELEMENTSET1 and nonempty ELEMENTS must be ELEMENTSET2:
where ELEMENTS contains ELEMENT,
where ELEMENTSET1 is ELEMENT ELEMENTSET2,
the result of set operation subset ELEMENTSET1 times subset ELEMENTS is subset ELEMENTSET2.
where ELEMENT is not in ELEMENTS,
the result of set operation subset ELEMENTSET1 times subset ELEMENTS is subset ELEMENTSET2.

STACK and HEAPETY the RSIDE will give value or set or pointer or sequence DBOXORSTRG:
STACK and HEAPETY from RSIDE which is an address obtain the sequence DBOXORSTRG.
STACK and HEAPETY the RSIDE eventually introduces DBOXORSTRG.

STACK and HEAPETY from ADDRESS of array defined by one dimensional left TYPE right of CHARTYPE needs ONESET1 boxes which is an address obtain the sequence string KCHARS:
STACK and HEAPETY from ADDRESS access to a group of ONESET1 characters KCHARS.

STACK and HEAPETY from STKHP ONESt box access to a group of ONESET1 characters KCHARS:
STACK and HEAPETY the STKHP ONESt box eventually introduces konst KCHAR tsnok.

TABLES the four types of relational operation DBOXORSTRG RELOP DBOXORSTRG1 introduce the boolean IDEN:
string value DBOXORSTRG RELOP string value DBOXORSTRG1 is equal to boolean IDEN.
TABLES scalar or set or pointer operands DBOXORSTRG
RELOP DBOXORSTRG1 give boolean value IDEN.

string value string KCHARS end RELOP string value string KCHARS1 end is equal to boolean IDEN:
there are ONESET1 characters in a string KCHARS,
there are ONESET1 characters in a string KCHARS1,
first character of KCHARS RELOP first character of KCHARS1 might result the boolean IDEN.
H073 first character of KCHAR KCHARSETY RELOP first character of KCHAR1 KCHARSETY1 might result the boolean IDEN ;
where KCHAR is KCHAR1 ;
first character of KCHARSETY RELOP first character of KCHARSETY1 might result the boolean IDEN i3op73
where KCHAR differs from KCHAR1 ;
find the order of character KCHAR in KARSET to be INT i3op74
find the order of character KCHAR1 in KARSET to be INT1 i3op74
integer INT RELOP integer INT1 give boolean answer IDEN i3op77-82
where KCHARSETY KCHARSETY1 is EMPTY ;
where KCHAR is KCHAR1 ;
find the order of character KCHAR in KARSET to be INT i3op74
integer INT RELOP integer INT give boolean answer IDEN i3op77-82

H074 find the order of character KCHAR in KCHAR1 KCHARSETY to uplus one ONESETY ;
where KCHAR is KCHAR1 ;
where ONESETY is EMPTY ;
where KCHAR differs from KCHAR1 ;
find the order of character KCHARSETY to be uplus ONESETY. i3op74
H075 TABBLES scalar value or set or pointer operands BOX RELOP DBOX1 give boolean value IDEN :
TABBLES scalar value BOX RELOP scalar value DBOX1 give IDEN i3op76
pointer value DBOX RELOP pointer value DBOX1 give IDEN i3op90-91
set value DBOX RELOP set value DBOX1 give IDEN. i3op92-95
H076 TABBLES scalar value konst VALUE tsnok RELOP scalar value konst VALUE1 tsnok give IDEN :
integer VALUE RELOP integer VALUE1 give boolean answer IDEN i3op77-82
integer VALUE RELOP real VALUE1 give boolean answer IDEN i3op84
real VALUE RELOP integer VALUE1 give boolean answer IDEN i3op85
real VALUE RELOP real VALUE1 give boolean answer IDEN i3op86
single character VALUE RELOP single character VALUE1 will prepare IDEN i3op87
TABBLES identifier value VALUE RELOP identifier VALUE1 introduce boolean IDEN. i3op88
H077 integer INT equal to integer INT1 give boolean answer IDEN :
where INT is INT1 ;
where IDEN is TRUE in one table ;
where INT differs from INT1 ;
where IDEN is FALSE in one table.
H078 integer INT unequal to integer INT1 give boolean answer
   IDEN :
   where INT differs from INT1 ,
   where IDEN is TRUE in one table ;
   where INT is INT1 ,
   where IDEN is FALSE in one table .

H079 integer INT less than integer INT1 give boolean answer
   IDEN :
   the integer uminus ONES equal to integer INT minus
   integer INT1 @op27
   where IDEN is TRUE in one table ;
   the integer uplus ONES equal to integer INT minus
   integer INT1 @op27
   where IDEN is FALSE in one table ;
   the integer UOP zero equal to integer INT minus
   integer INT1 @op27
   where IDEN is FALSE in one table .

H080 integer INT greater than integer INT1 give boolean answer
   IDEN :
   integer INT1 less than integer INT give boolean answer
   IDEN.@op79

H081 integer INT less or equal to integer INT1 give boolean
   answer IDEN :
   the integer uminus ONES equal to integer INT minus
   integer INT1 @op27
   where IDEN is TRUE in one table ;
   the integer UOP zero equal to integer INT minus integer
   INT1 @op27
   where IDEN is TRUE in one table ;
   the integer uplus ONES equal to integer INT minus integer
   INT1 @op27
   where IDEN is FALSE in one table .

H082 integer INT greater or equal to integer INT1 give boolean
   answer IDEN :
   integer INT1 less or equal to integer INT give boolean
   answer IDEN.@op81

H083 the integer INT RELOP integer INT1 :
   integer INT RELOP integer INT1 give boolean answer
   TRUE in one table.@op77-82

H084 integer INT RELOP real INT1 e INT2 give boolean answer
   IDEN :
   real INT e uplus zero RELOP real INT1 e INT2 give
   boolean answer IDEN.@op86

H085 real INT e INT1 RELOP integer INT2 give boolean answer
   IDEN :
   real INT e INT1 RELOP real INT2 e uplus zero give
   boolean answer IDEN.@op86
H086 real INT e INT1 RELOP real INT2 e INT3 give boolean answer IDEN:
the integer INT1 grater than integer INT3 @op83
the integer uplus ONES equal to integer INT1 minus integer INT3 @op27
the production of INT and ten to non null ONES is INT4 due to @op46
integer INT4 RELOP integer INT2 give boolean answer IDEN @op77-82
the integer INT3 greater than integer INT1 @op83
the integer uplus ONES equal to integer INT3 minus integer INT1 @op27
the production of INT2 and ten to non null ONES is INT4 due to @op46
integer INT RELOP integer INT4 give boolean answer IDEN @op77-82
where INT1 is INT3 , integer INT RELOP integer INT2 give boolean answer IDEN.@op77-82

H087 single character KCHAR RELOP single character char
KCHAR1 will prepare IDEN:
find the order of character KCHAR in KARSET to be INT @op74
find the order of character KCHAR1 in KARSET to be INT1 @op74
integer INT RELOP integer INT1 give boolean answer IDEN.@op77-82

H088 TABLESETY the ONESth LOCSETY loc NAMETY be type of scalar
defined by BNAMES needs one boxes end LOCSETY1 slink to ONESETY table TABLESETY1 identifier value NAME in ONES1 table
RELOP identifier value NAME1 in ONES1 table introduce IDEN:
where ONES is ONES1 ,
where BNAMES contains left NAME right ,
where BNAMES contains left NAME1 right ,
find the order of identifier left NAME right in BNAMES to be INT @op89
find the order of identifier left NAME1 right in BNAMES to be INT1 @op89
integer INT RELOP integer INT1 give boolean answer IDEN.@op77-82

H089 find the order of identifier BNAME in BNAME1 BNAMESETY to be uplus one ONESETY :
where BNAME1 is BNAME ,
where ONESETY is EMPTY ;
where BNAME differs from BNAME1 ,
find the order of identifier BNAME in BNAMESETY to be uplus ONESETY.@op89

H090 pointer value DBOX equal to pointer value DBOX1 give IDEN:
where DBOX is DBOX1 ,
where IDEN is TRUE in one table ;
where DBOX differs from DBOX1 ,
where IDEN is FALSE in one table.
H091 pointer value DBOX unequal to pointer value DBOX1 give IDEN:
  where DBOX differs from DBOX1,
  where IDEN is TRUE in one table,
  where DBOX is DBOX1,
  where IDEN is FALSE in one table.

H092 set value SET equal to set value SET1 give IDEN:
  set value SET less or equal to set value SET1 give TRUE in one table \( \oplus \) op95
  set value SET1 less or equal to set value SET give TRUE in one table \( \oplus \) op95
  where IDEN is TRUE in one table;
  where IDEN is FALSE in one table.

H093 set value SET unequal to set value SET1 give IDEN:
  set value SET less or equal to set value SET1 give FALSE in one table \( \ominus \) op95
  where IDEN is TRUE in one table;
  set value SET1 less or equal to set value SET give FALSE in one table \( \ominus \) op95
  where IDEN is TRUE in one table;
  where IDEN is FALSE in one table.

H094 set value SET greater or equal to set value SET1 give IDEN:
  set value SET1 less or equal to set value SET give IDEN.\( \oplus \) op95

H095 set value subset ELEMENTSETY tesbus less or equal to set value SET give IDEN:
  all items of ELEMENTSETY must belong to SET to give IDEN \( \ominus \) op96
  where ELEMENTSETY is EMPTY,
  where IDEN is TRUE in one table.

H096 all items of ELEMENT ELEMENTSETY must belong to SET to give IDEN:
  ELEMENT may or may not be in set SET to produce IDEN \( \oplus \) op97
  where IDEN1 is TRUE in one table,
  set value subset ELEMENTSETY tesbus less or equal to set value SET give IDEN \( \ominus \) op95
  ELEMENT may or may not be in set SET to produce IDEN \( \ominus \) op97
  where IDEN is FALSE in one table.

H097 ELEMENT may or may not be in set SET to produce IDEN:
  where SET contains ELEMENT,
  where IDEN is TRUE in one table;
  where ELEMENT is not in SET,
  where IDEN is FALSE in one table.

H098 membership operation KONSTANT in SET give IDEN:
  elem KONSTANT mele may or may not be in set SET to produce IDEN.\( \oplus \) op97
H099 the boolean value FALSE in one table is opposite of the boolean value TRUE in one table:
true.

H100 the boolean value TRUE in one table is opposite of the boolean value FALSE in one table:
true.

H101 TABLES obtain the ordinal number VALUE to be INT:
TABLES the ordinal of scalar identifier VALUE is INT:
find the order of character VALUE in KARSET to be INT:
the ordinal of integer VALUE is itself and so the same as INT.

H102 TABLESETY the ONESth LOCSETY loc NAMETY be type of scalar defined by BNAMESETY needs one boxes end LOCSETY1 slink to ONESETY table TABLESETY1 the ordinal of scalar identifier NAME in ONES1 table is INT:
where ONES is ONES1,
where BNAMESETY contains left NAME right,
find the order of identifier left NAME right in BNAMESETY to be INT.

H103 the ordinal of integer INT is itself and so the same as INT:
where INT is INT.

H104 the integer successor of INT due to INTERVAL should be INT:
the integer INT1 equal to integer INT uplus integer uplus one INTERVAL.

H105 the character successor of KCHAR due to KCHARS should be KCHAR1:
where KCHARS contains KCHAR KCHAR1.

H106 TABLESETY the ONESth LOCSETY slink to ONESETY table TABLESETY1 the successor of scalar identifier NAME in ONES1 table must be in ONES2 table:
where ONES is ONES1,
where LOCSETY contains loc NAMETY be scalar defined by BNAMESETY left NAME right left NAME1 right BNAMESETY1 needs one boxes end,
where ONES2 is ONES.
Definitions related to stack frames

H001 TABLES elementary actions to transfer the standard table
the oneth LOCs slink to table in the oneth frame from one
1 table and FLOCS link to frame:
withdraw nonempty LOCs from type locations to give LOCs1
LOC "fr2
TABLES nonempty locations LOCs1 build memory locations
FLOCS present box onen."fr3

H002 withdraw nonempty LOCSETY loc NAME be type of TYPE end
LOCSETY1 from type locations to give LOCSETY2:
withdraw nonempty LOCSETY LOCSETY1 from type locations
to give LOCSETY2 "fr2
where LOCSETY2 is LOCSETY LOCSETY1.

H003 TABLES nonempty locations LOC LOCSETY build memory
locations FLOC FLOCSETY present box ONES:
TABLES single LOC make memory unit FLOC present box ONES
last ONES1 "fr4,6,8,9
TABLES nonempty locations LOCSETY build memory locations
FLOCSETY present box ONES1 "fr3
where LOCSETY FLOCSETY is EMPTY:
TABLES single LOC make memory unit FLOC present box ONES
last ONES1, "fr4,6,8,9

H004 TABLES single loc NAME be formal parameter of SHAPE of
IDEN end make memory unit loc NAME1 refers the ONESth
undefined ht end present box ONES1 last ONES2:
SHAPE could be in one of three forms "fr5
where NAME1 is NAME:
where ONES is ONES1:
where ONES2 is ONES1 one.

H005 SHAPE could be in one of three forms:
where SHAPE is variable:
where SHAPE is FUNPROC.

H006 TABLES single loc NAME be formal parameter of value of
IDEN end make memory unit loc NAME1 refers NBOXES end
present box ONES last ONES1:
where NAME1 is NAME:
TABLES the type identifier IDEN introduce TYPEDEF needs
ONES2 boxes "fry12
there are ONES2 boxes of NBOXES present box ONES last
box ONES1."fr7

H007 there are one ONESETY boxes of the ONESth undefined
NBOXESETY present box ONES1 last box ONES2:
where ONES is ONES1:
there are ONESETY boxes of NBOXESETY present box ONES1
one last box ONES2 "fr7
where ONESETY NBOXESETY is EMPTY:
where ONES is ONES1:
where ONES2 is ONES1 one.
H008 TABLES single loc NAME be variable of TYPE end make memory
unit loc NAME1 refers NBOXES end present box ONES last
ONES1 :
  where NAME1 is NAME ;
  TABLES from the type TYPE get type definition TYPEDEF
  needs ONES13 boxes @ty2
  there are ONES2 boxes of NBOXES present box ONES last
  ONES1.@fr7

H009 TABLES single loc NAME be FUNPROC of PROCFDEC end make
memory unit loc NAME1 refers end present box ONES last
ONES1 :
  where NAME1 is NAME ,
  where ONES1 is ONES .

H010 TABLES create a new frame on the oneth frame from one
table FLOCSETY FLOC link to frame for the ONES table to
produce STACK :
  take LOCSETY from ONESth table of TABLES @fr11
  TABLES might empty LOCSETY make memory locations
  FLOCSETY1 present box one @fr12
  where STACK is the oneth frame from one table FLOCSETY
  FLOC link to frame the one oneth frame from ONES table
  FLOCSETY1 link to one frame .

H011 take LOCSETY from ONESth table of TABLESETY the ONES1th
LOCSETY1 slink to ONESETY table TABLESETY1 :
  where ONES1 is ONES ,
  where LOCSETY is LOCSETY1 .

H012 TABLES might empty LOCSETY make memory locations FLOCSETY
present box ONES :
  evacuate LOCSETY from constant and type locations to
give LOCSETY1 @fr13
  TABLES the possibly empty locations LOCSETY1 produce
  memory FLOCSETY present box ONES.@fr17

H013 evacuate LOCSETY from constant and type locations to give
LOCSETY1 :
  evacuate constant and type locations from nonempty
  LOCSETY to give LOCSETY1 @fr14
  where LOCSETY LOCSETY1 is EMPTY .

H014 evacuate constant and type locations from nonempty LOCS
to give LOCSETY :
  evacuate LOCS from constant locations to give
  LOCSETY .@fr15
  evacuate LOCS from type locations to give LOCSETY.@fr16

H015 evacuate LOCSETY loc NAME be constant of CONST end
LOCSETY1 from constant locations to give LOCSETY2 :
  evacuate LOCSETY LOCSETY1 from constant locations
to give LOCSETY2 @fr15
  evacuate LOCSETY LOCSETY1 from type locations to give
  LOCSETY1.@fr16
H016 evacuate LOCSETY from type locations to give LOCSETY1:
withdraw nonempty LOCSETY from type locations to give
LOCSETY1 if LOCSETY is EMPTY.

H017 TABLES the possibly empty locations LOCSETY produce memory
FLOCSETY present box ONES:
TABLES nonempty locations LOCSETY build memory locations
FLOCSETY present box ONES if LOCSETY is EMPTY.

H018 TABLES open new frame on STACK from ONES table to
introduce STACK1 with FUNPROC head NAME and linkage ONES1:
find the last frame ONES2 box ONESETY from STACK, if
the nature of FUNPROC create a number ONESETY1, if
there might be ONESETY1 boxes of NBOXETY present box
ONESETY one last box ONES3 if
take LOCSETY from ONES1th table of TABLES, if
TABLES might empty LOCSETY make memory locations
FLOCSETY1 present box ONES3 if
where STACK1 is STACK the ONES2 oneth frame from ONES
table the block NAME be FUNPROC refers NBOXETY and
FLOCSETY1 link to ONES1 frame.

H019 find the last frame ONES box ONESETY frame FRAMESETY the
ONES1th FRAME1 link to ONESETY1 frame:
where ONES is ONES1,
find the last box no ONESETY in might empty FRAMESETY
the ONES1th FRAME1 link to ONESETY1 frame.

H020 find the last box no ONESETY in might empty FRAMESETY:
find the last box no ONESETY in nonempty
FRAMESETY if
where FRAMESETY is EMPTY,
where ONESETY is EMPTY.

H021 find the last box no ONESETY in nonempty FRAMESETY the
ONES1th HEADETY and FLOCSETY link to ONESETY1 frame:
box number ONESETY is found from locations of
FLOCSETY if
if FLOCSETY either have not box or be empty, if
box number ONESETY be found from head of HEADETY, if
if FLOCSETY either have not box or be empty, if
when HEADETY either be procedure head or empty, if
find last box no ONESETY in might empty FRAMESETY.

H022 box number ONES is found from locations of FLOCSETY
loc NAME refers NBOXESEETY end:
ONES could be the top box number of NBOXESEETY if
where NBOXESEETY is EMPTY,
box number ONES is found from locations of
FLOCSETY.

H023 ONES could be the top box number of NBOXESEETY the ONES1th
BOX:
where ONES is ONES1.
H024 if FLOCSETY either have not box or be empty:
   whenever FLOCSETY has not box H025 where FLOCSETY is EMPTY.

H025 whenever FLOCSETY loc NAME refers end has not box:
   if FLOCSETY either have not box or be empty.H024

H026 box number ONES be found from head of the block NAME be
   function of TYPE refers the ONES1th BOX:
   where ONES is ONES1.

H027 when HEADETY either be procedure head or empty:
   nonempty HEADETY would be a procedure H028 where HEADETY is EMTPY.

H028 nonempty the block NAME be procedure refers would be a
   procedure:
   true.

H029 the nature of FUNPROC creat a number ONESETY:
   where FUNPROC is procedure:
   where ONESETY is EMPTY:
   where FUNPROC is function:
   where ONESETY is one.

H030 there might be ONESETY boxes of NBOXESETY present box ONES
   last box ONES1:
   there are ONESETY box of NBOXESETY present box ONES last
   box ONES1 H030
   where ONESETY NBOXESETY is EMPTY,
   where ONES1 is ONES.
Definitions related to input output

H001 KCHARSETY end of zero file chars in a line reset input file to give INFILF:
  nonempty KCHARSETY from input file preparing the
  KCHARS;@fi2
  where INFILF is KCHARS end of zero file chars in a
  line;
  where KCHARSETY is EMPTY;
  @=f7
  where INFILF is end of one file chars in a line.

H002 nonempty char CHAR KCHARSETY from input file preparing the
  KCHARS:
    CHAR symbol ,@sy3-94
  where KCHR is char CHAR KCHARSETY.

H003 KCHARSETY end of UINT file ONESETY chars in a line
  execute standard function end of line or end of file ELON
to give konst IDEN tsnok:
  where INLINLEN is ONESETY;
  where IDEN is TRUE in one table;
  where INLINLEN contains ONESETY one;
  where IDEN is FALSE in one table.

H004 KCHARSETY end of UINT file ONESETY char in a line execute
  standard function end of line end of file EOF to give
  konst IDEN tsnok:
  where UINT is one;
  where IDEN is TRUE in one table;
  where UINT is zero;
  where IDEN is FALSE in one table.

H005 INFILF with STACK and HEAPETY input the address of
  TYPADRES to make INFILF1 with STACK1 and HEAPETY1:
    INFILF with STACK and HEAPETY when input a character in
    TYPADRES produce INFILF1 with STACK1 and HEAPETY1 @=fi6
    INFILF with STACK and HEAPETY input might space with
    number in address TYPADRES to give INFILF1 with STACK1
    and HEAPETY1.@fi11

H006 INFILF with STACK and HEAPETY when input a character in
  ADDRESS of SCRG defined by KCHARS needs one boxes produce
INFILF1 with STACK1 and HEAPETY1:
  remove end of line from INFILF when necessary to be
INFILF2.@fi7
  INFILF2 with STACK and HEAPETY read a character in
  address ADDRESS of SCRG defined by KCHARS needs one
  boxes to deleiver INFILF1 with STACK1 and HEAPETY1.@fi9
H007 remove end of line from KCHARS end of zero file ONESETY chars in a line when necessary:
where ONESETY is INLINELEN,
determine syntax of the first character of KCHARS and with give INFilen.9f18
where INLINELEN contains ONESETY one,
where INFilen is KCHARS end of zero file ONESETY chars in a line.

H008 determine syntax of the first character of char CHAR
KCHARSETY and with ONESETY give INFilen:
CHAR symbol .@sy3-?4
where INFilen is char CHAR KCHARSETY end zero file ONESETY chars in a line.

H009 KCHAR KCHARSETY end of zero file ONESETY chars in a line
with STACK and HEAPETY read a character in address ADDRESS
of SCRG defined by KCHARS needs one boxes to deliver
INFilen with STACK1 and HEAPETY1:
where KCHARS contains KCHAR,
STACK and HEAPETY put the quantity of konst KCHAR tsnok
in ADDRESS to have STACK1 and HEAPETY1 .@op6
KCHARSETY end of zero file ONESETY one chars in a line
get the input file to be INFilen.9f10

H010 KCHARSETY end of zero file ONES chars in a line get the
input FILE:
where KCHARSETY is EMPTY :
eof ;
where INFilen is end of one file ONES chars in a line:
where ONES is INLINELEN ,
where INFilen is KCHARSETY end of zero file ONES chars in a line:
where INLINELEN contains ONES one,
determine syntax of the first character of KCHARSETY and with ONES give INFilen.9f18

H011 char CHAR KCHARSETY end of zero file ONESETY chars in a
a with STACK and HEAPETY input might space with number in
address TYPADRES to give INFilen with STACK1 and HEAPETY1:
where CHAR is space,
KCHARSETY end of zero file ONESETY one chars in a line
after removing preceded blanks to have INFILE1 .@f112
INFILE1 with STACK and HEAPETY input might space with
number in address TYPADRES to give INFILE with STACK1
and HEAPETY1 .@f112
where CHAR differs from space ,
char CHAR KCHARSETY end of zero file ONESETY chars in a
line with STACK and HEAPETY input number in address
TYPADRES to give INFILE with STACK1 and HEAPETY1 .@f113
H012 KCHARS end of zero file ONES chars in a line after removing preceded blanks to have INFILE:
   where ONES is INLINLEN,
   determine syntax of the first character of KCHARS and
   with give INFILE 1.  
where INLINLEN contains ONES one,
   determine syntax of the first character of KCHARS and
   with ONES give INFILE. 3.8

H013 INFILE with STACK and HEAPETY input number in address
   ADDRESS of TYPE to give INFILE1 with STACK and HEAPETY:
   INFILE delete the number TNUMBER from input file to give
   INFILE1 1.  
decimal base number TNUMBER will prepare a unary number
   NUMBER  1. 2. 5
   due to compatibility NUMBER with TYPE could have new
   NUMBER1 1. 2. 6-2.8
   STACK and HEAPETY put the quantity of konst NUMBER1 tsnok
   in ADDRESS to have STACK1 and HEAPETY1. 5.6

H014 char CHAR KCHARSETY end of zero file ONESETY chars in a
   line delete the number SIGN UTNUMBER from input file to
   give INFILE:
   where CHAR is SIGN,
   determine syntax of the first character of KCHARSETY and
   with ONESETY one give INFILE1 1. 8
   INFILE1 take the unsigned number UTNUMBER from input
   file to give INFILE 1. 15
   where CHAR differs from SIGN1:
   where SIGN is +,
   char CHAR KCHARSETY end of zero file ONESETY chars in a
   line take the unsigned number UTNUMBER from input file
to give FILE. 1.15

H015 INFILE take the unsigned number UTNUMBER from input file
   to give INFILE1:
   INFILE take the unsigned integer UTNUMBER to have
   INFILE1 1. 16
   INFILE take the unsigned real UTNUMBER to have
   INFILE1 1. 2. 24

H016 char DIGIT KCHARSETY end of zero file ONESETY chars in a
   line take the unsigned integer DIGIT1 DIGITSETY to have
   INFILE:
   end of line ONESETY one or end of file KCHARSETY or both
   give UINT 1. 17
   where DIGITSETY is EMPTY,
   where DIGIT1 is DIGIT,
   where INFILE is KCHARSETY end of UINT file ONESETY one
   chars in a line:
   where INLINLEN contains ONESETY one one,
   where DIGIT1 is DIGIT,
   determine syntax of the first character of KCHARSETY and
   with ONESETY one give INFILE1 1. 18
   INFILE1 the first char be digit or nondigit to make
   DIGITSETY and give INFILE. 1.18
H017 end of line ONES or end of file KCHARSETY or both give UINT:
    where KCHARSETY is EMPTY,
    eof,
    where UINT is one,
    where INLINLEN is ONES;
    where KCHARSETY is EMPTY,
    eof,
    where UINT is one,
    where INLINLEN contains ONES one;
    where INLINLEN is ONES,
    where UINT is zero.

H018 char CHAR KCHARSETY end of zero file ONES chars in a line
the first char be digit or non digit to make DIGITSETY and
give INFILE:
    where CHAR is DIGIT,
    char CHAR KCHARSETY end of zero file ONES chars in a
    line take the unsigned integer DIGITSETY to have
    INFILE 16
    where CHAR differs from DIGIT,
    where INFILE is char CHAR KCHARSETY end of zero file
    ONES chars in a line.

H019 KCHARS char . KCHARS1 KCHARETY end of zero file ONESETY
chars in a line take the unsigned real DIGITS dot DIGITS1
EXPONENTY to have INFILE:
    KCHARS check syntax except first character and give
    DIGITS and ONESETY character number 16
    determine syntax of first character of KCHARS1 and with
    ONESETY ONES one give FILE1 18
    FILE1 decimal part of real number give DIGITS1 EXPONENTY
    and new FILE.22

H020 char DIGIT KCHARSETY check the syntax except first
character and give DIGIT1 DIGITSETY and one ONESETY
character number:
    where DIGIT1 is DIGIT,
    KCHARSETY check the syntax of more digits to have
    DIGITSETY and ONESETY number char 16
    where KCHARSETY DIGITSETY ONESETY is EMPTY,
    where DIGIT1 is DIGIT.

H021 char DIGIT KCHARSETY check the syntax of more digits to
have DIGIT1 DIGITSETY and one ONESETY number of char:
    DIGIT symbol 93-94
    where DIGIT1 is DIGIT,
    KCHARSETY check the syntax of more digits to have
    DIGITSETY and ONESETY number of char 16
    where KCHARSETY DIGITSETY ONESETY is EMPTY,
    DIGIT symbol 93-94
    where DIGIT1 is DIGIT.
H022 char DIGIT KCHARSETY end of zero file ONES chars in a line
decimal part of real number give DIGIT1 DIGITSETY EXPONETY
and new INFILE :
end of line ONES one or end of file KCHARSETY or both
give UINT i8fi17
where DIGIT1 is DIGIT ,
where KCHARSETY DIGITSETY EXPONETY is EMPTY ,
where INFILE is KCHARSETY end of UINT file ONES one
chars a line ;
determine syntax of first character of KCHARSETY and
with ONES one give char DIGIT KCHARSETY1 end of zero
file ONES one chars in a line i8fi18
char DIGIT2 KCHARSETY1 end of zero file ONES one chars
in a line decimal part of real number give DIGITSETY
EXPONETY and new INFILE i8fi22
determine syntax of first character of KCHARSETY and
with ONES one give char e KCHARSETY1 end of zero file
ONES one chars in a line i8fi18
where DIGITSETY is EMPTY ;
char e KCHARSETY1 end of file ONES one chars in a line
exponential part of real number EXPONETY give
INFILE.@fi23

H023 char e KCHARS end of zero file ONES chars in a line
exponential part of real number e INT give INFILE :
determine syntax of the first character of KCHARS and
with ONES one give INFILE1 i8fi8
INFILE1 delete the number TINT from input file to give
INFILE.@fi14

H024 KCHARS char e KCHARS1 end of zero file ONESETY chars in a
line take unsigned real DIGITS e TINT to have INFILE :
KCHARS check syntax except first character and give
DIGITS and ONES character number i8fi20
e ,
char e KCHARS1 end of zero file ONESETY ONES chars in a
exponential part of real number e TINT give INFILE.@fi23

H025 decimal base number SIGN UTNUMBER will prepare a unary
number UOP UNUMBER :
where SIGN UOP is + uplus ,
UOP unsigned number UTNUMBER produce UNUMBER iaco5
where SIGN UOP is - uminus ,
UOP unsigned number UTNUMBER produce UNUMBER.3co5

H026 due to compatibility INT with SCRG defined integer from
INT1 to INT2 could have new INT3 :
where INT3 is INT ,
the integer INT3 is in the interval integer from INT1 to
INT2.3op31

H027 due to compatibility INT with scalar defined by REALTYPE
could have new REYAL :
the two cases for real INT e uplus zero will give
REYAL.3co26+27
due to compatibility INT e INT1 with scalar defined by REALTYPE could have new REYAL:
the two cases for real INT e INT1 will give REYAL.2co26\[27

KCHARSETY end of zero file ONESETY chars in a line execute
input newline procedure to give INFILE:
where KCHARSETY is EMPTY:
EOF
where INFILE is end of one file chars in a line:
where LINLEN contains ONESETY one,
KCHARSETY end of zero file ONESETY chars in a line take
one character to give INFILE i3fi30
where ONESETY is LINLEN,
determine syntax of first character of KCHARSETY and
with give INFILE.3fi38

char CHAR KCHARSETY end of zero file ONESETY chars in a
line take one character to give INFILE:
CHAR symbol.2sy3-94
KCHARSETY end of zero file ONESETY one chars in a line
execute input newline procedure to give INFILE.2fi29

OUTFILE write in output file either an string or constant
value CONST to have OUTFILE1:
OUTFILE print the string of characters CONST in output
file OUTFILE1 i3fi32
OUTFILE print the constant value CONST in output file
OUTFILE1.2fi35.37.38

OUTFILE print the string of characters string KCHARS end
in output file OUTFILE1:
OUTFILE write in output file a string of KCHARS to give
OUTFILE1.2fi33

OUTFILE write in output file a string of KCHAR KCHARSETY
to give OUTFILE1:
OUTFILE output single character KCHAR to the output file
OUTFILE2.2fi34
OUTFILE2 write in output file a string of KCHARSETY to
give OUTFILE1 i3fi33
where KCHARSETY is EMPTY,
OUTFILE output single character KCHAR to the output file
OUTFILE1.2fi34

KCHARSETY with ONESETY chars in a line and ONESETY1 total
chars output single character KCHAR to the output file
OUTFILE1
where MAXOUT contains ONESETY1 one,
where OULINLEN is ONESETY,
where OUTFILE is KCHARSETY KCHAR with one chars in a
line and ONESETY1 one total chars:
where MAXOUT contains ONESETY1 one,
where OULINLEN contains ONESETY one,
where OUTFILE is KCHARSETY KCHAR with ONESETY one chars
in line and ONESETY1 one total chars.
H035 OUTFILE print the constant value konst left NAME in one table right tsnok in output file OUTFILE1:
  the boolean NAME will be turned to KCHARS.
OUTFILE write in output file a string of KCHARS to give OUTFILE1.

H036 the boolean letter CHAR LETTERSETY will be turned to char CHAR1 KCHARSETY:
  where CHAR1 is CHAR,
  the boolean LETTERSETY will be turned to KCHARSETY.
  where LETTERSETY CHARSETY is EMPTY,
  where CHAR1 is CHAR.

H037 OUTFILE print the constant value konst KCHAR tsnok in output file OUTFILE1:
  OUTFILE output single character KCHAR to the output file OUTFILE1.

H038 OUTFILE print the constant value konst NUMBER tsnok in output file OUTFILE1:
  base one integer NUMBER is transfered to integer format KCHARS.
  OUTFILE after inserting of number KCHARS produce OUTFILE1.
  base one real NUMBER is transfered to real format KCHARS.
  OUTFILE after inserting of number KCHARS produce OUTFILE1.

H039 base one integer INT is transfered to integer format KCHAR KCHARSETY:
  the unary integer INT would be transfered to KCHARS.
  fill KCHARS up to KCHARSETY due to number format INFORMAT.

H040 the unary integer UOP UINT would be transfered to char CHAR KCHARS:
  where CHAR UOP is + uplus,
  the unsigned unary integer UINT is transfered to unsigned integer format KCHARS.
  where CHAR UOP is - uminus,
  the unsigned unary integer UINT is transfered to unsigned integer format KCHARS.

H041 the unsigned unary integer UINT is transfered to unsigned integer format KCHARS:
  unsigned non null unary integer UINT give unsigned integer format KCHARS.
  where UINT is zero,
  unary digit UINT is equivalent to KCHARS.
unsigned non null unary integer ONES give unsigned integer format KCHARSETY KCHAR:
the result of unsigned integer ONES divided by unsigned integer RADIX is unsigned integer ONES1, @op50 scalar value or set operation konst uplus ONES tsnok mod konst uplus RADIX tsnok give konst uplus UINT tsnok, @op58
unary digit UINT is equivalent to KCHAR, @fi43
unsigned non null unary integer ONES1 give unsigned integer format KCHARSETY @fi42
where KCHARSETY is EMPTY,
the integer uplus ONES less than integer uplus RADIX, @op83
unary digit ONES is equivalent to KCHAR, @fi43

unary digit UINT is equivalent to char CHAR:
where UINT CHAR is zero 0;
where UINT CHAR is one 1;
where UINT CHAR is one one 2;
where UINT CHAR is one one one 3;
where UINT CHAR is one one one one 4;
where UINT CHAR is one one one one one 5;
where UINT CHAR is one one one one one one 6;
where UINT CHAR is one one one one one one one 7;
where UINT CHAR is one one one one one one one one 8;
where UINT CHAR is one one one one one one one one one one one 9.

fill KCHARS up to KCHARSETY due to number format ONES:
there exist ONES1 characters in the sequence of KCHARS, @fi45
where ONES is ONES1 ONES2, KCHARSETY there are ONES2th spaces in front @fi46
where KCHARSETY is EMPTY,
there exist ONES characters in the sequence of KCHARS, @fi44

there exist one ONESETY characters in the sequence of KCHAR KCHARSETY:
there exist ONESETY characters in the sequence of KCHARSETY, @fi45
where ONESETY KCHARSETY is EMPTY.

char space KCHARSETY there are one ONESETYth spaces in front:
KCHARSETY there are ONESETYth spaces in front, @fi46
KCHARSETY ONESETY is EMPTY.

KCHARSETY with ONESETY chars in a line and ONESETY1 total chars after inserting of number KCHARS produce OUTFILE:
there exist ONES characters in the sequence of KCHARS, @fi45
where OUTLINLEN contains ONESETY ONES,
where MAXOUT contains ONESETY1 ONES,
where OUTFILE is KCHARSETY KCHARS with ONESETY ONES chars in a line and ONESETY1 ONES total chars.
H048 base one real INT e INT1 is transferred to real format
KCHARS char e KCHARS1 KCHARSETY :
the unary integer INT would be transferred to
KCHARS \( @f140 \)
the unary integer INT1 would be transferred to
KCHARS1 \( @f140 \)
fill KCHARS char e KCHARS1 up to KCHARSETY due to number
format REALFORMAT.\( @f144 \)

H049 KCHARSETY with ONESETY chars in a line and ONESETY1 total
chars execute output newline procedure to give OUTFILE1 :
where OUTLINLEN is ONESETY ONES ;
where MAXOUT contains ONESETY1 ONES ;
KCHARS there are ONESTh spaces in front \( @f146 \)
where OUTFILE is KCHARSETY KCHARS with chars in a line
and ONESETY1 ONES total chars ;
where OUTLINLEN is ONESETY ;
where OUTFILE is KCHARSETY with chars in a line and
ONESETY1 total chars.
SY

Syntax definitions

H001 limitation on NAME identifier:
   NAME identifier $sy2,
   where left NAME right is not in WORDDELMITERS,
   where MAXLENID contains NAME.

H002 letter ALPHA LETTERSETY identifier:
   ALPHA symbol $sy3-94,
   LETTERSETY denatation $sy95;
   where LETTERSETY is EMPTY,
   ALPHA symbol $sy3-94.

H003 A symbol : A.

H004 B symbol : B.
   :.
   |:
   H027 Z symbol : Z.

H028 a symbol : a.

H029 b symbol : b.
   :.
   :.
   H053 z symbol : z.

H054 0 symbol : 0.

H055 1 symbol : 1.

H056 2 symbol : 2.
   :.
   :.
   H063 9 symbol : 9.

H064 + symbol : ±.

H065 - symbol : =.

H066 * symbol : *.

H067 / symbol : /.
   :.
   :.

H084 double colon symbol : ::
H085 $ symbol : $.

H093 ? symbol : ?.

H094 space symbol : .

H095 letter ALDIG LETTERSETY denotation : ALDIG symbol \symbol{94} LETTERSETY denotation \symbol{95}
where LETTERSETY is EMPTY ; ALDIG symbol.\symbol{94}

H096 the syntax of OBJECT be checked :
const , where OBJECT is constant ;
type , where OBJECT is type ;
var , where OBJECT is variable.

H097 uminus operation : =.

H098 uplus operation : ±.

H099 syntax of unsigned integer DIGIT DIGITSETY is correct :
DIGIT symbol \symbol{94} syntax of unsigned integer DIGITSETY is correct \symbol{99}
where DIGITSETY is EMPTY ; DIGIT symbol.\symbol{94}

H100 syntax of unsigned real DIGITS dot DISITS1 EXPONETY is correct :
syntax of unsigned integer DIGITS is correct \symbol{99}

syntax of unsigned integer DIGITS1 is correct \symbol{99}
syntax of exponent part EXPONETY is correct \symbol{102}
where EXPONETY is EMPTY ; syntax of unsigned integer DIGITS is correct \symbol{99}

syntax of unsigned integer DIGITS1 is correct.\symbol{99}

H101 syntax of unsigned real DIGITS EXPON is correct :
syntax of unsigned integer DIGITS is correct \symbol{99}
syntax of exponent part EXPON is correct.\symbol{102}

H102 syntax of exponent part e.SIGNETY DIGITS is correct :
E , SIGNETY symbol \symbol{94} syntax of unsigned integer DIGITS is correct \symbol{99}
where SIGNETY is EMPTY ; E , syntax of unsigned integer DIGITS is correct.\symbol{99}
H103 SHAPE sort is of four types:
  \texttt{var}, where SHAPE is variable;
  \texttt{function}, where SHAPE is function;
  \texttt{procedure}, where SHAPE is procedure;
  \texttt{value}, where SHAPE is value.

H104 \texttt{times} binary multiplication operator: * symbol.

H105 \texttt{realdiv} binary multiplication operator: / symbol.

H106 \texttt{intdiv} binary multiplication operator: \texttt{div}.

H107 \texttt{mod} binary multiplication operator: \texttt{mod}.

H108 \texttt{and} binary multiplication operator: \texttt{and}.

H109 \texttt{plus} binary adding operator: + symbol.

H110 \texttt{minus} binary adding operator: - symbol.

H111 \texttt{or} binary adding operator: \texttt{or}.

H112 \texttt{in} binary relational operator: \texttt{in}.

H113 \texttt{equal} to binary relational operator: \texttt{=}.

H114 \texttt{unequal} to binary relational operator: \texttt{\neq}.

H115 \texttt{less} than binary relational operator: \texttt{<}.

H116 \texttt{greater} than binary relational operator: \texttt{>}.

H117 \texttt{less or equal} to binary relational operator: \texttt{\leq}.

H118 \texttt{greater or equal} to binary relational operator: \texttt{\geq}.
Primitive rules

H001 where CHARSETY is CHARSETY :
   true. 

H002 true : EMPTY.

H003 where CHARSETS is not in CHARSETS :
   the string consist of ONES characters CHARSETS , @pr4
   ONES of CHARSETS can not be in CHARSETS @pr5
   the string consist of ONES characters CHARSETS @pr4
   CHARSETS consist of less than ONES characters. @pr6

H004 the string consist of one ONESETY characters CHAR
   CHARSETS :
   the string consist of ONESETY characters CHARSETS @pr3
   where ONESETY CHARSETS is EMPTY. @pr1

H005 ONES of CHARSETS can not be in CHAR CHARSETS CHARSETS1 :
   the string consist of ONES characters CHAR
   CHARSETS @pr4
   where CHARATS differs from CHAR CHARSETS @pr8
   ONES of CHARATS can not be in CHARSETS CHARSETS1 @pr5
   where CHARSETS1 is EMPTY @pr1
   the string consist of ONES characters CHAR
   CHARSETS @pr4
   where CHARATS differs from CHAR CHARSETS. @pr8

H006 CHARSETS consist of less than ONES characters :
   the string consist of ONES1 characters CHARSETS @pr4
   where ONES contains ONES1 one @pr7
   where CHARSETS is EMPTY. @pr1

H007 where CHARSETS CHARSETS1 CHARSETS2 contains CHARSETS1 :
   true. @pr2

H008 where CHARSETS CHAR differs from CHARSETS1 CHAR1 :
   where CHARSETS differs from CHARSETS1 @pr8
   where CHAR preceds CHAR1 in ALLCHARACTER @pr9
   where CHAR1 preceds CHAR in ALLCHARACTER. @pr9

H009 where CHAR preceds CHAR1 in CHARSETS CHAR CHARSETS1 CHAR1
   CHARSETS2 :
   true. @pr2

H010 whenever the CHARATS be not the empty : true. @pr2
APPENDIX 2

FORMAL DEFINITION OF PSS USING THE CARABINER SYSTEM
CPS rules for PSS

001 (program) ::= (program heading) loc(#BPTR); link(#BPTR, k \_ of STORE); loc(\_); at(\_); trans(loc(\_)); link(\_ of STORE, \_ of STORE); at(\_); link(\_ of ATTRIB, \_ of STORE)

(block)
trans(trim(ATTRIB, \_));

trim(STORE, \_);
trimr(ATTRIB, \_); trimr(STORE, \_); delr(STORE, #BPTR);
aw(1 k(\_ of STORE))

002 (program heading) ::= program (identifier) idlimit(\_)

003 (identifier) ::= (letter)(letter or digit)

004 (letter or digit) ::= (letter)(digit)

005 (block) ::= (constant definition part) (type definition part) (variable declaration part) (subroutine declaration part) (statement part)

006 (constant definition part) ::= (empty)
const (constant definition)(constant definition)
007 (constant definition)::=
  (new identifier) =
    \( \text{loc}(\text{STEMP}) \)

008 (new identifier)::=
  (identifier)

009 (constant)::=
  (signed const) | (unsigned const)

010 (signed const)::=
  (sign) (unsigned number)
    10.1 % numconst((), STEMPE)
    10.1
  (sign) (identifier)
    10.2 (identifer)
    10.3
  -rconstid((),)

011 (sign)::=
  +/-

012 (unsigned number)::=
  (unsigned integer)
    12.1 % int((),)

013 (unsigned real)::=
  (unsigned real)
    12.2 % real((),)
013 \texttt{unsigned integer} ::= \\
\quad \langle\text{digit}\rangle \langle\text{digit}\rangle \\

014 \texttt{unsigned real} ::= \\
\quad \langle\text{unsigned integer}\rangle . \langle\text{digit}\rangle \langle\text{digit}\rangle \\
\quad \langle\text{unsigned integer}\rangle . \langle\text{digit}\rangle \langle\text{digit}\rangle \ E \langle\text{scalar factor}\rangle \\
\quad \langle\text{unsigned integer}\rangle \ E \langle\text{scalar factor}\rangle \\

015 \texttt{scalar factor} ::= \\
\quad \langle\text{unsigned integer}\rangle \langle\text{sign}\rangle \langle\text{unsigned integer}\rangle \\

016 \texttt{unsigned const} ::= \\
\quad \langle\text{identifier}\rangle \\
\quad 16.1
\quad \langle\text{constid}()\rangle \} \\
\quad 16.1 \\
\quad \langle\text{other unsigned constant}\rangle \\

017 \texttt{other unsigned constant} ::= \\
\quad \langle\text{unsigned number}\rangle \langle\text{character}\rangle \langle\text{string}\rangle \\

018 \texttt{character} ::= \\
\quad ' \langle\text{char}\rangle \\
\quad 18.1 \\
\quad \langle\text{kar}()\rangle \} \\
\quad 18.1 \\

019 \texttt{string} ::= \\
\quad ' \langle\text{char}\rangle \langle\text{char}\rangle \\
\quad 19.1 \quad 19.2 \\
\quad \langle\text{twochar}()\rangle \langle()\rangle \} \\
\quad 19.1 \quad 19.2 \\
\quad \langle\text{augl}(\text{STORE} \# \text{WIDTH})\rangle \\
\quad 19.1 \quad 19.2 \\
\quad \langle\text{store}(\# \text{WIDTH})\rangle \\
\quad 19.1 \\
\quad \langle\text{del}(\text{STORE} \# \text{WIDTH})\rangle \\
\quad 19.1 \\
\quad \langle\text{char}\rangle \langle\text{morechar}()\rangle \} \\
\quad 19.3 \quad 19.3 \\
\quad \langle\text{augr}(\text{WIDTH} \# \text{T of ATTRIB} \# \text{STORE})\rangle \\
\quad 19.3 \\
\quad \langle\text{store}(\# \text{WIDTH})\rangle \\
\quad 19.3 \\
\quad \langle\text{del}(\text{STORE} \# \text{WIDTH})\rangle \\
\quad 19.3 \\

020 \texttt{type definition part} ::= \\
\quad \langle\text{empty}\rangle \\
\quad \langle\text{type} \langle\text{type definition}\rangle \langle\text{type definition}\rangle\rangle \\
\quad \langle\text{ptridchk}\rangle \\
\quad \langle\text{translate types}\rangle \\
\quad \langle\text{empty}\rangle
021 (type definition)::=
  (new identifier) = (type)
  21.1
  \% if e(k k (#T of ATTRIB) \cap TYPE) eq type identifier
  then
    si #T of ATTRIB, ()
  \% else
    at() is 1(#T of ATTRIB));
    21.1
    at(#ATEMP);
    link(#ATEMP, #T of ATTRIB);
    at(#ATEMP);
    link(#ATEMP, () of ATTRIB);
    21.1
    transfer(prev(#ATEMP), #ATEMP);
    trimr(ATTRIB, prev(#ATEMP));
    loc(#STEMP);
    link(#STEMP, next(#STEMP#STEMP));
    while k #STEMP ineq \O do
      2
      (if k k #STEMP#STEMP#STEMP, ATTRIB eq #T of ATTRIB
      then
        2
        link(); of ATTRIB, k #STEMP);
        21.1
        forward(#STEMP)
      ));
    delr(ATTRIB, #STEMP);
    del(ATTRIB, #T)
  \%}

022 (type)::=
  \% at(#T, #ATEMP));
  \% link(#ATEMP, #T of ATTRIB)
  \%
  \% {type1}
  \% {delr(ATTRIB, #ATEMP)}

023 (type1)::=
  \% {simple type}; {structured type}; {pointer type}

024 (simple type)::=
  \% {scalar type}; {subrange type}; {type identifier}

025 (scalar type)::=
  \% {link(scalar of TYPE, k #ATEMP);
  k #ATEMP is (DEF is (), WIDTH is (1));
  adjint#ATEMP)}
  \% {scalar identifier}; \% {scalar identifier}
{scalar identifier} ::= 
(new identifier) 

\[ \text{augr}(k \ \#ATEMP,()\)\] 
\[ \text{loc}()\] 
\[ \text{link}(\text{constant of STORE},()\) \text{ of STORE);}\] 
\[ \text{sl } k(() \) \text{ of STORE},()\] 
\[ \text{link}(k \ \#ATEMP,k(() \) \text{ of STORE});\] 

\begin{align*}
\{\text{subrange}\} & ::= \\
& \text{loc} (#STEMP); \\
& (\text{constant}) \\
& \text{if } k \ \#STEMP \cap \text{ TYPE ineq scalar} \\
& \text{then halt;} \\
& \text{loc} (#STEMP); \\
& (\text{constant}) \\
& \text{if } k \ \#STEMP \cap \text{ TYPE ineq scalar} \\
& \text{then halt;} \\
& \text{if } k \ (\text{prev} (#STEMP) \ \text{eq } k \ \#STEMP) \\
& \text{then} \\
& \text{if } () \ \text{eq } \text{ copyl (DEF of } k \ \#STEMP,()) \\
& \text{then} \\
& k \ \#ATEMP \ \text{is } (\text{DEF is } \text{ copyr (DEF of } k \ \#STEMP, \\
& \text{prev} (()) ) \\
& \text{trimr (DEF of } k \ \#ATEMP,\text{next} (()) )\}; \\
& \text{augr} (k \ \#ATEMP,\text{WIDTH is } (1))\}; \\
& \text{link} (\text{subrange of TYPE}, k \ \#ATEMP) \\
& \text{else halt} \\
& \text{else halt;} \\
& \text{trimr} (\text{STORE,prev} (#STEMP)) \\
\end{align*}
(type identifier) ::= (identifier)

28.1

if () then

link(k #ATEMP, () of ATTRIB);

link(type identifier of ATTRIB, k #ATEMP)

else halt


(structured type) ::= (array type)!(record type)! (set type)

(array type) ::= array

k #ATEMP is ();

link(array of TYPE, k #ATEMP);

augl(STORE, #DIMEN);

si k(#DIMEN lm STORE), 0;

augl(STORE, #MULBND);

si k(#MULBND lm STORE), +1

(index type)(); (index type))

augl(k #ATEMP, #DIMEN is (e k(#DIMEN lm STORE)));

dell(STORE, #DIMEN);

augr(k #ATEMP, #T);

at(#ATEMP);

link(#ATEMP, k prev(#ATEMP))

(explicit)(#ATEMP);

si k(#MULBND lm STORE), 2

e < (e k(#MULBND lm STORE), i* 2

e < (+$1 of WIDTH of k #ATEMP)

con )

delr(ATTRIB, #ATEMP);

augr(k #ATEMP, 2

WIDTH is (e < (+$1 k(#MULBND lm STORE))

sep )

dell(STORE, #MULBND)
031 \text{(index type)}::=
\begin{align*}
\text{i} & \triangleright (\text{DIMEN \_m} \text{ STORE}), e \triangleright (e \triangleright (\text{DIMEN \_m} \text{ STORE})); \\
& \text{ augr}(k \text{ \_ATEMP};\#T); \\
& \text{ at}(\text{\_ATEMP}); \\
& \text{ link}(\text{\_ATEMP};\#T \text{ of } k \text{ prev}(\text{\_ATEMP})); \\
& \text{ at}(\$); \\
& \text{ loc}(\$). \\
\end{align*}

\text{ (simple type) }:
\begin{align*}
& \text{ trimr}(\text{ATTRIB};\$); \\
& \text{ trimr}(\text{STORE};\$); \\
& \text{ explicit}(\text{\_ATEMP}); \\
& \text{ if } e(k \text{ \_ATEMP \_m} \text{ TYPE}) \in \{\text{scalar, subrange}\} \text{ then } \\
& \text{ if } k \text{ \_ATEMP \_m} \text{ ineq } \text{ real \_m} \text{ ATTRIB } \text{ then } \\
& \text{ adjin}(\text{\_ATEMP}); \\
& (\text{i} (\text{k \_MULBND \_m} \text{ STORE}), \text{2}; \\
& e \triangleright (e \triangleright (\text{k \_MULBND \_m} \text{ STORE})); \\
& e \triangleright (+\text{ ordinal}((e \& k \text{ \_ATEMP} \text{ of } \text{con} \text{ k \_ATEMP}))) \\
& \text{ else halt } \\
& \text{ else halt } \\
& \text{ delr}(\text{ATTRIB};\text{\_ATEMP}).
\end{align*}
032 (record type)::=
  record
  \( k \#\text{ATEMP} = () \);  
  link(record of TYPE, k \#\text{ATEMP});
  augl(STORE, (#\text{FIXWIDTH}, #\text{VARWIDTH}));
  si l k(#\text{FIXWIDTH} \lor STORE), +01;
  si l k #\text{VARWIDTH} \lor STORE, +01
\)
  \{ field list \}
  \{ augr(k \#\text{ATEMP}, 2
    WIDTH is (e, d, (+,
    sep
    2
    e, d (e k(#\text{VARWIDTH} \lor STORE),
    i+e k(#\text{FIXWIDTH} \lor STORE)
    )
  \}
  dell(STORE, #\text{WIDTH})
  dell(STORE, #\text{FIXWIDTH})
\)

033 (field list)::=
  \{ augr(k \#\text{ATEMP}, F1)\}
  \{ field list1 \}
  \{ si F1 of k \#\text{ATEMP}, #F1
    augr(k \#\text{ATEMP}, #EF)\}
\}

034 (field list1)::=
  \{ fixed part \}
  \{ augr(k \#\text{ATEMP}, (#\text{VP}, #\text{EVP}))\}
  \{ fixed part \}
  \{ augr(k \#\text{ATEMP}, #\text{VP1})\}
  \{ variant part \}
  \{ si \text{VP1 of } k \#\text{ATEMP}, #\text{VP1}
    augr(k \#\text{ATEMP}, #\text{EVP})\}
\}
  \{ augr(k \#\text{ATEMP}, #\text{VP1})\}
  \{ variant part \}
  \{ si \text{VP1 of } k \#\text{ATEMP}, #\text{VP1}
    augr(k \#\text{ATEMP}, #\text{EVP})\}
\}

035 (fixed part)::=
  \{ record section \}\{ (record section)\}
036 {record section}::=  
    <identifier>  
      36.1  
        { fldid(()) *ATEMP);  
          36.1  
            at(#ATEMP);  
            link(#ATEMP,()) of k prev(#ATEMP))  
          36.1  
        }  
      }  
    }  
    <identifier>  
      36.2  
        { fldid((() prev(#ATEMP)));  
          36.2  
            ;  
            augr(prev(#ATEMP),#T);  
            at(#ATEMP);  
            link(#ATEMP,#T of k prev(prev(#ATEMP)))  
          ;  
          (type)  
          }  
        endsec;  
        trimr(ATTRIB,prev(#ATEMP))  
      ;  

037 {variant part}::=  
    case  
        { augr(k #ATEMP,#CASE);  
          at(#ATEMP)  
        }  
    { tag field}  
        { augr(k prev(#ATEMP),#T);  
          at(#ATEMP);  
          link(#ATEMP,#T of k prev(prev(#ATEMP)))  
        }  
    { type identifier}  
        { endsec;  
          trimr(ATTRIB,prev(#ATEMP))  
        }  
    of  
        { augr(k #ATEMP,#SWITCH);  
          augl(#ATTRIB,#TAGTYPE);  
          link(#TAGTYPE lm ATTRIB,#T of k #ATEMP);  
          explicit(#TAGTYPE lm ATTRIB);  
          augl(STORE,#LABEL);  
          sl k(#LABEL lm STORE),M is ()  
        }  
    {variant}{}{variant}  
        { dell(STORE,#LABEL);  
          dell(ATTRIB,#TAGTYPE)  
        }  
    ;
038 \(\text{tag field} \equiv\)
\[\text{identifier}\]
\[\text{feldid()}\]
\[\text{link(#ATEMP,()) of k prev(#ATEMP)}\]
\[\text{augr(k prev(#ATEMP),(#EXTRA,#OFFSET is ()})\]
\[\text{(empty)}\]
\[\text{augr(k prev(#ATEMP),(#EXTRA,#OFFSET is ()});\]
\[\text{link(#ATEMP,#EXTRA of k prev(#ATEMP))}\]

039 \(\text{variant} \equiv\)
\[\text{augr(k #ATEMP,#V1)}\]
\[\text{at(#ATEMP)}\]
\[\text{link(#ATEMP,#T of k prev(#ATEMP))}\]
\[\text{explicit(#ATEMP)}\]
\[\text{loc(#STEMP)}\]
\[\text{(case label list)}:\]
\[\text{delr(ATTRIB,#ATEMP)}\]
\[\text{augr(k #ATEMP,# STEMPEmp)}\]
\[\text{casecompare(#LABEL lm STORE,#STEMPEmp)}\]
\[\text{delr(STORE,#STEMPEmp)}\]
\[\text{augl(STORE,(#FIXWIDTH,#VARWIDTH))}\]
\[\text{sl k(#FIXWIDTH lm STORE)}\]
\[\text{sl k(#VARWIDTH lm STORE)}\]
\[\text{(field list)}\]
\[\text{maxwidth(#VARWIDTH lm STORE,#FIXWIDTH lm STORE,}\]
\[\text{next(#FIXWIDTH lm STORE)}\]
\[\text{delr(STORE,#VARWIDTH)}\]
\[\text{delr(STORE,#FIXWIDTH)}\]
\[\text{sl #V1 of k #ATEMP,#V1}\]
\[\text{augr(-#ATEMP,#EV)}\]
\[\text{(empty)}\]

040 \(\text{case label list} \equiv\)
\[\text{s1 k #STEMPEmp,#LABEL is ()}\]
\(\text{(case label}{, (case label)}\)

041 \(\text{case label} \equiv\)
\[\text{loc(#STEMPEmp)}\]
\(\text{constant}\)
\[\text{caseitem}\]
\[\text{delr(STORE,#STEMPEmp)}\]
042 \text{(set type) ::=}
\text{\textbf{set of}}
  \text{\{k #ATEMP is (#T,WIDTH is (1))\}}
  \text{link(set of TYPE,k #ATEMP);}
  \text{at(#ATEMP);}
  \text{link(#ATEMP,#T of k prev(#ATEMP))}
\}
\text{\{simple type\}}
\text{\textbf{explicit(#ATEMP);}}
\text{\textbf{if \text{\{k #ATEMP \& TYPE\}} \notin \{scalar, subrange\}\ then\ halt;}}
\text{\textbf{delr(ATTRIB,#ATEMP)}}
\}

043 \text{(pointer type) ::=}
\text{\{k #ATEMP is (#T,WIDTH is (1))\}}
\text{link(pointer of TYPE,k #ATEMP);}
\text{at(#ATEMP);}
\text{link(#ATEMP,#T of k prev(#ATEMP))}
\}
\text{\{ptr identifier\}}
\text{\textbf{delr(ATTRIB,#ATEMP)}}
\}

044 \text{(ptr identifier) ::=}
\text{\{identifier\}}
\text{44.1}
\text{\textbf{link(type identifier of TYPE,k #ATEMP) \ if \ () \ \& \ n \ \sub attrib \ then \}}
\text{44.1}
\text{\textbf{link(k #ATEMP,() of ATTRIB)}}
\text{44.1}
\text{\textbf{else \ augl(ATTRIB,#TMPPTR is ())\ ; \ 44.1}}
\text{\textbf{link(k #ATEMP,#TEMPPTR of ATTRIB)}}
\}

045 \text{(variable declaration par) ::=}
  \text{\{empty\}}
  \text{var \ (variable declaration)\{\{variable declaration\}\}}
046 (variable declaration)::=
  \( \text{augl(STORE,#MEM); loc(STEMP);} \)
  \( \text{trans(loc(STEMP))} \)
  
  \( \text{loc(LISTEMP); trans(loc(LISTEMP))} \)
  
  \( \text{(new identifier)} \)
  
  \( \text{firstvar((),#STEMP);} \)
  
  \( \text{morevar((),()}; \)
  
  \( \text{(type)} \)
  
  \( \text{single typetrans;} \)
  \( \text{linktovars(#STEMP,#MEM);} \)
  \( \text{trans(delr(STORE,#STEMP));} \)
  \( \text{delr(STORE,#STEMP);} \)
  \( \text{delr(STORE,#MEM);} \)

047 (subroutine declaration part)::=
  \( \text{augl(STORE,#FORWARD);} \)
  
  \( \text{(subroutine declaration);} \)
  
  \( \text{if k(#FORWARD in STORE) eq 0} \)
  
  \( \text{then delr(STORE,#FORWARD)} \)
  \( \text{else halt} \)


048 (subroutine declaration)::=
  { loc(#STEMP)}
  (subroutine heading) forward
  \[ if \#AHEAD \neq n \mid k \#STEMP \]
  then
    \[ if \#AHEAD \neq \#STEMP \]
    \[ augr(k \#STEMP; \#AHEAD); \]
    \[ link(#FORWARD in STORE, k \#STEMP) \]
  else
    \[ if \#AHEAD \neq \#STEMP \]
    \[ link(#BPTR of STORE, k \#STEMP); \]
    \[ loc(#STEMP) \]
    \[ if \#AHEAD \neq \#STEMP \]
    \[ augr(k \#STEMP; \#AHEAD); \]
    \[ link(#FORWARD in STORE, k \#STEMP) \]
    \[ else \]
    \[ delr(STORE, \#STEMP) \]

049 (subroutine heading)::=
  <procedure heading>; <function heading>
050 (procedure heading)::=
   procedure (identifier) ! 50.1
   if () then
      if () then
         oldsubname() , procedure) 50.1
         else
            newsubname() , procedure) 50.1
      else
         procedure (identifier) 50.2
         if () then
            newsubname() , procedure) 50.2
            else halt
            (formal parameter section)
   {;} {formal parameter section}

051 (formal parameter section)::=
   insert (STORE, #STEMP, #STEMP)!
   (formal parameter section1)
   delr (STORE, #STEMP)!

052 (formal parameter section1)::=
   (value formal parameters)!
   (variable formal parameters)!
   (function formal parameters)!
   (procedure formal parameters)

053 (value formal parameters)::=
   (val var par group)
   if parkind(formal value)!
   (par-type)

054 (val var par group)::=
   (val var par id)
   if link(#STEMP, () of #prev(#STEMP))!
   {, (val var par id)}
055 (val var par id)::=
  (identifier)
  idlimit()
  if ()
  then
  else
hilt

056 (par type)::=
  (identifier)
  if ()
  then
  repeat
  link()
  until
else
hilt

057 (variable formal parameters)::=
  var (val var par group)
  parkind(formal variable)
  (par type)

058 (function formal parameters)::=
  function (fun proc par group)
  parkind(formal function)
  (par type)

059 (fun proc par group)::=
  (fun proc par id)
  link(#STEMP,() of k prev(#STEMP))

060  \(\text{fun proc par id} :=\)
    \(\text{(identifier)}\)
    \(\text{60.1}\)
    \(\text{idlimit()}\)
    \(\text{60.1}\)
    \(\text{if () } \in 1 k \text{ prev(#STEMP)}\)
    \(\text{60.1}\)
    \(\text{then}\)
    \(\text{augr(k prev(#STEMP),{}) is (#LATER));}\)
    \(\text{60.1}\)
    \(\text{link(} of \text{ATTRIB,#LATER of {}} of k \text{prev(#STEMP))}\)
    \(\text{60.1}\)
    \(\text{else halt}\)
\)

061  \(\text{procedure formal parameters} :=\)
    \(\text{procedure (fun proc par id)}\)
    \(\text{parkind(formal procedure)}\)
\)

062  \(\text{function heading} :=\)
    \(\text{function (identifier)}\)
    \(\text{62.1}\)
    \(\text{if () } \in \text{copyr(STORE,)}\)
    \(\text{62.1}\)
    \(\text{then}\)
    \(\text{newsurname{}} , \text{function}\)
    \(\text{62.1}\)
    \(\text{else halt}\)
\(\text{62.1}\)
\(\text{func type} :=\)
    \(\text{function (identifier)}\)
    \(\text{62.2}\)
    \(\text{if () } \in \text{copyr(STORE,)}\)
    \(\text{62.2}\)
    \(\text{then}\)
    \(\text{newsurname{}} , \text{function}\)
    \(\text{62.2}\)
    \(\text{else halt}\)
\(\text{62.2}\)
\(\text{formal parameter section} :=\)
\(\text{if(formal parameter section)}\)
\(\text{62.3}\)
\(\text{func type} :=\)
    \(\text{function (identifier)}\)
    \(\text{62.3}\)
    \(\text{if () } \in \text{copyr(STORE,)}\)
    \(\text{62.3}\)
    \(\text{then}\)
    \(\text{oldsubname{}} , \text{procedure}\)
    \(\text{62.3}\)
    \(\text{else halt}\)
\)
063 (func type)::=
  (identifier)

  {if () £ n 1 ATTRIB
   then link(()) of ATTRIB,k #STEMP)
   else halt

064 (statement part)::=
  (compound statement)

065 (statement)::=
  (simple statement) | (structured statement)

066 (simple statement)::=
  (assignment statement) | (procedure statement) | (empty statement)

067 (assignment statement)::=
  l at(#ATEMP);
  trans(loc(#STEMP))

  {variable}::=
  l at(#ATEMP);
  trans(loc(#STEMP))

  {expression}
  l typecheck(prev(#ATEMP),#ATEMP);
  trim(ATRIB,prev(#ATEMP));
  trans(assign(prev(#STEMP),#STEMP));
  trimr(STORE,prev(#STEMP))

068 (lvariable)::=
  (identifier)

  {clientvar() ,#ATEMP};

  {forstmtcheck() };

  trans(clientvar(); #STEMP))

069 (other lvariable)::=
  {lcomponent variable} | {lreferenced variable}
070 \text{(lcomponent variable)} ::= \text{(indexed variable)} \mid \text{(lfield variable)}

071 \text{(indexed variable)} ::= \text{(lnonfuncvar)}
\begin{itemize}
\item[\&] \text{array(blog)};
\item[\&] \text{augl(STORE,}@\text{DIMEN)};
\item[\&] \text{trans(array(#STEMP)};
\item[\&] \text{augl(STORE,#SUM)};
\item \text{sl} _{k(#SUM \text{ in STORE}),+0}
\end{itemize}

\text{indices}

\text{correctindex};
\begin{itemize}
\item[\&] \text{forward(blog)};
\item[\&] \text{dell(STORE,#DIMEN)};
\item[\&] \text{trans(forward(#STEMP)};
\item[\&] \text{sliceunit(#STEMP)};
\item[\&] \text{dell(STORE,#SUM)}
\end{itemize}

\text{indices}

072 \text{(lnonfuncvar)} ::= \text{(entire variable)} \mid \text{(other lvariable)}

073 \text{(entire variable)} ::= \text{(identifier)}
\begin{itemize}
\item[\&] \text{creatvar()} ,@\text{ATEMP)};
\item[\&] \text{trans(creatvar()} ,@\text{STEMP)}
\end{itemize}

074 \text{(indices)} ::= \begin{itemize}
\item[\&] \text{moreindex};
\item[\&] \text{index expression} \mid \text{moreindex} \mid \text{index expression}
\end{itemize}
075 (index expression) ::= 
forward(#ATEMP); 
trans(rforward(#STEMP)); 
loc(); indexaction; 
trans(insertr(STORE,#STEMP,)); 
link(of STORE, of STORE); indexaction(#STEMP); 
at(#ATEMP); trans(loc(#STEMP)) 
(expression) 
indextype(prev(#ATEMP),#ATEMP); 
trans(moresum(prev(#STEMP),#STEMP)); 
delr(STORE,#STEMP) 
); delr(ATRIB,#ATEMP); 
trans(trimr(STORE,#STEMP)); 
trimr(STORE,#STEMP)) 

076 (1field designator) ::= 
(nonfuncvar). 
crecord(#ATEMP); 
trans(rcrecord(#STEMP)) 
(identifier) 
76.1 
cselector(),#ATEMP); 
trans(rselector(),#STEMP) 76.1 

077 (1referenced variable) ::= 
(nonfuncvar)↑ 
cpointer(#ATEMP); 
trans(rpointer(#STEMP))
078 (expression):=
  (simple expression); at(#ATEMP);
  trans(loc(#STEMP))
 /
  (simple expression)(relational operator)
  at(#ATEMP);
  trans(loc(#STEMP))
 /
  (simple expression)
  ccompexp(prev(#ATEMP),{},#ATEMP);
  trimr(ATRIB,prev(#ATEMP));
  link(#ATEMP,Boolean lm ATRIB);
  trans(rcompexp(prev(prev(#STEMP)),prev(#STEMP),
    {} ,#STEMP
  ) ,#STEMP
  );
  trimr(STORE,prev(#STEMP));
  link(Boolean lm ATRIB,#STEMP)
 /

\[ \text{simple expression} ::= \text{term} \]

\[ \text{term} ::= \text{sign} \]

\[ \text{sign} ::= \text{at}(\#\text{ATEMP}); \]
\[ \text{trans}(\text{loc}(\#\text{ATEMP})) \]

\[ \text{term} ::= \text{csign}(\text{prev}(\text{#ATEMP}),\#\text{ATEMP}); \]
\[ \text{delr}(\text{ATTRIB},\#\text{ATEMP}); \]
\[ \text{trans}(\text{rsign}(\text{prev}(\#\text{STEMP}),(),\#\text{STEMP}); \]
\[ \text{delr}(\text{STORE},\#\text{STEMP}) \]

\[ \text{term} ::= \text{at}(\#\text{ATEMP}); \]
\[ \text{trans}(\text{loc}(\#\text{STEMP})) \]

\[ \text{simple expression} \text{ adding operator} \]

\[ \text{term} ::= \text{additional}(\text{prev}(\text{prev}(\#\text{ATEMP})),\text{prev}(\#\text{ATEMP}),() \]
\[ \text{delr}(\text{ATTRIB},\#\text{ATEMP}); \]
\[ \text{trans}(\text{additional}(\text{prev}(\text{prev}(\#\text{STEMP})),\text{prev}(\#\text{STEMP})); \]
\[ \text{trimr}(\text{ATTRIB},\text{prev}(\#\text{ATEMP})); \]
\[ \text{trans}(\text{additional}(\text{prev}(\text{prev}(\#\text{STEMP})),\text{prev}(\#\text{STEMP})); \]
\[ \text{trimr}(\text{TRIMR},\text{prev}(\#\text{STEMP}) \]

\[ \text{term} \]
080 (term)::=  
   (factor);  
   (term) (multiplying operator) 

081 (factor)::=  
   (expression);  
   not 
   (factor) 
   (factor1) 

082 (factor1)::=  
   (identifier) 
   (other variable); (other cons in exp); (set); (par func designator)

083 (other variable)::=  
   (component variable); (referenced variable) 

084 (component variable)::=  
   (indexed variable); (field designator)
085 <indexed variable>::=
  <variable>
  [ [ caarray(#ATEMP);
    augl(STORE,#DIMEN);
    trans(rarray(#STEMP));
    augl(STORE,#SUM);
    +1 k(#SUM 1m STORE),+0 ]
  ]
  <indices>

  ]
  <correct index;
  forward(#ATEMP);
  dell(STORE,#DIMEN);
  trans(rforward(#STEMP));
  sliceunit(#STEMP);
  dell(STORE,#SUM)
  ]

086 <variable>::=
  <entire variable>!<other variable>

087 <field designator>::=
  <variable>.
  [ crecord(#ATEMP);
    trans(rrecord(#STEMP))
  ]
  <identifier>
  87.1
  cselector(() ,#ATEMP);
  87.1
  trans(rrselector(() ,#STEMP))
  87.1

088 <referenced variable>::=
  <variable>↑
  [ cpointer(#ATEMP);
    trans(rpointer(#STEMP))
  ]
089 \texttt{(other const in exp)}::=
  \texttt{\{ loc(#STEMP) \} \texttt{(other unsigned constant)}}
  -1
  \texttt{\{ link(#ATEMP,k k #STEMP\texttt{\_ATTRIB)}}; 
  \texttt{trans(s1 \_k \_k \_STEMP\._e \_k \#STEMP)}; 
  \texttt{delr(STORE,#STEMP)}; 
  \texttt{constaction \} \texttt{\} .}

090 \texttt{(set)}::=
  \texttt{\{ \{ trans(s1 \_k \_STEMP\_\{ \}) \}\} \} 
  \texttt{(element list)}
091 (element list):=
  \{ insert1(ATTRIB, #ATEMP, #T is (WIDTH is (1))); link(set of TYPE, #T of ATTRIB); link(#ATEMP, #T of ATTRIB); at(#ATEMP); trans(loc(#STEMP)) \}

\{element\}
\{ insert1(#T of ATTRIB, #T is #1 k #ATEMP); link(set of TYPE, #T of #T of ATTRIB); delr(ATTRIB, #ATEMP); trans(at(#1 k #ATEMP)); link(set of TYPE, #T of ATTRIB); link(scalar of TYPE, #T of #T of ATTRIB); link(#T of ATTRIB, prev(#STEMP)); \}
\{ element\}
\{ insert1(#1 k #ATEMP); link(set of TYPE, #T of ATTRIB); link(scalar of TYPE, #T of #T of ATTRIB); link(#T of ATTRIB, prev(#STEMP)); \}
\{ element\}
\{ insert1(#1 k #STEMP); \}
\{ insert1(#STEMP); \}
\{ element\}
\{ insert1(#STEMP); \}
\{ element\}
\{ insert1(#STEMP); delr(STORE, #STEMP) \}

\{ element\}
\{ insert1(#STEMP); delr(STORE, #STEMP) \}

\{ element\}
\{ insert1(#STEMP); delr(STORE, #STEMP) \}
092 \{element\}::= \\
\{set exp\}
  \{trans\(\text{sl} \ k \ #\text{TEMP}, (e_k \ #\text{TEMP})\)\};

\{set exp\}.. \\
\{at\(#\text{ATEMP}\);
  trans(\text{loc}(\#\text{STEMP}))
\} \\
\{set exp\}
  \{scalarorsubrangetypeidentical\(\text{prev}(\#\text{ATEMP}, \#\text{ATEMP})\); \\
  \text{delr}(\text{ATTRIB}, \#\text{ATEMP})\}; \\
trans(\text{setrange}(\text{prev}(\#\text{STEMP}, \#\text{STEMP})\}; \\
  \text{delr}(\text{STORE}, \#\text{STEMP}) \}

093 \{set exp\}::= \\
\{expression\} \\
\{explicit\(#\text{ATEMP}\); \\
  \if e(\(\text{k} \ k \ #\text{ATEMP} \land \text{TYPE}) \notin \{\text{scalar, subrange}\} \\
  \text{then} \text{halt} \}

094 \{par func designator\}::= \\
\{identifier\} \\
  \{\text{loc}(\#\text{STEMP}); \\
  \text{fsub}()\} \\
  \{\text{actual pars}\} \\
  \{\text{if} \ #\text{LATER} \in n \ l \ k \ #\text{STEMP} \\
  \text{then} \\text{delr}(\k \ #\text{STEMP}, \#\text{LATER})\}; \\
  \{\text{if} l \ k \ #\text{STEMP} \text{eq} () \\
  \text{then} \\text{augr}(\k \ #\text{STEMP}, \#\text{LATER})\}; \\
  \}\text{delr}(\text{STORE}, \#\text{STEMP}); \\
\text{trans}(\text{call})
095 \( \text{actual parameter} := \)
\( \text{at}(\text{#ATEMP}); \)
\( \text{loc}(\text{#STEMP}); \)
\( \text{trans}(\text{loc}(\text{#STEMP})); \)
\( \text{firstpar} \)
\( \text{actual parameter} \)
\( \{ \text{morepar} \} \)
\( \text{actual parameter} \)
\( \text{trans} \left( \text{delr}(\text{STORE}, \text{#STEMP}) \right) ; \)
\( \text{delr}(\text{STORE}, \text{#STEMP}); \)
\( \text{delr}(\text{ATTRIB}, \text{#ATEMP}); \)

096 \( \text{actual parameter} := \)
\( \text{at}(\text{#ATEMP}); \)
\( \text{trans}(\text{loc}(\text{#STEMP})); \)
\( \text{actual parameter} \)
\( \text{trans} \left( \text{standcall}(\text{prev}(\text{prev}(\text{#STEMP}) \text{, #STEMP})); \right) \)
\( \text{delr}(\text{STORE}, \text{#STEMP}); \)
\( \text{delr}(\text{ATTRIB}, \text{#ATEMP}); \)

097 \( \text{actual parameter} := \) (identifier)
\( 97.1 \)
\( \text{if} () \in \text{STORE} 97.1 \)
\( \text{then} \)
\( \text{idpar} () 97.1 \)
\( \text{else} \text{halt} \)
\( \text{other variable} \text{valvarpar} \)
\( \text{expression} \text{valvarpar} \)
098 (procedure.statement)::=
  \(\text{at}(@\text{TEMP})\)
  \(\text{loc}(@\text{TEMP})\)
  \(\text{trans}(\text{loc}(\text{TEMP}))\)

\(\langle\text{procedure identifier}\rangle\langle\text{might empty par proc}\rangle\)
\(\langle\text{if} \ #\text{LATER} \ \& \ # \ #\text{TEMP} \\text{then}\)
  \(\text{delr}(#\text{TEMP}, #\text{LATER})\)
  \(\text{if} \ # \ #\text{TEMP} \ \text{eq} (\)
  \(\text{then}\)
  \(\text{augr}(#\text{TEMP}, #\text{LATER})\); \)
  \(\text{delr}(\text{ATTRIB}, @\text{TEMP})\);
  \(\text{delr}(\text{STORE}, #\text{TEMP})\)
  \(\text{trans}(\text{call})\)
  \(\text{delr}(\text{STORE}, #\text{TEMP})\)
\(\rangle\)

099 (procedure.identifier)::=
  \(\langle\text{identifier}\rangle\)
  99.1
  \(\langle\text{if} (\rangle \ \& \ n \ \& \ \text{STORE} \)
  99.1
  \(\text{then}\)
  \(-1\)
  \(\langle\text{if} \ e(k (\rangle \ \text{of} \ \text{STORE}) \ \text{not} \ \text{OBJECT} \rangle c\)
  99.1
  \(\langle\text{procedure,formal procedure}\rangle\)
  \(\text{then}\)
  \(\text{link}(#\text{TEMP}, () \ \text{of} \ \text{STORE})\); \)
  99.1
  \(\text{trans}(\text{rpsub}(() \ ))\) 99.1
  \(\langle\text{else} \ \text{halt} \)
  \(\langle\text{else} \ \text{halt} \rangle\)
\(\rangle\)

100 (might empty par proc)::=
  \(\langle\text{(actual pars)}\rangle\)
  \(\langle\text{empty}\rangle\)
  \(\langle\text{maybeformalsub}(e \ #\text{TEMP})\rangle\)

101 (empty statement)::=
  \(\langle\text{empty}\rangle\)

102 (empty)::=

103 (structured statement)::=
   (compound statement)|(conditional statement)|
   (repatitive statement).

104 (compound statement)::=
   begin (statement) {  } (statement) end

105 (conditional statement)::=
   (if statement)|(case statement)

106 (if statement)::=
   if
      at(#ATEMP);  
      trans(loc(#STEMP))  
      {expression}  
      boolcheck(#ATEMP);  
      delr(ATTRIB,#ATEMP)  
    then
      loc(#BPTR);  
      si k(#BPTR of STORE),ST is ()
    } (statement)
    si #BPTR of STORE,#BPTR1;
    putifthen(#BPTR1 of STORE);
    delr(STORE,#BPTR1);
    trans(delr(STORE,#STEMP))  
    else
      at(#ATEMP);
      trans(loc(#STEMP))  
      {expression}  
      boolcheck(#ATEMP);
      delr(ATTRIB,#ATEMP)  
    then
      loc(#BPTR);  
      si k(#BPTR of STORE),ST is ()
    } (statement)
    si #BPTR of STORE,#BPTR1;  
    else
      loc(#BPTR);
      si k(#BPTR of STORE),ST is ()
    } (statement)
    si #BPTR of STORE,#BPTR2;
    putifthenelse(#BPTR1 of STORE,#BPTR2 of STORE);
    delr(STORE,#BPTR2);
    delr(STORE,#BPTR1);
    trans(delr(STORE,#STEMP))  

107 \begin{case}
\text{\texttt{case}}
\texttt{at(\#ATEMP);}
\texttt{trans(loc(\#STEMP))}
\end{case}

108 \begin{case}
\text{\texttt{expression}}
\texttt{explicit(\#ATEMP);}
\texttt{nonreal(\#ATEMP);}
\texttt{augl(ATTRIB,\#TAGTYPE)}
\texttt{link(\#TAGTYPE \texttt{in} ATTRIB, \#ATEMP);}
\texttt{delr(ATTRIB, \#ATEMP)}
\end{case}

\begin{of}
\texttt{ugl(\#STORE, \#LABEL);}
\texttt{sl \texttt{k(\#LABEL \texttt{in} \#STORE)} \texttt{is ()}}
\end{of}

\begin{case}
\text{\texttt{case list element}}}
\end{case}

\begin{case}
\text{\texttt{dell(\#STORE, \#LABEL);}}
\texttt{trans(\#ATAGTYPE);}
\texttt{delr(\#STEMP, \#STORE)}
\end{case}

109 \begin{repetitive}
\texttt{while statement);}
\texttt{(repeat statement);}
\texttt{(for statement)}
\end{repetitive}
110 (while statement)::=
   while
   \( \text{at} (#ATEMP); \)
   \( \text{trans}(\text{loc}(#STEMP)) \)
   \( \text{expression} \)
   \( \text{boolcheck}(#ATEMP); \)
   \( \text{delr}(\text{ATTRIB}, #ATEMP) \)
   \( \text{do} \)
   \( \text{loc}(#BPTR); \)
   \( \text{si} \ k \ #BPTR \text{ST is } () \)
   \( \text{statement} \)
   \( \text{si} \ #BPTR \text{ of } \text{STORE}, #BPTR1; \)
   \( \text{trans}(\text{while } e \ e \ k \ #STEMP \text{ eq true do} \)
   \( e \ e \ l \ k \ #BPTR1 \)
   \( ) \)
   \( \text{delr}(\text{STORE}, #BPTR1); \)
   \( \text{trans}(\text{delr}(\text{STORE}, #STEMP)) \)

111 (repeat statement)::=
   repeat
   \( \text{loc}(#BPTR); \)
   \( \text{si} \ k \ (#BPTR \text{ of } \text{STORE}, T \text{ is } () \)
   \( \text{statement} \)
   \( \text{statement} \}
   \( \text{until} \)
   \( \text{at}(#ATEMP); \)
   \( \text{trans}(\text{loc}(#STEMP)) \)
   \( \text{expression} \)
   \( \text{boolcheck}(#ATEMP); \)
   \( \text{delr}(\text{ATTRIB}, #ATEMP); \)
   \( \text{si} \ #BPTR \text{ of } \text{STORE}, #BPTR1; \)
   \( \text{trans}(e \ e \ l \ k \ #BPTR1); \)
   \( \text{while} \ e \ k \ #STEMP \text{ eq false do} \)
   \( e \ e \ l \ k \ #BPTR1 \)
   \( ) \)
   \( \text{delr}(\text{STORE}, #BPTR1); \)
   \( \text{trans}(\text{delr}(\text{STORE}, #STEMP)) \)

112 (for statement)::=
   (to statement); (down to statement)
113 \text{(to statement)}::= \\
\text{for} \\
\quad \text{at}(\#ATEMP); \\
\quad \text{trans}\left(\text{loc}(\#STEMP)\right) \\
\quad \{\text{identifier}\} := \\
\quad 113.1 \\
\quad \text{ccontvar}(\cdot, \#ATEMP); \\
\quad 113.1 \\
\quad \text{augl}(\text{STORE,}\#\text{CONTVAR}); \\
\quad \text{sl k}(\#\text{CONTVAR}\text{\:\:}\text{\_store})); 113.1 \\
\quad \} \\
\quad \text{initial value)}\text{ to} \\
\quad \text{at}(\#ATEMP); \\
\quad \text{trans}(\text{loc}(\#STEMP)) \\
\quad \} \\
\quad \text{(expression)} \\
\quad \text{typecheck}(\text{prev}(\#ATEMP), \#ATEMP); \\
\quad \text{trimr}(\text{ATTIB, prev}(\#ATEMP)); \\
\quad \text{loc}(\#BPTR); \\
\quad \text{sl k}(\#BPTR\text{\:}\text{of}\text{\:}\text{\_store}), \text{ST is} \cdot) 113.1 \\
\quad \} \\
\quad \text{(statement)} \\
\quad \text{delr}(\text{STORE,}\#\text{CONTVAR}); \\
\quad \text{sl \#BPTR\::\text{of}:\text{\_store}, \#BPTR1}; \\
\quad \text{fortotrans}(\#BPTR1\text{\:}\text{of}\text{\:}\text{\_store}); \\
\quad \text{delr}(\text{STORE,}\#\text{BPTR}); \\
\quad \text{trans}(\text{trimr}\text{\:}\text{\_store, prev}(\#STEMP)) 113.1 \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \\
\} \<expression> \\
\text{typecheck}(\text{prev}(\#ATEMP), \#ATEMP); \\
\text{trimr}(\text{ATTIB, prev}(\#ATEMP)); \\
\text{loc}(\#BPTR); \\
\text{sl k}(\#BPTR\text{\:}\text{of}\text{\:}\text{\_store}), \text{ST is} \cdot) 113.1 \\
\{ \\
\text{statement} \\
\text{delr}(\text{STORE,}\#\text{CONTVAR}); \\
\text{sl \#BPTR\::\text{of}:\text{\_store}, \#BPTR1}; \\
\text{fortotrans}(\#BPTR1\text{\:}\text{of}\text{\:}\text{\_store}); \\
\text{delr}(\text{STORE,}\#\text{BPTR}); \\
\text{trans}(\text{trimr}\text{\:}\text{\_store, prev}(\#STEMP)) \\
\} \\
\] \\
\} \\
\} \\
\} \\
\} \\
\} \<expression> \\
\text{typecheck}(\text{prev}(\#ATEMP), \#ATEMP); \\
\text{trimr}(\text{ATTIB, prev}(\#ATEMP)); \\
\text{delr}(\text{ATTIB,}\#\text{ATEMP}); \\
\text{trans}(\text{delr}(\text{STORE,}\#\text{STEMP})) \\
\} \<expression>
115 (down to statement)::=
  for
    \texttt{at}(#ATEMP);
    \texttt{trans(loc(#STEMP))}
  \{identifier\} :=
  \texttt{ccontvar}();
\texttt{augl}(STORE,#CONTVAR);
  \texttt{si} #(CONTVAR in STORE),();
\texttt{trans(loc(IISTEMP)}
  \texttt{expression},[typecheck(prev(#ATEMP),#ATEMP);
  trimr(ATTRIB,prev(#ATEMP)));
  \texttt{loc}(#BPTR);
  \texttt{si} k(#BPRE of STORE),ST is ();
  \texttt{statement})
  \texttt{delr}(STORE,#CONTVAR);
  \texttt{si} #BPTR of STORE,#BPTR1;
  for downto \texttt{trans(#BPTR1 of STORE)};
  \texttt{delr}(STORE,#BPTR1);
  \texttt{trans(trimr(STORE,prev(#STEMP))})
CPS Macros for PSS

001 \texttt{adjin}(\alpha) = \texttt{adj}1(\alpha, k \alpha, \#1 \text{ of } k \alpha)
\text{ if } \alpha = \#\text{ATEMP}, \#\text{STEMP} @

002 \texttt{adjsin}(\alpha, \beta) = \texttt{adjin}(\alpha); \texttt{adjin}(\beta)
\text{ if } \alpha, \beta = \#\text{ATEMP}, \#\text{STEMP} @

003 \texttt{and}(\alpha, \beta, \gamma) = \neg 1
\text{ if } e(k, k \beta \cap \text{TYPE}) \notin \{\text{scalar, subrange}\}
\text{ then } \neg 1
\text{ if } e(k, k \gamma \cap \text{TYPE}) \notin \{\text{scalar, subrange}\}
\text{ then } \texttt{adjsin}(\beta, \gamma);
\text{ if } l \neq \beta \text{ eq BOOL}
\text{ then }
\text{ if } l \neq \gamma \text{ eq BOOL}
\text{ then }
\texttt{link}(\alpha, \text{Boolean in ATTRIB})
\texttt{else halt}
\texttt{else halt}
\texttt{else halt}
\text{ else halt}
\text{ \texttt{\alpha = prev(prev(\#ATEMP)), } \beta = prev(\#ATEMP), \gamma = \#ATEMP @}
arrayiden(α, β) =
  adjsin(α, β); if $e \# 1$ of $k \alpha \}$ eq $e \# 1$ of $k \beta \}$
    then
      augl(STORE, #DIMEN); s1 $k$ (#DIMEN lm STORE), $e \# 1$ of $k \alpha$
      repeat
        forwards(α, β); at(#ATEMP); link(#ATEMP, k prev(α));
        at(#ATEMP); link(#ATEMP, k prev(prev(β)));
        explicits(prev(#ATEMP), #ATEMP); scalarorsubrangesetpidentical(prev(#ATEMP), #ATEMP);
        trimr(ATTRIB, prev(#ATEMP)); (s1 $k$ (#DIMEN lm STORE),
          2 e $\alpha$ (e $k$ (#DIMEN lm STORE))
          prd
          )
      until e $k$ (#DIMEN lm STORE) eq 1
      dell(STORE, #DIMEN); /
      forwards(α, β); identical(α, β); else halt
      a α=prev(#ATEMP), β=#ATEMP a

assign(α, β) =
  explicits(α, β); if e $k \beta$ eq nil
    then
      link(k α, nil of HEAP) else
      if #TAGFIELD $\epsilon \# 1$ STORE
        then
          dell(STORE, #TAGFIELD); corresponds(α, β); forward(α); corresponds(α, β) else corresponds(α, β)
      a α=prev(#TEMP), β=#TEMP a
006 backidtp(α) =
    link(#STEMP, k ∧ of STORE);
    link(#ATEMP, k ∧ STEMP ∩ ATTRIB);
    adj1(#ATEMP, k ∧ ATEMP, prev(k ∧ ATEMP));
    while e k ∧ ATEMP ineq α do
      if e k ∧ ATEMP eq φ
        then
          adj1(#STEMP, k ∧ STEMP, k ∧ STEMP);
          adj1(#ATEMP, k ∧ ATEMP, k ∧ STEMP ∩ ATTRIB);
        adj1(#ATEMP, k ∧ ATEMP, prev(k ∧ ATEMP));
        a α = (identifier) φ
    if e k ∧ ATEMP eq φ
      then
        adj1(α, k ∧ ATEMP, prev(k ∧ ATEMP));
      else adj1(α, k ∧ ATEMP, prev(k ∧ ATEMP));
        a α = #STEMP φ

008 boolcheck(α) =
    explicit(α);
    if e(k ∧ ATEMP ∩ TYPE) ∈ {scalar, subrange} then
      if l DEF of k ∧ ATEMP ineq (false, true)
        then halt
      else halt
        a α = #ATEMP φ
009 \( \text{cadditional}(\alpha, \beta, \gamma, \delta) = \)
\( \text{explicit}(\beta, \delta); \)
\( \text{if} \ \gamma \notin \{+,-\}; \)
\( \text{then} \)
\( \text{-1}; \)
\( \text{if} \ e(k, k \beta \cap \text{TYPE}) = \text{set} \)
\( \text{then} \)
\( \text{-1}; \)
\( \text{if} \ e(k, k \delta \cap \text{TYPE}) = \text{set} \)
\( \text{then} \)
\( \text{typesets}(\alpha, \beta, \delta) \)
\( \text{else} \text{halt} \)
\( \text{else} \)
\( \text{timesnum}(\alpha, \beta, \delta) \)
\( \text{else} \)
\( \text{and}(\alpha, \beta, \delta) \)
\( \alpha = \text{prev}(\text{prev} (#ATEMP)), \beta = \text{prev} (#ATEMP), \gamma = \text{adding operat} \)
\( \delta = #ATEMP \) \( \alpha \)

010 \( \text{call} = \)
\( \text{if} \ k \#STEMP = \text{readln} \ 1m \text{STORE} \)
\( \text{then} \)
\( \text{exinputln} \)
\( \text{else} \)
\( \text{if} \ k \#STEMP = \text{writeln} \ 1m \text{STORE} \)
\( \text{then} \)
\( \text{exoutputln} \)
\( \text{else} \)
\( \text{if} \ k \#STEMP \notin \text{STANDF} \cup \text{AB} \cup \text{STANDP}1 \cup \text{STANDP}2 \)
\( \text{then} \)
\( 2; \)
\( e(1, k \#STEMP) \)

011 \( \text{carray}(\alpha) = \)
\( \text{explicit}(\alpha); \)
\( \text{-1}; \)
\( \text{if} \ e(k, k \alpha \cap \text{TYPE}) = \text{array} \)
\( \text{then} \)
\( \text{adjin}(\alpha); \)
\( \text{if} \ k(#DIMEN), 1m \text{STORE}, e \#1 \text{ of } k \alpha \)
\( \text{else} \text{halt} \)
\( \alpha = #ATEMP \) \( \alpha \)

012 \( \text{casecompair}(\alpha, \beta) = \)
\( \text{if} \ (1, k \alpha) \cup (1, k \beta) = \emptyset \)
\( \text{then} \)
\( \text{if} \ k \alpha, e \ k \beta \)
\( \text{else} \text{halt} \)
\( \alpha = #ALABEL \ 1m \text{STORE}, \beta = #STEMP \) \( \alpha \)
013 \textbf{case} \text{id}(\alpha, \beta) = -1 \\
\text{while } e(k \beta \in \{\alpha, \#SWITCH\} \text{ do} \\
\text{   } \text{rforward}(\beta); \\
\text{   } \text{if } e(k \beta = \#EXTRA} \\
\text{then} \\
\text{   } \text{link}((\#EXTRA \text{ im } \text{STORE}, k \beta)); \\
\text{   } \text{rlfixedsec}((\#EXTRA \text{ im } \text{STORE}) \\
\text{)} \\
\text{\quad } \alpha = \langle \text{identifier} \rangle, \beta = \#\text{TEMP} \oplus \\
\textbf{014 caseitem}= \\
\text{-2} \\
\text{if } e(k \#\text{TEMP} \in \text{TYPE}) \in \{\text{scalar, subrange}\} \\
\text{then} \\
\text{   } \text{-1} \\
\text{   } \text{if } l(k \#\text{TEMP} \in \text{ATTRIB}) \in \l k(#\text{TAGTYPE im ATTRIB}) \\
\text{then} \\
\text{   } \text{aupr}(k \text{ prev}((\#\text{TEMP}), e(k \#\text{TEMP})) \\
\text{else } \text{halt} \\
\text{else } \text{halt}
\textbf{015 ccompexp}(\alpha, \beta, \gamma) = \\
\text{if } \beta \in \{=, \langle\rangle\} \\
\text{then} \\
\text{   } \text{egnoteq}(\alpha, \gamma) \\
\text{else} \\
\text{if } \beta \in \{\langle\rangle\} \\
\text{then} \\
\text{   } \text{lorg}(\alpha, \gamma) \\
\text{else} \\
\text{if } \beta \in \{=, \langle\rangle\} \\
\text{then} \\
\text{   } \text{leorge}(\alpha, \gamma) \\
\text{else } \text{in}(\alpha, \beta) \\
\text{\quad } \alpha = \text{prev}(\#\text{ATEMP}), \beta = \langle \text{relational operator} \rangle, \gamma = \#\text{TEMP} \oplus
016 `contvar(α, β) =
  if α ∈ n 1 STORE
  then
    -1
    if e(k (α of STORE) ∩ OBJECT) ∈
    \{variable, formal value\}
    then
      -1
      link(β, k (α of STORE) ∩ ATTRIB);
      explicit(β);
    -1
    if (k β ∩ TYPE) ∈ \{scalar, subrange\}
    then
      if k β eg real \text{ in} ATTRIB
      then halt
      else halt
    else halt
  else halt

α = \{identifier\}, β = #ATEMP @

017 `contvar(α, β) =
  if α ∈ n 1 STORE
  then
    -1
    if e(k (α of STORE) ∩ OBJECT) ∈
    \{variable, function, formal value, formal variable\}
    then
      -1
      link(β, k (α of STORE) ∩ ATTRIB)
    else halt
  else halt

α = \{identifier\}, β = #ATEMP @

018 `not(α, β) =
  explicit(β);
  -1
  if e(k β ∩ TYPE) ∈ \{scalar, subrange\}
  then
    if n 1 DEF of k β eq BOOL
    then
      link(α, k β)
    else halt
  else halt

α = prev(#ATEMP), β = #ATEMP @
019 collectdigits(α) =
  if e $1 of INPUT ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  then
    s1 kα, e $1 of INPUT;
    dell(INPUT, $1);
  else halt;
while e $1 of INPUT ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} do
  (s1 kα, e $1 of INPUT);
  con
  dell(INPUT, $1)

α =#STEMP α

020 construction=
  if k #ATEMP eq integer lm ATTRIB
  then
    trans(link(integer lm ATTRIB, #STEMP))
  else
    if k #ATEMP eq real lm ATTRIB
    then
      trans(link(real lm ATTRIB, #STEMP))
    else
      if k #ATEMP eq Boolean lm ATTRIB
      then
        trans(link(Boolean lm ATTRIB, #STEMP))
      else
        if k #ATEMP eq char lm ATTRIB
        then
          trans(link(char lm ATTRIB, #STEMP))
        else
          trans(at(e k #ATEMP);
          at(#ATEMP);
          link(#ATEMP, #T of ATTRIB))
          typetranslate(#ATEMP);
          trans(dell(ATTRIB, #ATEMP);
          link(#T of ATTRIB, #STEMP))
      end
    end
  end
  if k #ATEMP & TYPE eq array
  then trans(adjin(#STEMP))
021 constid(α) =
   if α ∈ n STORE
      then -1
         if e(k (α of STORE) ∩ OBJECT) eq constant
            then s1 k #TEMP, e (k (α of STORE));
               -1
               link (k (α of STORE) ∩ ATTRIB, k #TEMP)
            else halt
            else halt
      else α = (identifier) θ

022 contnewtype(α, β; Y) =
   while k β ineq 0 do
      (if n k β eq #T
       then
         joins (α; Y); 
         newtype (β; Y)
      else -1
      if k k β ineq 0
      then
        (link (α, k Y);
         while next (n k β) ineq #T do forwards (β; Y))
       )
       forwards (β; Y)
   )
  @ α = #IDTEMP lm ATTRIB, β = prev (#TEMP), Y = #ATEMP @

023 conttrans(α, β; Y) =
   while k β ineq 0 do
      (if n k β eq #T
       then
         joins (α; Y); 
         transfer (β; Y)
      else -1
      if k k β ineq 0
      then
        (link (α, k Y);
         while next (n k β) ineq #T do forwards (β; Y))
       )
       forwards (β; Y)
   )
  @ α = #IDTEMP of ATTRIB, β = prev (#ATEMP), Y = #ATEMP @
024 contvarpart(α, β) =
  if n k & ineq #T then
    if n k β ineq #T then
      if e k α eg e k β then
        offsetsunique(α, β);
        contvarpart(α, β);
        else halt
        else halt
      else if e k β eq #T then
        typesunique(α, β);
        forwards(α, β);
        forwards(α, β);
        variants(α, β);
        else halt
      else halt
    else halt
    else halt
  else halt
  else if e k β eq #T then
    typesunique(α, β);
    forwards(α, β);
    forwards(α, β);
    variants(α, β);
    else halt
  else halt
  else halt
  @ α=prev(#ATEMP), β=#ATEMP @

025 correctindex=:
  if e k (#DIMEN Im STORE) ineq 0 then halt

026 correctpar(α) =
  if k α eq # then halt
  @α=#STEMP @

027 corresponds(α, β) =
  -1
  if k α eg real Im ATTRIB then
    -1
    if DEF of k β subset IN then
      2
      si k β eq (e k β; Ê+O); con
      finalassg(α, β);
      else finalassson(α, β)
    @ α=prev(#STEMP), β=#STEMP @
    else finalassg(α, β);
    finalassson(α, β)
  @ α=prev(#STEMP), β=#STEMP @
028 \texttt{cpointer}(\alpha) = \texttt{explicit}(\alpha); \\
\quad \texttt{-1} \\
\quad \texttt{if} (k \ k \alpha \cap \text{TYPE}) \texttt{eq} \texttt{pointer} \\
\quad \quad \texttt{then} \\
\quad \quad \quad \texttt{adjin}(\alpha) \\
\quad \quad \texttt{else} \texttt{halt} \\
\quad \alpha = \#\text{ATEMP} \ \beta \\

029 \texttt{crecord}(\alpha) = \texttt{explicit}(\alpha); \\
\quad \texttt{-1} \\
\quad \texttt{if} (k \ k \alpha \cap \text{TYPE}) \texttt{ineq} \texttt{record} \\
\quad \quad \texttt{then} \texttt{halt} \\
\quad \alpha = \#\text{ATEMP} \ \beta \\

030 \texttt{crentvar}(\alpha, \beta) = \\
\quad \texttt{if} \alpha \in \text{\_l ST OR E} \\
\quad \quad \texttt{then} \\
\quad \quad \texttt{-1} \\
\quad \quad \texttt{if} (k \ (\alpha \text{of ST OR E}) \cap \text{OBJECT}) \in \\
\quad \quad \quad \{\text{variable, formal value, formal variable}\} \\
\quad \quad \quad \texttt{then} \\
\quad \quad \quad \texttt{-1} \\
\quad \quad \quad \texttt{link}(\beta, k \ (\alpha \text{of ST OR E}) \cap \text{ATTRIB}) \\
\quad \quad \quad \texttt{else} \texttt{halt} \\
\quad \quad \texttt{else} \texttt{halt} \\
\quad \alpha = (\text{identifier}), \beta = \#\text{ATEMP} \ \beta \\

031 \texttt{cselector}(\alpha, \beta) = \\
\quad \texttt{if} \alpha \in \text{\_l k \beta} \\
\quad \quad \texttt{then} \\
\quad \quad \quad \texttt{adj}(\beta, k \beta, \alpha \text{of} \ k \beta); \\
\quad \quad \quad \texttt{while} n k \beta \texttt{ineq} \texttt{#T do} \\
\quad \quad \quad \quad \texttt{forward}(\beta) \\
\quad \quad \quad \texttt{else} \texttt{halt} \\
\quad \alpha = (\text{identifier}), \beta = \#\text{ATEMP} \ \beta \\

032 \texttt{csign}(\alpha, \beta) = \texttt{explicit}(\alpha); \\
\quad \texttt{-1} \\
\quad \texttt{if} (k \ k \beta \cap \text{TYPE}) \in \{\text{scalar, subrange}\} \\
\quad \quad \texttt{then} \\
\quad \quad \quad \texttt{adj}(\beta) \\
\quad \quad \quad \texttt{if}(1 k \beta \text{subset IN}) \lor (1 k \beta \texttt{eq} \text{R}) \\
\quad \quad \quad \texttt{then} \\
\quad \quad \quad \quad \texttt{link}(\alpha, k \alpha) \\
\quad \quad \quad \quad \texttt{else} \texttt{halt} \\
\quad \quad \texttt{else} \texttt{halt} \\
\quad \alpha = \text{prev}(\#\text{ATEMP}), \beta = \#\text{ATEMP} \ \beta
033 $c_{term}(\alpha, \beta, y, \delta) =$
   $explicitits(\beta, \delta);$  
   if $y$ eq *
   then
      $1$
      if $e(k, k \beta \alpha \mathbb{N} \text{TYPE})$ eq set
      then
         $1$
         if $(k, k \beta \alpha \mathbb{N} \text{TYPE})$ eq set
         then  
            if $\#\text{ANYSET of ATTRIB} \in (k, k \beta, k \delta)$
            then
               link($\alpha, \#\text{ANYSET of ATTRIB}$)
               else
                  typesets($\alpha, \beta, \delta$)
               else
                  halt
         else
            timesnum($\alpha, \beta, \delta$)
      else
         if $y$ eq '/'
         then  
            realdiv($\alpha, \beta, \delta$)
         else
            if $y$ eq div
            then
               intdiv($\alpha, \beta, \delta$)
            else
               if $y$ eq mod
               then
                  intdiv($\alpha, \beta, \delta$)
               else
                  and($\alpha, \beta, \delta$)
      \vspace{1cm}
      $\alpha = \text{prev}($prev($\#\text{ATEMP}$)$), \beta = \text{prev}($\#\text{ATEMP}$),$
      $y = (\text{multiplying operator}), \delta = \#\text{ATEMP} \theta$

034 $\text{decimal}(\alpha) =$
   $\text{loc}($\#\text{STEM}\text{P}$)$;
   \text{collectdigits}($\#\text{STEM}\text{P}$)$;
   $2$
   $\text{si} \ k \ \text{prev}($\#\text{STEM}\text{P}$)$, $e \rightleftharpoons (e \ k \ \text{prev}($\#\text{STEM}\text{P}$), e \ k \ \#\text{STEM}\text{P}$)
   \text{con}$
   \text{delr}($\text{STORE}, $\#\text{STEM}\text{P}$)$;
   if $e \ #1$ of INPUT eq $E$
   then
      $2$
      $\text{si} \ k \alpha, e \rightleftharpoons (e \ k \alpha, E)$
      \text{con}$
   \text{dell}($\text{INPUT}, $\#1$)$;
   \text{factor}$
   $\alpha = $\#\text{STEM}\text{P} \theta$

035 $\text{dispose}(\alpha) =$
   $2$
   $\text{break}($k \alpha, k \alpha \mathbb{N} \#\text{HEAP})$
   $\alpha = $\#\text{STEM}\text{P} \theta$
endsec=
at(#ATEMP);
link(#ATEMP,k prev(#ATEMP));
secids(prev(prev(#ATEMP)),prev(#ATEMP),#ATEMP);
deir(ATTRIB,#ATEMP)

eofeoln(α,β)=
break(α,k α);
if β eq eoln
    /
      if e $1 of INPUT eq #EOLN
        then
          si k α,true
        else
          si k α,false
      else
        if e $1 of INPUT eq #EOF
          then
            si k α,true
          else
            si k α,false
      /

α=#STEMP,β=eoln,eof @

eqnoteq(α,β)=
explicit(α,β);
if e(k k α ∩ TYPE) eq pointer
    /
      if e(k k β ∩ TYPE) eq pointer
        then
          pointeriden(α,β)
        else
          halt
      else
        leorge(α,β)
    /

α=prev(#ATEMP),β=#ATEMP @
039 \textit{exclr}(\alpha, \beta) = \\
\textbf{break}(\alpha, \text{\texttt{k}} \alpha) ; \\
\textbf{at}(\#\text{ATEMP}) ; \\
\textbf{link}(\#\text{ATEMP}, \#1 \text{ of DEF of (char \texttt{l}m ATTrib)}) ; \\
\textbf{while} \\
\textbf{2} \\
\textbf{con} \\
\textbf{ordihal}(\#k \text{ #ATEMP of DEF of (char \texttt{l}m ATTrib)}) \text{ ineq } \beta \\
\textbf{do} \\
\textbf{Forward}(\#\text{ATEMP}) ; \\
\textbf{if} \#k \text{ #ATEMP ineq } \emptyset \\
\textbf{then} \\
\textbf{1} \kappa, \alpha \not\in \emptyset \text{ k #ATEMP} \\
\textbf{else} \text{halt} \\
\alpha = \text{prev}(\text{prev}(\#\text{STEMP})) ; \beta = \text{#STEMP} \not\in \\
\hline

040 \textit{exinputln} = \\
\textbf{while} \#1 \text{ of INPUT ineq } \#\text{EOLN do} \\
\textbf{dell}(\text{#INPUT}, \#1) ; \\
\textbf{dell}(\text{#INPUT}, \#1) \\
\hline

041 \textit{exnew}(\alpha) = \\
\textbf{at}(\#\text{ATEMP}) ; \\
\textbf{link}(\#\text{ATEMP}, \#1 \text{ of } \emptyset \kappa \not\in \emptyset \text{ ATTrib}) ; \\
\textbf{explicit}(\alpha) ; \\
\textbf{if} \ e(\#1 \text{ of WIDTH}) \text{ of k #ATEMP ineq } \emptyset \\
\textbf{then} \\
\textbf{augr}(\text{HEAP, w}) ; \\
\textbf{link}(\kappa, \text{ w of HEAP}) ; \\
\textbf{augl}(\text{STORE, #WIDTH}) ; \\
\textbf{2} \\
\textbf{if} \ e(\#1 \text{ of WIDTH}) \text{ of k #ATEMP} \\
\textbf{while} \ e \emptyset \emptyset \text{ k #WIDTH \#STORE ineq } \emptyset \text{ do} \\
\textbf{augr}(\text{HEAP, w}) ; \\
\textbf{2} \\
\textbf{dell}(\text{#INPUT}, \#\text{ATEMP}) \\
\textbf{1} \alpha = \#\text{STEMP} \emptyset
042 \text{exord}(\alpha,\beta) = \\
\quad \text{break}(\alpha, k \alpha); \\
\quad 2 \\
\quad \text{si } k \alpha, e \to (+, \text{ordinal}(e(k \beta) \text{of DEF of } k \beta)); \\
\quad \text{con} \\
\quad \text{if } g(k \alpha \notin \text{IN} \\
\quad \text{then } \text{halt} \\
\quad \alpha = \text{prev}(/\text{prev}(\text{#STEMP})\), \beta = \text{prev}(\text{#STEMP} \) \\
043 \text{exoutputln} = \\
\quad \text{augr}(\text{OUTPUT}, \text{#EDLN}) \\
044 \text{explicit}(\alpha) = \\
\quad -1 \\
\quad \text{if } e(k \alpha \cap \text{TYPE}) \text{ eq type identifier} \\
\quad \text{then} \\
\quad \quad 2 \\
\quad \quad \text{adj}(\alpha, k \alpha, k \alpha); \\
\quad \quad \text{explicit}(\alpha) \\
\quad \alpha = \text{#ATEMP}, \text{#TAGTYPE lm ATTRIB} \) \\
045 \text{explicit}(\alpha, \beta) = \\
\quad -1 \\
\quad \text{explicit}(\alpha); \\
\quad \text{explicit}(\beta) \\
\quad \alpha = \text{prev}(\text{#ATEMP}), \beta = \text{#ATEMP} \) \\
046 \text{expred}(\alpha, \beta) = \\
\quad \text{break}(\alpha, k \alpha); \\
\quad -1 \\
\quad \text{if } k \alpha \text{ eq } \varnothing \\
\quad \text{then} \\
\quad \quad -1 \\
\quad \quad \text{link}(k \beta, \alpha); \\
\quad \quad \text{if } \text{prev}(e(k \beta) \text{of DEF of } k \beta) \text{ ineq } \varnothing \\
\quad \quad \text{then} \\
\quad \quad \quad -1 \\
\quad \quad \quad \text{si } k \alpha, \varnothing \text{ prev}(e(k \beta) \text{of DEF of } k \beta) \\
\quad \quad \quad \text{else } \text{halt} \\
\quad \quad \alpha = \text{prev}(/\text{prev}(\text{#STEMP})\), \beta = \text{#STEMP} \)
047 exread(α)=
    if 1 INPUT ineq ()
        then
            while e $1$ of INPUT eq #EOLN do
                dell(INPUT,#EOLN);
                if l DEF of k k α subset IN
                    then
                        while e $1$ of INPUT eq do
                            dell(INPUT,$1);
                            exreadint(α)
                        else
                            -1
                            if k k α eq real lm ATTRIB
                                then
                                    while e $1$ of INPUT eq do
                                        dell(INPUT,$1);
                                        exreadreal(α)
                                else
                                    exreadchar(α)
                        -1
                        if e k α $n$ 1 DEF of k k α
                            then halt
                            else halt
            α = #STEMP α
        end
048 exreadchar(α)=
    si k α $e$ $1$ of INPUT;
    dell(INPUT,$1)
    α = #STEMP α
049 exreadint(α)=
    exsign(α);
    loc(#STEMP);
    exunsignedint(#STEMP);
    $2$
    si k prev(#STEMP),e A
        (e k prev(#STEMP), e k #STEMP);
        con
    dell(STORE,#STEMP)
    α = #STEMP α
050 exreadreal(α)=
    exsign(α);
    loc(#STEMP);
    exunsignedreal(#STEMP);
    $2$
    si k prev(#STEMP),e A
        (e k prev(#STEMP), e k #STEMP);
        con
    dell(STORE,#STEMP)
    α = #STEMP α
051 $\text{exsign}(\alpha) =$
\[
\text{if } e \in \{+,-\} \text{ of INPUT then} \quad \\
\text{si } k \alpha, e \in \{+,-\} \text{ of INPUT;} \quad \\
dell(\text{INPUT, } e) \quad \\
\text{else} \quad \\
\text{si } k \alpha, + \quad \\
\alpha = \#\text{TEMP } \alpha \quad \\
\]

052 $\text{exsucc}(\alpha, \beta) =$
\[
\text{break}(\alpha, k \alpha); \quad \\
\text{if } k \alpha = \emptyset \text{ then} \quad \\
\text{si } \text{link}(k \beta, \alpha); \quad \\
\text{if } \text{next}(e k \beta \text{ of DEF of } k \beta) \in \emptyset \text{ then} \quad \\
\text{si } k \alpha, \text{next}(e k \beta \text{ of DEF of } k \beta) \quad \\
\text{else} \quad \\
\text{halt} \quad \\
\alpha = \text{prev}((\text{prev}(\#\text{TEMP})), \beta = \#\text{TEMP } \alpha \quad \\
\]

053 $\text{extrablock}(\alpha) =$
\[
\text{si } e(k \alpha \cap \text{TYPE}) = \text{scalar} \text{ then} \quad \\
\text{si } k \alpha \not= \{\text{integer } l, \text{ char } l, \text{ ATTRIB} \} \text{ then} \quad \\
\text{at}(\#\text{TEMP}); \quad \\
\text{link}(\#\text{TEMP}, e \text{ of DEF of } k \text{ prev}(\#\text{TEMP})); \quad \\
\text{repeat} \quad \\
\text{insert}(\#\text{TEMP}, e \text{ of } k \#\text{TEMP}); \quad \\
\text{si } k \text{ next}(\#\text{TEMP}, e \text{ of } k \#\text{TEMP}); \quad \\
\text{link}(\text{constant of } \text{OBJECT}, \text{next}(\#\text{TEMP})); \quad \\
\text{link}(k \text{ prev}(\#\text{TEMP}), k \text{ next}(\#\text{TEMP})); \quad \\
\text{adj}(\#\text{TEMP}, k \#\text{TEMP}, \text{prev}(k \#\text{TEMP})); \quad \\
\text{until } k \#\text{TEMP} = \emptyset; \quad \\
\text{dell}(\text{ATTRIB}, \#\text{TEMP}); \quad \\
\alpha = \#\text{TEMP } \alpha \quad \\
\]

054 $\text{exunsignedint}(\alpha) =$
\[
\text{collectdigits}(\alpha); \quad \\
\text{si } k \alpha, e \# (e k \alpha) \quad \\
\alpha = \#\text{TEMP } \alpha \quad \\
\]
055 expsinsedreal(\(\alpha\)) =
    collectdigits(\(\alpha\));
    
    (if \(a \neq 1\) of INPUT eq .
    then
    2
    \(sl \ k \alpha, e \neq (e \ k \alpha,.)\);
    con
dell(INPUT,\$1));
    decimal(\(\alpha\));
    
    else
    if \(a \neq 1\) of INPUT eq E
    then
    2
    \(sl \ k \alpha, e \neq (e \ k \alpha,E)\);
    con
dell(INPUT,\$1));
    factor
    else
    2
    \(sl \ k \alpha, e \neq (e \ k \alpha,E+0)\);
    con
    )

    2
    \(sl \ k \alpha, e \neq (e \ k \alpha)\);
    rch

    2
    \(sl \ k \alpha, e \neq (e \ k \alpha)\).
    rm

\(\alpha = #STEMP \ a\)
056 \texttt{exwrite}(\alpha) = \begin{cases} -1 & \text{if } e(k,k \alpha \in \text{TYPE}) \in \{\text{scalar, subrange}\} \\
\quad \text{if } e \, k \alpha \text{ ineq } \omega \\
\quad \text{then} \\
\quad \quad \text{augr(OUTPUT, } e \, k \alpha \text{)} \\
\quad \text{else halt} \\
\quad \text{else} \\
\quad \quad \text{if } e \, k \alpha \text{ ineq } \omega \\
\quad \quad \text{then} \\
\quad \quad \quad \text{augr(OUTPUT, } e \, k \alpha \text{)} \\
\quad \quad \quad \text{forward(} \alpha \text{)}; \\
\quad \quad \quad \text{augl(STATO, } \#\text{WIDTH}) \\
\quad \quad \text{si } k(\#\text{WIDTH in STORE}, e \, k \alpha); \\
\quad \quad \text{while } e \, k(\#\text{WIDTH in STORE}) \text{ ineq } 1 \text{ do} \\
\quad \quad \text{if } e \, k \alpha \text{ ineq } \omega \\
\quad \quad \quad \text{then} \\
\quad \quad \quad \quad \text{si } \#\text{ of OUTPUT, } e \, \alpha \quad (e \#\text{ of OUTPUT, } e \, k \alpha); \\
\quad \quad \quad \quad \text{con} \\
\quad \quad \quad \quad \text{forward(} \alpha \text{)}; \\
\quad \quad \quad \quad \text{si } k \#\text{WIDTH, } e \, \alpha \quad (e \#k \#\text{WIDTH}) \\
\quad \quad \quad \quad \text{prd} \\
\quad \quad \text{else halt;} \\
\quad \quad \text{delr(STATO, } \#\text{WIDTH}) \\
\quad \text{else halt} \\
\end{cases} \quad \alpha = \#\text{STEM} \quad \alpha

057 \texttt{factor} = \\
\quad \text{loc(} \#\text{STEM}) \\
\quad \text{exreadint(} \#\text{STEM}) \\
\quad \text{si } k \#\text{prev(} \#\text{STEM), } e \, \alpha \quad (e \#k \#\text{prev(} \#\text{STEM), } e \, k \#\text{STEM}) \\
\quad \text{con} \\
\quad \text{delr(STATO, } \#\text{STEM}) \\

058 \texttt{field}(\alpha, \beta) = \\
\quad \text{forwards(} \alpha, \beta) \\
\quad \text{(if } e \#k \alpha \text{ ineq } \#\text{VP} \text{ then} \\
\quad \quad \text{if } e \#k \beta \text{ ineq } \#\text{VP} \\
\quad \quad \text{then} \\
\quad \quad \quad \text{fixedpartiden}(\alpha, \beta) \\
\quad \quad \text{else halt} \\
\quad \text{else} \\
\quad \quad \text{if } e \#k \beta \text{ eq } \#\text{VP} \\
\quad \quad \text{then} \\
\quad \quad \quad \text{variantpartiden}(\alpha, \beta) \\
\quad \quad \text{else halt} \\
\quad \text{)} \\
\quad \text{forwards(} \alpha, \beta) \\
\quad \alpha \#\text{prev(} \#\text{ATEMP}), \beta = \#\text{ATEMP} \quad \alpha
059 finalassign(\(\alpha, \beta\)) =
    \[\text{if } e(k \alpha, \emptyset \text{ TYPE}) \& \{\text{scalar, subrange}\} \text{ then} \]
    \[\begin{align*}
    \text{if } e k \beta \& n 1 \text{ DEF of } k \alpha & \text{ then} \\
    \text{else \hspace{0.5cm} halt} & \\
    \end{align*}\]
    \[\text{else} \hspace{0.5cm} \text{augl(STORE, \#WIDTH);} \]
    \[\begin{align*}
    \text{sl } k(\#\text{WIDTH}\ l m \text{ STORE}), \#1 \text{ of \#WIDTH of } k \alpha; \\
    \text{while } e k(\#\text{WIDTH} \ l m \text{ STORE}) \text{ ineq } 0 \text{ do} \\
    \text{sl } k \alpha, e k \beta; \\
    \text{forwards}(\alpha, \beta); \\
    \text{sl } k(\#\text{WIDTH} \ l m \text{ STORE}), e \text{ of } (e k(\#\text{WIDTH} \ l m \text{ STORE})) \text{ prd} \\
    \text{dell(STORE, \#WIDTH)} \\
    \end{align*}\]
    \[\text{a } \alpha = \text{prev}(\#\text{STEMP}), \beta = \#\text{STEMP} \& \]

060 finduint(\(\alpha\)) =
    \[\text{while } e k(\#\text{SUM} \ l m \text{ STORE}) \text{ ineq } 0 \text{ do} \]
    \[\begin{align*}
    \text{forward}(\alpha); \hspace{2cm} & \\
    \text{sl } k(\#\text{SUM} \ l m \text{ STORE}), e \text{ of } (e k(\#\text{SUM} \ l m \text{ STORE})) \text{ prd} \hspace{2cm} & \\
    \end{align*}\]
    \[\text{a } \alpha = \text{EXTRA of STORE, \#STEMP} \& \]
O61 firstpar =
-1
  if a k k prev(#STEMP) ∩ OBJECT ∈
  {formal function, formal procedure}
  then
    if #LATER ∉ m 1 k prev(#STEMP)
      then
        link(#STEMP,1 of k prev(#STEMP));
        link(#ATEMP, k #STEMP ∩ ATTRIB);
        trans(link(#STEMP,1 of k prev(#STEMP));
        -1
        link(k k #STEMP ∩ ATTRIB, #STEMP)
      else
        if k prev(#STEMP) ∉ STANDP
          then
            link(#STEMP,1 of k prev(#STEMP));
            trans(link(#STEMP,1 of k prev(#STEMP));
              if(k prev(#STEMP) ∉ AB) ∧
              -1
              (a(k k #STEMP ∩ OBJECT) ineq formal procedure)
              then
                link(#ATEMP, k k #STEMP ∩ ATTRIB);
                trans(link(k k #STEMP ∩ ATTRIB, #STEMP))
            
O62 firstvar(α, β) =
-loc(α);
  link(variable of OBJECT, α of STORE);
  link(β, α of STORE);
  trans(loc(α));
    link(variable of OBJECT, α of STORE);
    link(β, α of STORE)
  )

  α = {identifier}, β = #STEMP &
063 fixedpartiden(α, β) =
  if n k α ineq #T
  then
    if e k β ineq #T
    then
      if e k α eq e k β
      then
        offsetsunique(α, β);
        fixedpartiden(α, β)
      else
        halt
    else
      if e k β eq #T
      then
        typesunique(α, β);
        field(α, β)
      else
        halt
  a α=prev(#ATEMP), β=#ATEMP a

064 fidid(α, β) =
  idlimit(α);
  if α ∈ n l k β
  then augr(k β, (α, #OFFSET is ()))
  else
    halt
  a α=(identifier), β=prev(#ATEMP), #ATEMP a

065 fordowntotrans(α) =
  trans(rexPLICIT(prev(#STEMP)));
  (if e k prev(#STEMP) ∈
   -1
   copyr(DEF of k prev(#STEMP),
       (e k #STEMP) of DEF of k prev(#STEMP)
     ) )
  then
    while e k prev(#STEMP) ineq e k #STEMP do
      e e l k α;
      (s1 k prev(#STEMP),
       -1
       e prev(e k #STEMP) of DEF of k prev(#STEMP)
      ) ;
      s1 = prev(#STEMP),
  )
  a α=#BPTR of STORE a
066 formalff(α) =
  if α of STORE ∈ AB
  then
    spectral(α)
  else
    typecheck(prev(#ATEMP),#ATEMP);
    if α of STORE ≡ ord lm STORE
    then
      parord
    else
      maylatersub(α)
  end
  @ α = ⟨identifier⟩ @

067 formalpp(α) =
  if α of STORE ∈ STAN(DP)
  then halt
  else
    if α of STORE ≡ write lm STORE
    then
      writeactual
    else
      if α of STORE ≡ writeln lm STORE
      then
        writelnactual
      else
        maylatersub(α)
  end
  @ α = ⟨identifier⟩ @

068 formalsub(α) =
  augr(κ, α, x);
  link(formal value of OBJECT, x of k α);
  partype(α)
  @ α = prev(#STEMP),#STEMP @

069 formaltransfer =
  if #LATER ∈ n 1 k #STEMP
  then
    -1
    link(k, #LATER of k #STEMP,#LATER of $ of STORE)
  else
    insertr(STORE,#STEMP,#STEMP);
    link(#STEMP,$1 of k prev(#STEMP));
    insertr(STORE,#STEMP,#STEMP);
    link(#STEMP,$1 of $ of STORE);
    while $ k prev(#STEMP) ineq Ø do
      (link(formal value of OBJECT, k #STEMP);
      -1
      link(k, k prev(#STEMP) ⊆ ATTRIB, k #STEMP);
      forwards(prev(#STEMP),#STEMP))
    trimr(STORE,prev(#STEMP))
070 forstmtcheck(α) =
  if #CONTVAR n 1 STORE
  then
    while #CONTVAR n 1 STORE do
      if e k(#CONTVAR lm STORE) ineg α
      then
        s1 #CONTVAR lm STORE,#CONTVAR1
      else halt;
    while #CONTVAR1 n 1 STORE do
      s1 #CONTVAR lm STORE,#CONTVAR
    a α=(identifier) a
  .

071 fortoftrans(α) =
  trans(rexplit(prev(#STEMP)))
  (if e k prev(#STEMP) E
   -1
   copyl(DEF of k prev(#STEMP),
   /)
   next((e k #TEMP) of DEF of k
   prev(#STEMP)
   )
   then
   while e k prev(#TEMP) ineg e k #STEMP do
    (e e l k a;  
    (s1 k prev(#STEMP),
    -1
    e next(e k #STEMP) of DEF of k
    prev(#STEMP)
    )
    )
    s1 k prev(#STEMP), w
   )
  a α=#BPTR of STORE a

072 forward(α) =
  adj1(α,kα, next(kα))
  a α=#STEMP a
073 forwards(α, β) =
    forward(α);
    forward(β)

    α = prev(#STEMP), prev(#ATEMP), β = #STEMP, #ATEMP =>

074 fptidtrans(α, β) =
    if ordinal(α of ATTRIB) ≤ copyr(UINT, ordinal(β of ATTRIB))
       then
       link(α of ATTRIB, β)
    else
       at(#ATEMP);
       loc(#STEMP);
       backidtp(α);
       link(k #ATEMP, β);
       delr(STORE, #STEMP);
       delr(ATTRIB, #ATEMP)

    α = (identifier),
    β = (identifier) of STORE, (identifier) of k #STEMP =>
fpvalvarpar(α) =

    if e(k, k #TEMP ∩ OBJECT) eq formal function
    then
      -1
      (if e(k, α of STORE) ∩ OBJECT) eq formal function
      then
        typecheck(prev(#ATEMP), #ATEMP);
        latersub(α)
      else
        -1
        if e(k, α of STORE) ∩ OBJECT) eq function
        then
          formalff(α)
        else
          halt
      trans(link(k prev(#TEMP), k #TEMP))
    else
      -1
      if e(k, k #TEMP ∩ OBJECT) eq formal procedure
      then
        -1
        (if e(k, α of STORE) ∩ OBJECT) eq formal procedure
        then
          latersub(α)
        else
          -1
          if e(k, α of STORE) ∩ OBJECT) eq procedure
          then
            formalpp(α)
          else
            halt
        trans(link(k prev(#TEMP), k #TEMP))
    else
      trans(#TEMP, k #TEMP, $1 of k #TEMP);
    valvarpar
  @ α=(identifier) @
076 \texttt{fsub}(\alpha) =
\begin{align*}
\text{if } \alpha \in \text{STORE} & \text{ then } -1 \\
\text{if } e(k \alpha \text{ of STORE} \cap \text{OBJECT}) \in & \\
\{\text{function, formal function}\} & \text{ then } -1 \\
\text{link}(\text{#ATEMP}, k (\alpha \text{ of STORE}) \cap \text{ATTRIB}) ; & \\
\text{link}(\text{#STEMP}, \alpha \text{ of STORE}) ; & \\
\text{trans}(\text{fsub}(\alpha)) & \\
\text{else halt} & \\
\text{else halt} & \\
\end{align*}
\[ \alpha = \text{(identifier)} \]

077 \texttt{funaction} =
\begin{align*}
\text{insert}(\text{STORE}, \text{#STEMP}, \text{#STEMP}) ; \\
\text{link}(\text{#STEMP}, k \text{ prev}(\text{#STEMP})) ; \\
\text{loc}(\text{OUTPUT FROM } e \text{ n } k \text{ #STEMP}) ; \\
\text{si } k \text{ $ } \text{of } \text{STORE}, M \text{ is } (\omega) ; & -1 \\
\text{link}(k \text{ prev}(\text{#STEMP}) \cap \text{ATTRIB}, k \text{ $ } \text{of } \text{STORE}) ; \\
\text{parameters}(\text{#STEMP}) ; \\
\text{delr}(\text{STORE}, \text{#STEMP}) & \\
\end{align*}

078 \texttt{funcformalfunc}(\alpha) =
\begin{align*}
\text{link}(\text{#ATEMP}, k (\alpha \text{ of STORE}) \cap \text{ATTRIB}) ; & -1 \\
\text{maybeformalsub}(\alpha) ; & \\
\text{trans}(\text{link}(k \text{ #STEMP} \cap \text{ATTRIB}, \text{#STEMP}) ; & -1 \\
\text{rparlesscallf}(\text{#STEMP}) ; & \\
\end{align*}
\[ \alpha = \text{(identifier)} \]

079 \texttt{funval}(\alpha) =
\begin{align*}
\text{break}(\text{#STEMP}, k \text{ #STEMP}) ; & \\
\text{si } k \text{ #STEMP}, s1 \text{ of } k \text{ (OUTPUT FROM } \alpha \text{ of STORE}) ; & -1 \\
\text{rexplicit}(\text{#STEMP}) ; & \\
\text{if } k \text{ #STEMP} \cap \text{ATTRIB} \text{ eq real } l1m \text{ ATTRIB} & \text{ then } \\
\text{inttor}(\text{#STEMP}) & \\
\end{align*}
\[ \alpha = \text{(identifier)} \]

080 headcompile=
  insertr(STORE,#TEMP,#TEMP);
  link(#TEMP,k prev(#TEMP));
  adjin(#TEMP);
while k #TEMP ineq 0 do
  log(e,k,#TEMP);
  -1
  if e(k,k,#TEMP OBJ) e {formal value,formal variable} then
    -1
    link(k,k,#TEMP OBJ,ATTRIB,STORE);
    -1
    link(k,k,#TEMP OBJ,ATTRIB,STORE);
  else
    -1
    if e(k,k,#TEMP OBJ) eq formal function then
      link(formal fcn OBJ,ATTRIB,STORE);
    else
      link(formal procedure OBJ,ATTRIB,STORE);
      formaltransfer
    end
  end
end

081 headrun=
  trans(link(ATTRIB,STORE))
  staticlink
); -1
  if e(k,k,#TEMP OBJ) eq function then
    trans(function)
  else
    trans(procedure)
  end
082 $\text{identical}(\alpha; \beta) =$

$\text{explicitst}(\alpha; \beta);$

-1

if $e(k \ k \alpha \cap \text{TYPE}) \in \{\text{scalar, subrange}\}$
then

-1

if $e(k \ k \beta \cap \text{TYPE}) \in \{\text{scalar, subrange}\}$
then

scalarorsubrange$\text{typeidentical}(\alpha; \beta)$
else

halt

else

-1

if $e(k \ k \alpha \cap \text{TYPE}) \equiv \text{array}$
then

-1

if $e(k \ k \beta \cap \text{TYPE}) \equiv \text{array}$
then

arrayiden($\alpha; \beta$)
else

halt

else

-1

if $e(k \ k \alpha \cap \text{TYPE}) \equiv \text{record}$
then

-1

if $e(k \ k \beta \cap \text{TYPE}) \equiv \text{record}$
then

recordiden($\alpha; \beta$)
else

halt

else

-1

if $e(k \ k \alpha \cap \text{TYPE}) \equiv \text{set}$
then

-1

if $e(k \ k \beta \cap \text{TYPE}) \equiv \text{set}$
then

setiden($\alpha; \beta$)
else

halt

else

-1

if $e(k \ k \alpha \cap \text{TYPE}) \equiv \text{pointer}$
then

-1

if $e(k \ k \beta \cap \text{TYPE}) \equiv \text{pointer}$
then

pointeriden($\alpha; \beta$)
else

halt

else

halt

$@ \alpha = \text{prev}(\#\text{ATEMP}), \beta = \#\text{ATEMP} @$

083 $\text{idlimit}(\alpha) =$

if $\alpha \notin \text{IDSET}$
then

halt
else

if $\alpha \in \text{WD}$
then

halt

$@ \alpha = \langle\text{identifier} \rangle @$
084 idpar(α)=
    if e(k (α of STORE) ∩ OBJECT) = constatnt
    then
        singleconst(α);
        valuepar
    else
        -1
        (link(#ATEMP,k (α of STORE) ∩ ATTRIB);
         trans(rrfind(α));
         -1
         link(k k #STEMP ∩ ATTRIB,#STEMP)
         fpvalvarpar(α))
    α = (identifier) α

085 idunique(α)=
    ifα ∈ n copyr(STORE,φ)
    then halt
    else
        ifα ∈ n copyr(ATTRIB,φ)
        then halt
    α = (identifier) α
\begin{verbatim}
086 improvehead=
inserter(STORE,#STEMP,#STEMP);
link(#STEMP,k prev(#STEMP));
inserter(STORE,#STEMP,#STEMP);
link(#STEMP, of STORE);
adjoin(prev(#STEMP),#STEMP);
while k prev(#STEMP) ineq Ø do
  (if e(k k prev(#STEMP) ∩ OBJECT) ∉
   {formal function, formal procedure}
  then
    (if (#LATER ⊂ k #STEMP) ∨ (#LATER ⊂ k prev(#STEMP)
    then
      (delr(k prev(#STEMP),#LATER);
        insertr(STORE,#STEMP,#STEMP);
        link(#STEMP,1 of k prev(#STEMP));
        while k #STEMP ineq Ø do
          (augr(k prev(prev(#STEMP)),x);
            link(formal value of OBJECT,
                 x of k prev(prev(#STEMP));
              )
            )
      1
      link(k k #STEMP ∪ ATTRIB,
           x of k prev(prev(#STEMP)));
      forwards(#STEMP)
    )
  )
  delr(STORE,#STEMP);
trimr(STORE,k prev(#STEMP))

087 in(α,β)=
  explicitis(α,β);
  if e(k k α ∩ TYPE) ∈ {scalar, subrange}
  then
    scalarorsubrangenotypeidentical(α,β)
  else halted
else halted
  α = prev(#ATEMP), β = #ATEMP @
\end{verbatim}
088 \texttt{indexaction} =
\begin{align*}
&\texttt{at}(\#\text{ATEMP}); \\
&\texttt{link}(\#\text{ATEMP}, \text{prev}(\#\text{ATEMP})); \\
&\texttt{explicit}(\#\text{ATEMP}); \\
&\texttt{extrablock}(\#\text{ATEMP}); \\
&\texttt{delr}(\text{ATTRIB}, \#\text{ATEMP})
\end{align*}

089 \texttt{indextype}(\alpha, \beta) =
\begin{align*}
&\texttt{explicit}(\alpha, \beta); \\
&\begin{cases}
-1 & \text{if } e(k, k \cap \alpha \cap \beta) \subseteq \{\text{scalar, subrange}\} \\
&\begin{cases}
-1 & \text{if } e(k, k \cap \beta \cap \alpha) \subseteq \{\text{scalar, subrange}\} \\
&\texttt{scalaror subrange\typeidentical}(\alpha, \beta) \\
&\text{else} \quad \text{halt}
\end{cases}
\end{cases}
\end{align*}

090 \texttt{indextypeofstr}(\alpha) =
\begin{align*}
&\begin{cases}
-1 & \text{if } e(k, k \cap \alpha \cap \beta) \subseteq \text{subrange} \\
&\begin{cases}
\texttt{adjin}(\alpha); \\
&\begin{cases}
\texttt{if } \texttt{sign}(\alpha) \text{ of } k \alpha \text{ eq +1} \\
&\begin{cases}
\texttt{if } 1 \text{ k } \alpha \text{ subset IN} \\
&\texttt{then} \quad \text{halt}
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\end{align*}

091 \texttt{int}(\alpha) =
\begin{align*}
&\begin{cases}
2 & \text{if } e(a, (+, e a (\alpha)) \subseteq \text{IN} \\
&\texttt{con } \texttt{rm}
\end{cases} \\
&\begin{cases}
2 & \text{if } k \#\text{STEMPEL (} (+, e a (\alpha)) \text{)} \\
&\texttt{con } \texttt{rm}
\end{cases} \\
&\texttt{link}(\text{integer in } \text{ATTRIB}, k \#\text{STEMPEL}) \\
&\texttt{else} \quad \text{halt}
\end{align*}

\begin{align*}
&\begin{cases}
2 & \text{if } \alpha = \{\text{unsigned integer}\} \\
&\texttt{con } \texttt{rm}
\end{cases}
\end{align*}
092 \ \text{intdiv}(\alpha, \beta, \gamma) = \begin{cases} -1 \\
\text{if } \in (k \in \beta \cap \text{TYPE}) \in \{\text{scalar, subrange}\} \\
\text{then} \\
-1 \\
\text{if } \in (k \in \gamma \text{ TYPE}) \in \{\text{scalar, subrange}\} \\
\text{then} \\
\text{if } (1 \leq k \leq \text{subset IN}) \land (1 \leq \gamma \leq \text{subset IN}) \\
\text{then} \\
\text{link}(\alpha, \text{integer lm ATTRIB}) \\
\text{else halt} \\
\text{else halt} \\
\text{else halt} \\
\end{cases} \\
\text{if } \alpha = \text{prev}(\text{prev}(\#ATEMP)), \beta = \text{prev}(\#ATEMP), \gamma = \#STEMP \triangleright

093 \ \text{joins}(\alpha, \beta) = \begin{cases} \\
\text{if } k \alpha \text{ ineq } \emptyset \\
\text{then} \\
\text{while } k \alpha \text{ ineq } k \beta \text{ do} \\
\text{link}(k \beta, k \alpha); \\
\text{forwards}(\alpha, \alpha) \\
\end{cases} \\
\text{if } \alpha = \#IDTEMP \text{ of ATTRIB}, \beta = \#ATEMP \triangleright

094 \ \text{kar}(\alpha) = \begin{cases} \\
\text{if } \alpha \in \text{CHAR} \\
\text{then} \\
\text{if } k \#STEMP, \alpha! \\
\text{link}(\text{char lm ATTRIB}, k \#STEMP) \\
\text{else halt} \\
\end{cases} \\
\text{if } \alpha = \langle\text{char} \rangle \triangleright
095 \texttt{latersub}(a) = \\
\textbf{if} (\texttt{#LATER} \neq n \land \texttt{#STEMP}) \land (\texttt{#LATER} \neq n \land \alpha \text{ of STORE}) \\
\textbf{then} \\
\quad \text{parformal}(a) \\
\textbf{else} \\
\quad \textbf{if} \ #\texttt{LATER} \in n \land \alpha \text{ of STORE} \\
\textbf{then} \\
\quad \quad \text{loc}(\texttt{#STEMP}); \\
\quad \quad \text{link}(\texttt{#STEMP}, \bot \text{ of } k \text{ prev}(\texttt{#STEMP}) ; \\
\quad \quad \text{at}(\texttt{#ATEMP}); \\
\quad \quad \quad -1 \\
\quad \quad \text{link}(\texttt{#ATEMP}, k \kern 1em \#\texttt{STEMP} \cap \texttt{ATTRIB}); \\
\quad \quad \text{loc}(\texttt{#STEMP}); \\
\quad \quad \text{link}(\texttt{#STEMP}, \alpha \text{ of } \texttt{STORE}); \\
\quad \quad \text{makeformalsub}; \\
\quad \quad \text{trim}(\texttt{STORE}, \text{prev}(\texttt{#STEMP}) ; \\
\quad \quad \text{delr}(\texttt{ATTRIB} \cap \texttt{#ATEMP}) \\
\textbf{else} \\
\quad \textbf{if} \ #\texttt{LATER} \in n \land k \#\texttt{STEMP} \\
\textbf{then} \\
\quad \quad \text{loc}(\texttt{#STEMP}); \\
\quad \quad \text{link}(\texttt{#STEMP}, \bot \text{ of } \alpha \text{ of } \texttt{STORE}); \\
\quad \quad \text{at}(\texttt{#ATEMP}); \\
\quad \quad \quad -1 \\
\quad \quad \text{link}(\texttt{#ATEMP}, k \kern 1em \#\texttt{STEMP} \cap \texttt{ATTRIB}); \\
\quad \quad \text{loc}(\texttt{#STEMP}); \\
\quad \quad \text{link}(\texttt{#STEMP}, \text{prev}(\texttt{#STEMP}) ; \\
\quad \quad \text{makeformalsub}; \\
\quad \quad \text{trim}(\texttt{STORE}, \text{prev}(\texttt{#STEMP}) ; \\
\quad \quad \text{delr}(\texttt{ATTRIB} \cap \texttt{#ATEMP}) \\
096 \texttt{leorge}(\alpha, \beta) = \\
\quad \text{explicit}(\alpha, \beta); \\
\quad -1 \\
\textbf{if} \ e(k \ k \beta \cap \texttt{TYPE}) \text{ eq } \texttt{set} \\
\textbf{then} \\
\quad -1 \\
\textbf{if} \ e(k \ k \beta \cap \texttt{TYPE}) \text{ eq } \texttt{set} \\
\textbf{then} \\
\quad \text{setiden}(\alpha, \beta) \\
\textbf{else } \text{halt} \\
\textbf{else} \\
\quad \text{leorge}(\alpha, \beta) \\
\quad \quad \alpha = \text{prev}(\texttt{#ATEMP}) \cap \beta = \texttt{#ATEMP} \lor 

097 \textbf{linktovars}(\alpha, \beta) = \\
\texttt{at}(\text{TATEMP}); \\
\texttt{link}(\text{TATEMP}, \text{\#T of ATTRIB}); \\
\texttt{memory}(\text{TATEMP}, \beta); \\
\texttt{repeat} \\
\texttt{link}(\text{\#T of ATTRIB}, \text{k} \alpha); \\
\texttt{trans}(\text{k} \#\text{STEMP}, \beta \#\text{STORE}); \\
\texttt{link}(\text{\#T of ATTRIB}, \text{k} \alpha); \\
\texttt{forward}(\alpha) \\
\texttt{);} \\
\texttt{forward}(\alpha) \\
\texttt{until k \alpha eq \emptyset} \\
\texttt{delr(ATTRIB, \#ATEMP)} \\
\texttt{? \alpha = \#STEMP, \beta = \#MEM} \texttt{?} \\

098 \textbf{loopfixed}(\alpha, \beta) = \\
\texttt{-1 while e k \beta \not\subseteq \{\alpha, \#VP\} do} \\
\texttt{rforward(\beta)} \\
\texttt{? \alpha = (identifier), \beta = \#STEMP} \texttt{?} \\

099 \textbf{loopvariants}(\alpha, \beta) = \\
\texttt{-1 while e k \beta \text{ ineq } \alpha do} \\
\texttt{rforward(\beta)} \\
\texttt{? \alpha = (identifier), \beta = \#STEMP} \texttt{?}
100 \texttt{larg}(\alpha, \beta) = \\
\texttt{explicit}(\alpha, \beta); \\
\texttt{if } e(k \alpha \cap \text{TYPE}) = \text{array} \\
\texttt{then} \\
\texttt{if } e(k \beta \cap \text{TYPE}) = \text{array} \\
\texttt{then} \\
\texttt{strgiden}(\alpha, \beta) \\
\texttt{else} \texttt{halt} \\
\texttt{else} \\
\texttt{if } e(k \alpha \cap \text{TYPE}) \in \{\text{scalar, subrange}\} \\
\texttt{then} \\
\texttt{if } e(k \beta \cap \text{TYPE}) \in \{\text{scalar, subrange}\} \\
\texttt{then} \\
\texttt{adjsin}(\alpha, \beta); \\
\texttt{if } (1 \leq k \alpha \text{ ineq } 1 \leq k \beta) \Delta \\
((1 \leq k \alpha \text{ subset IN}) \lor (1 \leq k \beta \text{ subset IN})) \Delta \\
((1 \leq k \alpha \text{ subset IN}) \lor (1 \leq k \beta \text{ ineq R})) \Delta \\
((1 \leq k \alpha \text{ ineq R}) \lor (1 \leq k \beta \text{ subset IN})) \\
\texttt{then} \texttt{halt} \\
\texttt{else} \texttt{halt} \\
\texttt{else} \texttt{halt} \\
\alpha = \text{prev}(\#ATEMP); \beta = \#ATEMP \& \& \\

101 \texttt{makeformalsub} = \\
\texttt{while } k \texttt{ prev}(\#STEMP) \texttt{ ineq } \varnothing \texttt{ do} \\
\texttt{(formalsub}(\#STEMP); \\
\texttt{forward}(k \texttt{ prev}(\#STEMP)); \\
\texttt{adj}((\#ATEMP, k \#ATEMP, k \texttt{ prev}(\#STEMP) \cap \text{ATTRIS})) \\
\texttt{delr}(k \#STEMP, \#LATER); \\

102 \texttt{maxwidth}(\alpha, \beta, Y) = \\
\texttt{2} \\
\texttt{1. } k \beta, \varepsilon \Delta (e \alpha, e \beta); \\
\texttt{1. } \texttt{if } e \beta \subseteq \text{copyr}(\text{IN}, e \beta, Y) \\
\texttt{then} \\
\texttt{1. } k \beta, \varepsilon \Delta k \beta \\
\alpha = \#VARWIDTH \texttt{ in STORE, } = \#FIXWIDTH \texttt{ in STORE} \\
\beta = \texttt{next}(\#FIXWIDTH \texttt{ in STORE}) \& \& \\

103 \texttt{maybeformalsub}(\alpha) = \\
\texttt{if } n \leq k(\alpha \text{ of STORE}) \texttt{ ineq } \#LATER \\
\texttt{then} \\
\texttt{delr}(\alpha \text{ of STORE}, \#LATER) \\
\texttt{else} \\
\texttt{if } k(\alpha \text{ of STORE}) \texttt{ ineq } () \\
\texttt{then} \texttt{halt} \\
\alpha = \text{identifier} \& \&
maylatsersub(\(a\)) =
\[
\begin{align*}
&\text{if } \#\text{LATER } \not\in \{1 \text{ k } \#\text{STEMP} \\
&\text{then} \quad \text{parformal}(\(a\)) \\
&\text{else} \\
&\quad \text{loc}(\#\text{STEMP}); \\
&\quad \text{link}(\#\text{STEMP}, \#1 \text{ of } \alpha \text{ of } \text{STORE}); \\
&\quad \text{at}(\#\text{ATEMP}); \\
&\quad \text{link}(\#\text{ATEMP}, \# \text{STEMP} \cup \text{ATTRIB}); \\
&\quad \text{loc}(\#\text{STEMP}); \\
&\quad \text{link}(\#\text{STEMP}, \text{prev}(\text{prev}(\#\text{STEMP}))); \\
&\quad \text{makeformalsub} ; \\
&\quad \text{trimr}(\text{STORE}, \text{prev}(\#\text{STEMP})); \\
&\quad \text{delr}(\text{ATTRIB}, \#\text{ATEMP}); \\
&\end{align*}
\]
\[\alpha = (\text{identifier}) \exists\]

memory(\(\alpha, \beta\)) =
\[
\begin{align*}
&\text{explicit}(\(\alpha\)); \\
&\quad \text{sl } \# \beta \text{ lm } \text{STORE}, \# \text{is } (); \\
&\quad \text{augr}(\text{STORE}, \#\text{WIDTH}); \\
&\quad \text{sl } \# \beta \text{ WIDTH} \cup \#1 \text{ of } \text{WIDTH of } \# \alpha; \\
&\quad \text{while } \# \text{WIDTH} \cup \text{STORE}, \# \text{is neg } 0 \text{ do} \\
&\quad \quad \text{augr}(\# \beta \text{ lm } \text{STORE}, \#); \\
&\quad \quad \text{sl } \# \beta \text{ WIDTH} \cup \text{STORE}, \# \text{ prd } \# \text{ WIDTH} \cup \text{STORE} \\
&\quad \quad \text{delr}(\text{STORE}, \#\text{WIDTH}); \\
&\quad \# \alpha = \#\text{ATEMP}, \# \beta = \#\text{MEM } \exists\]

morechar(\(\alpha\)) =
\[
\begin{align*}
&\text{if } \alpha \in \text{CHAR} \text{ then} \quad 2 \\
&\quad \text{if } \# \text{WIDTH} \uparrow \text{of } \text{STORE} \in \# \text{MAXCHAR} \\
&\quad \quad \text{scc} \text{ then} \quad 2 \\
&\quad \quad \quad \text{sl } \# \text{WIDTH} \uparrow \text{of } \text{STORE} \text{ e } \# \text{MAXCHAR} \text{ scc} \\
&\quad \quad \quad \quad \text{augr}(\text{DEF of } \text{prev}(\# \text{T of } \text{ATTRIB}), \# \text{WIDTH} \uparrow \text{of } \text{STORE}) \text{ con} \text{ scc} \\
&\quad \quad \quad \quad \quad \text{augr}(\# \text{STEMP}, \alpha) \\
&\quad \quad \quad \quad \text{else halt} \\
&\quad \quad \quad \text{else halt} \\
&\quad \# \alpha = (\text{char}) \exists
\end{align*}
\]
107 moreindex =
   s1 k(#DIMEN lm STORE), # DIMEN lm STORE);
   \ e \ k(#DIMEN lm STORE);
   if e k(#DIMEN lm STORE) eq -1
      then halt

108 morepar =
   \ e \ k \ prev(#STEMP) n OBJECT\{formal function, formal procedure\}
   then
      (if #LATER \ e \ n \ k \ prev(#STEMP)
         then
            forward(#STEMP);
            correctpar(#STEMP);
            \ e \ adj1(#ATEMP, k \ k \ #STEMP \ n \ ATTRIB)
         )
      trans(forward(#STEMP);

      \ e \ adj2(k \ #STEMP, k \ #STEMP \ n \ ATTRIB, #STEMP)
   )

   else
      if k prev(#STEMP) \ e \ STANDP
      then
         forward(#STEMP);
         correctpar(#STEMP);
         trans(forward(#STEMP));

      \ e \ adj1(#ATEMP, k \ #STEMP \ n \ ATTRIB)
      \ e \ adj2(k \ #STEMP, k \ #STEMP \ n \ ATTRIB, #STEMP)

109 moresettype(\alpha, \beta) =
   at(#ATEMP);
   link(#ATEMP, # of prev(\alpha));
   scalarorsubrangetypeidentical(prev(#ATEMP), #ATEMP);
   delr(ATTRIB, #ATEMP)
   \ e \ \alpha = prev(#ATEMP), \beta = #ATEMP \ e
110 \text{moresum}(\alpha, \beta) = \\
&\text{at}(\#\text{ATEMP})! \\
&\text{link}(\#\text{ATEMP}, k \in \emptyset \text{ ATTRIB}); \\
&\text{explicit}(\#\text{ATEMP}); \\
&\text{adjin}(\#\text{ATEMP}); \\
&\text{if} k(\#\text{SUM lm STORE}), \\
&\text{2} \\
&\text{e} \&\text{ (e k(\#\text{SUM lm STORE}),} \\
&\text{1+} \\
&\text{2} \\
&\text{e} \&\text{ ( +, ordinal(e k \beta \text{ of k #ATEMP) con} \\
&\text{of k #ATEMP})} \\
&\text{)} \\
&\text{)} \\
&\text{then} \\
&\text{if e k \beta \in n \in \#\text{ATEMP} then} \\
&\text{if e k \beta \in n \in \#\text{ATEMP} then} \\
&\text{2} \\
&\text{e} \&\text{ (e k(\#\text{SUM lm STORE}),} \\
&\text{1+} \\
&\text{2} \\
&\text{e} \&\text{ ( +, ordinal(e k \beta \text{ of k #ATEMP) con} \\
&\text{of k #ATEMP})} \\
&\text{)} \\
&\text{else} \text{halt}; \\
&\text{delr}(\text{ATTRIB, #ATEMP}) \\
&\text{2} \alpha = \text{prev(\#STEMP)}, \beta = \#\text{STEMP} \alpha \\

111 \text{morevar}(\alpha) = \\
&\text{loc}(\alpha); \\
&\text{link(\text{variable of OBJECT, } \alpha \text{ of STORE});} \\
&\text{trans(loc}(\alpha)); \\
&\text{link(\text{variable of OBJECT, } \alpha \text{ of STORE})} \\
&\text{2} \alpha = \langle \text{identifier} \rangle \alpha \\

112 \text{morewidth}(\alpha, \beta) = \\
&\text{2} \\
&\text{e k e } \&\text{ (e k \alpha e } \&\text{ ( +, +, \text{ of WIDTH of k } \beta) con} \\
&\text{)} \\
&\text{2} \alpha = \#\text{FIXWIDTH of STORE}, \beta = \#\text{ATEMP} \alpha
moveblock =
movehead

trans(sI k $ of STORE, e k #STEMP) 2

link(k k #STEMP & OBJECT, $ of STORE)

if k k #STEMP & ATTRIB ineq £

trans(fptidtrans(e n k k #STEMP & ATTRIB),

(g n $ of STORE) of STORE)

if l k #STEMP ineq ()

loc(#STEMP);

link(#STEMP, k prev(#STEMP));

adjin(#STEMP);

trans(loc(#STEMP));

link(#STEMP, $ of STORE);

adjin(#STEMP)

parstrans(#STEMP);

trans(delr(STORE, #STEMP));

delr(STORE, #STEMP)

movehead =

loc(#STEMP);

link(#STEMP, k prev(#STEMP));

trans(loc(e n k #STEMP is ()))

adjin(#STEMP);

while k #STEMP ineq £ do

trans(augr($ of STORE, e n k #STEMP));

forward(#STEMP)

); delr(STORE, #STEMP)

newsubname(α, β) =

idlimit(α);

idunique(α);

loc(α is ());

link(β of OBJECT, α of STORE);

sl k(α of STORE), TEMP is ( );

link(#STEMP, α of STORE)

α α = (identifier), β = procedure, function
newtype(α, β) =

-1

link(k α ∩ TYPE, k β)

if e(k α ∩ TYPE) eg type identifier
then
  newtypeid(α, β)
else
  k β is 1 e k α;
  othnewtype(α, β)
/

α = prev(#ATEMP), β = #ATEMP ⊙

newtypeid(α, β) =

2

if ordinal(k α ∩ ATTRIB)
  copy1(UINT, ordinal(# of ATTRIB))
then
  2
  link(k β, k α ∩ ATTRIB)
else
  insertr(ATTRIB, k #LATER of #TEMP; #T);
  2
  link(k β, next(k #LATER of #TEMP));
  at(#ATEMP);  2
  link(#ATEMP, (k prev(α)) ∩ ATTRIB);
  at(#ATEMP);
  link(#ATEMP, (k prev(prev(β))) ∩ ATTRIB);
  newtype(prev(#ATEMP), #ATEMP);
  trimr(ATTRIB, prev(#ATEMP))
/

α = prev(#ATEMP), β = #ATEMP ⊙

nonreal(α) =

-1

if e(k k α ∩ TYPE) E (scalar, subrange)
then
  if k α eg real in ATTRIB
  then halt
else halt
/

α = #ATEMP ⊙

numconst(α, β) =

2

if e (α, e k β) E DEF of (k k β ∩ ATTRIB)
sg
then
  2
  si k β, e  α (e k α)
sg
else halt
/

α = (sign), β = #TEMP ⊙
120 offsetsunique(α, β) =
  forwards(α, β);
  if e $1 of k α eq e $1 of k β
  then
    forwards(α, β)
  else
    halt
  α = prev(ATEM), β = #ATEM

121 oldsubname(α, β) =
  link(#STEMP, α of STORE);
  1
  if e (k k #STEMP $ object) eq β
  then
    link(STEMP, α of $1 of k #STEMP)
    if #AHEAD num 1 k #STEMP
    then
      halt
    else
      halt
  α = {identifier}, β = procedure, function

122 othnewtype(α, β) =
  if (k α $ object type) $ {scalar, subrange}
  then
    augl(ATTRIB, #IDTEMP);
    at(ATEM);
    link(ATEM, $1 of k prev(α))
    at(ATEM);
    link(ATEM, $1 of k prev(prev(ATEM)));
    contnewtype(#IDTEMP, prev(ATEM), #ATEM);
    dell(ATTRIB, #IDTEMP);
    trimr(ATTRIB, prev(#ATEM))
  α = prev(ATEM), β = #ATEM
123 \text{paraction} = \begin{aligned}
\text{log}(e \circ k \#\text{STEMP}); \\
\text{link}(k \circ k \#\text{STEMP} \cap \text{OBJECT}, \circ \#\text{STORE}); \\
\text{if } e(k \circ k \#\text{STEMP} \cap \text{OBJECT}) = \text{eq formal value, then} \\
\text{fi} \end{aligned}

124 \text{parameters}(\alpha) = \begin{aligned}
\text{adjin}(\alpha); \\
\text{while } k \alpha \text{ ineq } \emptyset \text{ do } \\
\text{paraction; (paraction; forward}(\alpha) \\
\text{break}(\alpha = \#\text{STEMP}) \end{aligned}
125 \texttt{parformal}(\alpha) = \\
\texttt{loc}(\#STEMP); \\
\texttt{link}(\#STEMP, \#1 \text{ of } \alpha \text{ of } \text{STORE}); \\
\texttt{at}(\#ATEMP); \\
-1 \\
\texttt{link}(\#ATEMP, k \#STEMP \cap \text{ATTRIB}); \\
\texttt{loc}(\#STEMP); \\
\texttt{link}(\#STEMP, \#1 \text{ of } k \text{ prev}(\text{prev}(\#STEMP))); \\
\texttt{at}(\#ATEMP); \\
-1 \\
\texttt{link}(\#ATEMP, k \#STEMP \cap \text{ATTRIB}); \\
\texttt{while}(k \text{ prev}(\#STEMP) \text{ ineq } \emptyset) \land (k \#STEMP \text{ ineq } \emptyset) \text{ do } \\
-1 \\
\texttt{if } k \text{ prev}(\#STEMP) \cap \text{OBJECT eq formal value} \\
\texttt{then} \\
\texttt{typecheck}(\text{prev}(\#ATEMP), \#ATEMP) \\
\texttt{else} \\
\texttt{halt}; \\
\texttt{forwards}(\text{prev}(\#STEMP), \#STEMP); \\
\texttt{adj}(\text{prev}(\#ATEMP), k \text{ prev}(\#ATEMP), \\
-1 \\
\texttt{k \text{ prev}(\#STEMP) \cap \text{ATTRIB})}; \\
-1 \\
\texttt{adj}(\#ATEMP, k \#ATEMP, k \#STEMP \cap \text{ATTRIB}) \\
\texttt{if } (k \text{ prev}(\#STEMP) \text{ ineq } \emptyset) \lor (k \#STEMP \text{ ineq } \emptyset) \\
\texttt{then} \\
\texttt{halt}; \\
\texttt{trim}(\text{STORE}, \text{prev}(\#STEMP)); \\
\texttt{trim}(\text{ATTRIB}, \text{prev}(\#ATEMP)); \\
\texttt{a } \alpha = (\text{identifier}) \ a \\

126 \texttt{parkind}(\alpha) = \\
\texttt{loc}(\#STEMP); \\
\texttt{link}(\#STEMP, k \text{ prev}(\#STEMP)); \\
\texttt{repeat} \\
\texttt{link}(\alpha \text{ of } \text{OBJECT}, k \#STEMP); \\
\texttt{forward}(\#STEMP) \\
\texttt{until } k \#STEMP \text{ eq } \emptyset; \\
\texttt{delr}(\text{STORE}, \#STEMP); \\
\texttt{a } \alpha = \text{formal value, formal variable, formal procedure } \\
\texttt{formal function a}
parord=
  if \#LATER \&\& \#STEMP
    then
      if \#STEMP eq (x)
        then
          at(#ATEMP);
          link(#ATEMP, k (x of \#STEMP) \& \#ATTRIB);
          explicit(#ATEMP);
        else
          if \#STEMP in \{scalar, subrange\}
            then
              if \#STEMP \&\& real \#ATTRIB
                then
                  delr(#ATTRIB, \#ATEMP)
                else
                  halt
              else
                halt
            else
              halt
          else
            halt
        else
          halt
parstrans(a) =
   while \( k \neq \text{ineq} \neq \text{do} \)
   ( trans(link(k #STEPM \text{OBJECT}, k #STEPM) )
     \[ \Rightarrow \]
     (if e(k #STEPM \text{OBJECT}) eq formal value
        then
        \[ \Rightarrow \]
        at(#ATEMP); 
        link(#ATEMP, k #STEPM \text{ATTRIB});
        augl(STORE, #MEM); 
        memory(#ATEMP, #MEM); 
        trans(s1 k #STEPM, e k(#MEM \text{STORE}));
        dell(STORE, #MEM); 
        delr(ATTRIB, #ATEMP); 
        trans(fptidtrans(e n(k k #STEPM \text{ATTRIB}),
        \[ \Rightarrow \]
        (e n k #STEPM) \text{of k #STEPM})
       
     )
   )
   else
   (if e(k #STEPM \text{OBJECT}) \notin 
      \{\text{formal variable, formal function} \}
      then 
      (trans(fptidtrans(e n(k k #STEPM \text{ATTRIB}),
      \[ \Rightarrow \]
      (e n k #STEPM) \text{of k #STEPM})
     
   )
   )
   trans(forward(#STEPM));
   forward(#STEPM)
   \[ \Rightarrow \]
   \( a \alpha = \#STEPM \ a \)
129  \( \text{partype}(\alpha) = \)

\[
\begin{align*}
\text{if } & \not\in \text{copyr}(\text{ATTRIB}, k) \# \text{LATER of } k \alpha \text{ then } \\
\text{link}(k \# \text{ATEMP}, x \ of \ k \alpha) \;
\text{else} \\
\text{at}(\# \text{ATEMP}); \\
\text{link}(\# \text{ATEMP}, \text{next}(k \# \text{LATER of } k \alpha \ of \ \text{ATTRIB}); \\
\text{while } e \ k \# \text{ATEMP} \text{ ineq } \phi \ do \\
\text{forward}(\# \text{ATEMP}); \\
\text{sl} \ k \# \text{ATEMP}, \#; \\
\text{delr}(\text{ATTRIB}, \# \text{ATEMP}); \\
\text{if } \text{ordinal}(k \# \text{ATEMP}) \\
\text{then} \\
\text{link}(k \# \text{ATEMP}, x \ of \ k \alpha); \\
\text{sl} \#; \text{ of } \text{ATTRIB}, \\
\text{else} \\
\text{at}(\# \text{ATEMP}); \\
\text{link}(\# \text{ATEMP}, k \# \text{LATER of } k \alpha); \\
\text{insertr}(\text{ATTRIB}, k \# \text{ATEMP}, \#T); \\
\text{forward}(\# \text{ATEMP}); \\
\text{link}(k \# \text{ATEMP}, x \ of \ k \alpha); \\
\text{newtype}(\text{prev}(\# \text{ATEMP}), \# \text{ATEMP}); \\
\text{delr}(\text{ATTRIB}, \# \text{ATEMP}); \\
\text{sl} \#; \text{ of } \text{ATTRIB}, \\
\end{align*}
\]

\( \alpha = \text{prev}(\# \text{STEMP}), \# \text{STEMP} \)  

\( 130 \) \text{pointeriden}(\alpha, \beta) =

\[
\begin{align*}
\text{if } \not\in \text{ANYPOINTER of } \text{ATTRIB} \not\in (k \alpha, k \beta) \text{ then } \\
\text{adj} = \text{in}(\alpha, \beta); \\
\text{identical}(\alpha, \beta) \\
\text{\( \alpha = \text{prev}(\# \text{ATEMP}), \beta = \# \text{ATEMP} \)  
\end{align*}
\]

\( 131 \) \text{procaction} =

\[
\begin{align*}
\text{loc}(\# \text{STEMP}); \\
\text{link}(\# \text{STEMP}, k \text{ prev}(\# \text{STEMP}) ) \\
\text{parameters}(\# \text{STEMP}); \\
\text{delr}(\text{STORE}, \# \text{STEMP}) \\
\end{align*}
\]
ptridchk =
   if \#TMPPTR \in \text{ATTRIB}
   then
      (if \$1 \text{of}(\#TMPPTR \in \text{ATTRIB}) \in \text{ATTRIB}
       then
         \text{link} \, k, \#TEMPPTR,
       of \text{ATTRIB}
      )
     \text{dell}(\text{ATTRIB}, \#TEMPPTR)
   else \text{halt};
   \text{ptridchk}

\text{putifthen}(\alpha) =
   \text{trans}(\text{if} \, e \, k, \#\text{TEMP} \, \text{eq} \, \text{true}
   \text{then} \, e = \, 1, \, k, \alpha)
   \Theta \alpha = \#\text{BPTR1} \, \text{of} \, \text{STORE} \, \Theta

\text{putifthenelse}(\alpha, \beta) =
   \text{trans}(\text{if} \, e \, k, \#\text{TEMP} \, \text{eq} \, \text{true}
   \text{then} \, e = \, 1, \, k, \alpha
   \text{else} \, e = \, 1, \, k, \beta)
   \Theta \alpha = \#\text{BPTR1} \, \text{of} \, \text{STORE}, \beta = \#\text{BPTR2} \, \text{of} \, \text{STORE} \, \Theta

\text{radditional}(\alpha, \beta, \gamma, \delta) =
   \text{rexlicit}(\beta, \delta);\text{if}(e \, k, \beta \in \text{IN}) \lor (e \, k, \beta \in \text{R})
   \text{then}
   \text{rnunadd}(\alpha, \beta, \gamma, \delta)
   \text{else}
   \text{if} \gamma \, \text{eq} \, \text{or}
   \text{then}
   \text{ror}(\alpha, \beta, \delta)
   \text{else}
   \text{rset}(\alpha, \beta, \gamma, \delta)
   \Theta \alpha = \text{prev(prev(#\text{TEMP})),} \beta = \text{prev(#\text{TEMP}),}
   \gamma = \langle \text{additional operator} \rangle, \delta = \#\text{TEMP} \, \Theta

\text{radjim}(\alpha) =
   -1
   \text{adj2}(k, \alpha, \$1 \, \text{of} \, k, \alpha, \alpha)
   \Theta \alpha = \#\text{TEMP} \, \Theta
137 \( \text{rand}(\alpha, \beta, \gamma) = \)
\begin{align*}
\text{link} & (\text{Boolean lm \texttt{ATTRIB}}) \alpha, \\
\text{if} & (e \land \beta \texttt{eq true}) \land (e \land \gamma \texttt{eq true}) \\
\text{then} & \\
\text{si} & k \alpha \texttt{, true} \\
\text{else} & \\
\text{si} & k \alpha \texttt{, false} \\
\end{align*}
\begin{align*}
\alpha & = \text{prev(prev(#STEMP))}, \\
\beta & = \text{prev(#STEMP)} , \\
\gamma & = \text{#STEMP} \ \triangleright \\
\end{align*}

138 \( \text{rarray}(\alpha) = \)
\begin{align*}
\text{rexplicit}(\alpha); \\
\text{radjinv}(\alpha) \\
\end{align*}
\begin{align*}
\alpha & = \text{#STEMP} \ \triangleright \\
\end{align*}

139 \( \text{rbackward}(\alpha) = \)
\begin{align*}
-1 \\
\text{adj2}(k \alpha, \text{prev}(k \alpha), \alpha) \\
\end{align*}
\begin{align*}
\alpha & = \text{#STEMP} \ \triangleright \\
\end{align*}

140 \( \text{rcomponexp}(\alpha, \beta, \gamma, \delta) = \)
\begin{align*}
\text{rexplicit}(\beta, \delta); \\
-2 \\
\text{if} & e(k \beta \texttt{ \& \ TYPE}) \texttt{ eq array} \\
\text{then} & \\
\text{augl}(\text{STORE}, \texttt{#WIDTH}); \\
-1 \\
\text{si} & k(\texttt{#WIDTH lm STORE}), e \#1 \texttt{ of WIDTH of } k \beta \\
-1 \\
\text{adj2}(k \beta, \texttt{#T of } k \beta); \\
\text{rexplicit}(\beta); \\
\text{rstrgrel}(\alpha, \beta, \gamma, \delta); \\
\text{dell}(\text{STORE}, \texttt{#WIDTH}) \\
\text{else} & -2 \\
\text{if} & e(k \beta \texttt{ \& \ TYPE}) \in \{\text{scalar, subrange}\} \\
\text{then} & \\
\text{rsrgrel}(\alpha, \beta, \gamma, \delta) \\
\text{else} & -2 \\
\text{if} & e(k \beta \texttt{ \& \ TYPE}) \texttt{ eq pointer} \\
\text{then} & \\
\text{rptrrel}(\alpha, \beta, \gamma, \delta) \\
\text{else} & \\
\text{rsrel}(\alpha, \beta, \gamma, \delta) \\
\end{align*}
\begin{align*}
\alpha & = \text{prev(prev(#STEMP))}, \\
\beta & = \text{prev(#STEMP)} , \\
\gamma & = \{\text{relational operator}\}, \delta = \text{#STEMP} \ \triangleright \\
\end{align*}
141 \text{readpar}(\alpha) = \text{explicit}(\alpha); \neg 1

\text{if } e(k \in \text{TYPE}) \in \{\text{scalar}, \text{subrange}\} \text{ then}
\begin{align*}
\text{if } (\text{DEF of } k \in \text{subset CHAR}) \land (\text{DEF of } k \in \text{subset IN}) \land (\text{DEF of } k \in \text{ineq R}) \text{ then halt}
\text{else halt}
\end{align*}
\text{\text{\begin{equation*} \alpha = \text{#ATEMP} \end{equation*}}}

142 \text{real}(\alpha) =
\begin{align*}
&2 \\
&\text{if } e \in \mathcal{A} (\text{+1e } e \in (\alpha)) \\
&\text{then}
\begin{align*}
&2 \\
&\text{if } k \text{ #STEMP} e \in \mathcal{A} (\text{+1e } e \in (\alpha)) \\
&\text{then}
\end{align*}
\end{align*}
\text{\text{\begin{equation*} \alpha = \text{\begin{array}{c}
\text{real } \text{IN}\text{ ATTRIB, k #STEMP}
\end{array}} \end{equation*}}}
\text{\text{\begin{equation*} \alpha = \text{\begin{array}{c}
\text{unsigned real}\end{array}} \end{equation*}}} \ \text{\begin{equation*} \alpha \end{equation*}}

143 \text{realdiv}(\alpha, \beta, \gamma) = \neg 1

\text{if } e(k \in \text{TYPE}) \in \{\text{scalar}, \text{subrange}\} \text{ then}
\begin{align*}
&\neg 1 \\
&\text{if } e(k \in \text{TYPE}) \in \{\text{scalar}, \text{subrange}\} \text{ then}
\begin{align*}
&\text{if } (\text{DEF of } k \in \text{subset IN}) \lor (\text{DEF of } k \in \text{subset IN}) \lor (\text{DEF of } k \in \text{ineq R}) \lor (\text{DEF of } k \in \text{ineq R}) \text{ then}
\text{\begin{equation*} \text{\begin{array}{c}
\text{link}(\alpha, \text{real } \text{IN}\text{ ATTRIB})
\end{array}} \end{equation*}} \text{\begin{equation*} \alpha = \text{\begin{array}{c}
\text{prev(\text{#STEMP})}\end{array}} \end{equation*}} \text{\begin{equation*} \beta = \text{\begin{array}{c}
\text{prev(\text{#STEMP})}\end{array}} \end{equation*}} \text{\begin{equation*} \gamma = \text{\begin{array}{c}
-\text{#STEMP} \end{array}} \end{equation*}} \text{\begin{equation*} \begin{array}{c}
\end{array}} \end{equation*}} \text{\begin{equation*} \alpha \end{equation*}}
\end{align*}

144 \text{recordiden}(\alpha, \beta) =
\begin{align*}
&\text{if } e \text{ $\%$ of WIDTH of } k \alpha \text{ eq } e \text{ $\%$ of WIDTH of } k \beta \\
&\text{then}
\begin{align*}
&\text{adjsin}(\alpha, \beta); \\
&\text{field}(\alpha, \beta) \\
&\text{else halt}
\end{align*}
\text{\text{\begin{equation*} \alpha = \text{\begin{array}{c}
\text{prev(\text{#STEMP})}\end{array}} \end{equation*}} \text{\begin{equation*} \beta = \text{\begin{array}{c}
\text{#STEMP} \end{array}} \end{equation*}} \text{\begin{equation*} \begin{array}{c}
\end{array}} \end{equation*}} \text{\begin{equation*} \alpha \end{equation*}}
\end{align*}

145 \text{rexplicit}(\alpha, \beta) =
\begin{align*}
&\text{rexplicit}(\alpha); \\
&\text{rexplicit}(\beta)
\end{align*}
\text{\text{\begin{equation*} \alpha = \text{\begin{array}{c}
\text{prev(\text{#STEMP})}\end{array}} \end{equation*}} \text{\begin{equation*} \beta = \text{\begin{array}{c}
\text{#STEMP} \end{array}} \end{equation*}} \text{\begin{equation*} \begin{array}{c}
\end{array}} \end{equation*}} \text{\begin{equation*} \alpha \end{equation*}}
146 \textit{rexplicit}(\alpha) = \\
\quad \text{while } e(k \alpha \cap \text{TYPE}) \text{ eq type identifier} \\
\quad \text{adj2}(k \alpha \cap k \alpha \cap \text{ATTRIB}, \alpha) \\
\quad \alpha = \#STEMP \ \alpha

147 \textit{rforward}(\alpha) = \\
\quad \text{adj2}(k \alpha \cap \text{next}(k \alpha), \alpha) \\
\quad \alpha = \#STEMP \ \alpha

148 \textit{rsub}(\alpha) = \\
\quad \text{rrfind}(\alpha, \#STEMP); \\
\quad \text{link}(k \alpha \cap \#STEMP \cap \text{ATTRIB}, \#STEMP); \\
\quad \text{while } e(k \alpha \cap \#STEMP \cap \text{OBJECT}) \text{ eq formal function do} \\
\quad \text{adj1}(\#STEMP, k \#STEMP, k \#STEMP) \\
\quad \alpha = \text{(identifier) } \alpha

149 \textit{rindexaction}(\alpha) = \\
\quad \text{at}(\#ATEMP); \\
\quad \text{link}(\#ATEMP, k \alpha \cap \text{ATTRIB}); \\
\quad \text{explicit}(\#ATEMP); \\
\quad \text{extrablock}(\#ATEMP); \\
\quad \text{delr}(\text{ATTRIB}, \#ATEMP) \\
\quad \alpha = \#STEMP \ \alpha

150 \textit{rintadd}(\alpha, \beta, \gamma, \delta) = \\
\quad \text{if } \gamma \text{ eq +} \\
\quad \quad \text{then} \\
\quad \quad \quad \text{si } k \alpha, e \neq (e k \beta, e k \delta) \\
\quad \quad \quad \quad \text{i=+} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{si } k \alpha, e \neq (e k \beta, e k \delta) \\
\quad \quad \quad \quad \text{i=} \\
\quad \quad \quad \alpha = \text{prev(prev(\#STEMP))); } \beta = \text{prev(\#STEMP),} \\
\quad \quad \quad \gamma = \text{(additional operator)}; \delta = \#STEMP \ \alpha
151. \texttt{rintdiv}(\alpha; \beta; \gamma) = \\
\text{link}(_{\text{integer}} \text{lm} \text{ ATTRIB; } \alpha) ; \\
\text{if} \ e \ k \ \gamma \not\in \{+0,-0\} \\
\text{then} \\
\quad 2 \\
\quad \text{si} \ k \ \alpha \in \gamma \ (\text{e} k \ \beta \in k \ \gamma) \\
\quad \text{ir} \\
\text{else} \ \\
\quad \text{halt} \\
\theta \ \alpha = \text{prev} (\text{prev} (\text{#TEMP} )) ; \beta = \text{prev} (\text{#TEMP} ) , \gamma = \text{#TEMP} \ \theta

152. \texttt{rintersect}(\alpha; \beta; \gamma) = \\
-1 \\
\text{if} \ k \ \beta \in \#\text{ANYSET of} \ \text{ATTRIB} \\
\text{then} \\
\quad -1 \\
\quad \text{link}(k \ \gamma; \alpha) \\
\text{else} \\
\quad -1 \\
\quad \text{link}(k \ \beta; \alpha) ; \\
\quad \text{si} \ k \ \alpha \in k \ \beta \cup \alpha \ k \ \gamma \\
\theta \ \alpha = \text{prev} (\text{prev} (\text{#TEMP} )) ; \beta = \text{prev} (\text{#TEMP} ) , \gamma = \text{#TEMP} \ \theta

153. \texttt{rintor}(\alpha) = \\
\text{if} \ e \ k \ \alpha \in \text{IN} \\
\text{then} \\
\quad 2 \\
\quad \text{si} \ k \ \alpha \in \gamma \ (\text{e} k \ \alpha) \\
\quad \text{ir} \\
\theta \ \alpha = \text{prev} (\text{#TEMP}) , \text{#TEMP} \ \theta

154. \texttt{rlentvar}(\alpha; \beta) = \\
\text{rlfind}(\alpha; \beta) ; \\
-1 \ 2 \\
\text{link}(k \ k \ \beta \cup \text{ATTRIB; } \beta) ; \\
\text{adj} (\beta \ k \ \beta ; \#1 \text{ of } k \ \beta) \\
\theta \ \alpha = \langle \text{identifier} \rangle ; \beta = \text{#TEMP} \ \theta

155. \texttt{rlfind}(\alpha; \beta) = \\
\text{link}(\beta ; \# \text{ of} \ \text{STORE}) ; \\
\text{while}(\alpha \ \text{ineq} \ e \ k \ \beta) \ \lor \ # \text{OUTPUT FROM } \alpha \ \text{ineq} \ e \ x \ \beta \ \text{do} \\
\text{backward}(\beta) \\
\theta \ \alpha = \langle \text{identifier} \rangle ; \beta = \text{#TEMP} \ \theta
156 \texttt{rlfixedsec(\alpha) =}
\begin{align*}
\text{augl(STORE, #SUM);} \\
\text{rforward(\alpha);} \\
-1 \\
\text{if } k(#SUM \text{ in STORE), e } \text{ $1$ of } k \alpha; \\
\text{findunit(\alpha);} \\
\text{dell(STORE, #SUM)} \\
\end{align*}
\begin{align*}
\alpha &= \text{#STEMP } \alpha
\end{align*}

/ 

157 \texttt{rlselector(\alpha, \beta) =}
\begin{align*}
\text{loopfind(\alpha, \beta);} \\
-1 \\
\text{if } e k \beta \text{ eq } \alpha \\
\text{then } \\
\text{rlfixedsec(\beta);} \\
-1 \\
\text{while } n k \beta \text{ ineq } #T \text{ do } \\
\text{rforward(\beta);} \\
\text{else } \\
\text{augl(STORE, #EXTRA);} \\
\text{caseid(\alpha, \beta);} \\
\text{(if } e k \beta \text{ eq } \alpha \\
\text{then } \\
\text{rlfixedsec(\beta);} \\
\text{while } n k \beta \text{ ineq } #T \text{ do } \\
\text{rforward(\beta);} \\
\text{augl(STORE, #TAGFIELD)} \\
\text{else } \\
\text{loopvariants(\alpha, \beta);} \\
\text{rlvariants(\alpha, \beta); } \\
\text{dell(STORE, #EXTRA)} \\
\end{align*}
\begin{align*}
\alpha &= \text{(identifier), } \beta = \text{#STEMP } \alpha
\end{align*}

158 \texttt{rlvariants(\alpha, \beta) =}
\begin{align*}
-1 \\
\text{while } e k \beta \text{ ineq } #V \text{ do } \\
\text{rbackward(\beta);} \\
\text{rforward(\beta);} \\
\text{if } e k(#EXTRA \text{ in STORE}) \in n l k \beta \\
\text{then } \\
\text{rlselector(\alpha, \beta)} \\
\text{else } \\
\text{rlvariants(\alpha, \beta); } \\
\text{dell(STORE, #EXTRA)} \\
\end{align*}
\begin{align*}
\alpha &= \text{(identifier), } \beta = \text{#STEMP } \alpha
\end{align*}
159 \texttt{rmod}(\alpha, \beta, \gamma) = \\
\texttt{link}(\text{integer } \gamma \text{ ATTRIB}, \alpha); \\
\texttt{if } k \alpha \equiv \gamma \mod \\
\langle \alpha, k \beta, \gamma \rangle \\
\texttt{if } \alpha = \text{prev}(\text{prev}(\#\text{STEMP})), \beta = \text{prev}(\#\text{STEMP}), \gamma = \#\text{STEMP} \text{ then}

160 \texttt{rnot}(\alpha, \beta) = \\
\texttt{explicit}(\beta); \\
\texttt{if } e(k \ beta \in \{\text{scalar, subrange}\} \text{ then} \\
\texttt{if } 1 \text{ DEF of } k \ beta \equiv \text{BOOL} \\
\texttt{then} \\
\texttt{link}(\alpha, k \beta) \\
\texttt{else} \texttt{halt} \\
\texttt{else} \texttt{halt} \\
\texttt{if } \alpha = \text{prev}(\#\text{ATEMP}), \beta = \#\text{ATEMP} \text{ then}

161 \texttt{rnumadd}(\alpha, \beta, \gamma, \delta) = \\
\texttt{if } e(k \ beta \in \text{IN} \wedge (e k \delta \in \text{IN}) \text{ then} \\
\texttt{link}(\text{integer } \gamma \text{ ATTRIB}, \alpha); \\
\texttt{rintadd}(\alpha, \beta, \gamma, \delta) \\
\texttt{else} \\
\texttt{link}(\text{real } \gamma \text{ ATTRIB}, \alpha); \\
\texttt{rinttor}(\beta); \\
\texttt{rinttor}(\delta); \\
\texttt{rrealadd}(\alpha, \beta, \gamma, \delta); \\
\texttt{link}(\text{real } \delta \text{ ATTRIB}, \alpha) \\
\texttt{if } \alpha = \text{prev}(\text{prev}(\#\text{STEMP})), \beta = \text{prev}(\#\text{STEMP}), \gamma = \text{(additional operator) \wedge \delta = \#\text{STEMP} \text{ then}

162 \texttt{ror}(\alpha, \beta, \gamma) = \\
\texttt{link}(\text{Boolean } \gamma \text{ ATTRIB}, \alpha); \\
\texttt{if } e(k \ beta \equiv \text{true}) \lor (e k \gamma \equiv \text{true}) \\
\texttt{then} \\
\texttt{\#1 k \alpha, \text{true} \\
\texttt{else} \\
\texttt{\#1 k \alpha, \text{false} \\
\texttt{if } \alpha = \text{prev}(\text{prev}(\#\text{STEMP})), \beta = \text{prev}(\#\text{STEMP}), \gamma = \#\text{STEMP} \text{ then}
163 \text{rparlesscalif}(\alpha) =
\text{while } e(k \ k \alpha \cap \text{OBJECT}) \text{ eg formal function do}
  \text{adj}(\alpha, k \alpha, k \alpha);
  \text{if } k \alpha \text{ eq eol in lm STORE then}
  \text{eol}(\alpha, \text{eoln})
  \text{else}
  \text{if } k \alpha \text{ eq eof lm STORE then}
  \text{eol}(\alpha, \text{eof})
  \text{else}
  2
  \text{e } 1 \text{ } k \alpha
\alpha = \#\text{TEMP } \alpha
\text{end}

164 \text{rpointer}(\alpha) =
  \text{adj}(\alpha, k \alpha, k \alpha \cap \text{HEAP});
  \text{adj}(\alpha)
\alpha = \#\text{TEMP } \alpha

165 \text{rpsub}(\alpha) =
  \text{rrfind}(\alpha, \#\text{TEMP});
  \text{while } e(k \ k \#\text{TEMP} \cap \text{OBJECT}) \text{ eg formal procedure do}
  \text{adj}(\#\text{TEMP}, k \#\text{TEMP}, k \#\text{TEMP})
\alpha = \{\text{identifier}\} \alpha

166 \text{rrtrrel}(\alpha, \beta, \gamma, \delta) =
  \text{if } \gamma \text{ eq } =
  \text{then}
  -1
  \text{if } k \ k \beta \cap \text{HEAP} \text{ eg } k \ k \delta \cap \text{HEAP}
  \text{then}
  \text{sl } k \alpha, \text{true}
  \text{else}
  \text{sl } k \alpha, \text{false}
  \text{else}
  -1
  \text{if } k \ k \beta \cap \text{HEAP} \text{ ineq } k \ k \delta \cap \text{HEAP}
  \text{then}
  \text{sl } k \alpha, \text{true}
  \text{else}
  \text{sl } k \alpha, \text{false}
\alpha = \text{prev}(\text{prev}(\#\text{TEMP})), \beta = \text{prev}(\#\text{TEMP}),
\gamma = \{\text{relational operator}\}, \delta = \#\text{TEMP } \alpha
167 \texttt{rldiv}(\alpha, \beta, \gamma) = \\
\quad \text{link\{real im ATTRIB,}\alpha\}; \\
\quad \text{rinttor}(\beta); \\
\quad \text{rinttor}(\gamma); \\
\quad \text{if } e \cdot k \cdot \gamma \in \{+0E+0, +0E-0, -0E+0, -0E-0\} \\
\quad \text{then} \\
\qquad 2 \\
\qquad \text{s1 k } \alpha, e, \beta, e, \gamma \\
\quad \text{else} \text{ halt} \\
\quad @ \alpha=\text{prev(prev(#STEMP))}, \beta=\text{prev(#STEMP)}, \gamma=\text{#STEMP} \\
\texttt{rrrealadd}(\alpha, \beta, \gamma, \delta) = \\
\quad \text{if } \gamma \text{ eq +} \\
\quad \text{then} \\
\qquad 2 \\
\qquad \text{s1 k } \alpha, e, \beta, e, \gamma, e, \delta \\
\quad \text{else} \\
\qquad 2 \\
\qquad \text{s1 k } \alpha, e, \beta, e, \gamma, e, \delta \\
\quad @ \alpha=\text{prev(prev(#STEMP))}, \beta=\text{prev(#STEMP)}, \\
\quad \gamma=\text{addition operator}, \delta=\text{#STEMP} \\
\texttt{rrrecord}(\alpha) = \\
\quad \text{rexplit}(\alpha); \\
\quad \text{radj}(\alpha) \\
\quad @ \alpha=\text{#STEMP} \\
\texttt{rrrentvar}(\alpha, \beta) = \\
\quad \text{rsemi}(\alpha, \beta); \\
\quad \text{adj}(\beta, \beta, \$1 \text{ of } k \beta) \\
\quad @ \alpha=\text{identifier}, \beta=\text{#STEMP} \\
\texttt{rrfind}(\alpha, \beta) = \\
\quad \text{link}(\beta, \$ \text{ of } \text{STORE}); \\
\quad \text{while } \alpha \text{ ineq n } k \beta \text{ do} \\
\quad \text{backward}(\beta) \\
\quad @ \alpha=\text{identifier}, \beta=\text{#STEMP}
173 \texttt{rrselector}(\alpha, \beta) = \\
\quad \text{loopfind}(\alpha, \beta); \\
\quad -1 \\
\quad \text{if} \ e \ k \ \beta \ \text{eq} \ \alpha \\
\quad \text{then} \\
\quad \quad \text{rlfixedsec}(\beta); \\
\quad \quad -1 \\
\quad \quad \text{while} \ n \ k \ \beta \ \text{ineq} \ #T \ \text{do} \\
\quad \quad \text{rforward}(\beta) \\
\quad \text{else} \\
\quad \quad \text{augl}(\text{STORE, #EXTRA}); \\
\quad \quad \text{caseid}(\alpha, \beta); \\
\quad \quad \text{if} \ e \ k \ \beta \ \text{eq} \ \alpha \\
\quad \quad \text{then} \\
\quad \quad \quad \text{rlfixedsec}(\beta); \\
\quad \quad \quad -1 \\
\quad \quad \quad \text{while} \ n \ k \ \beta \ \text{ineq} \ #T \ \text{do} \\
\quad \quad \quad \text{rforward}(\beta) \\
\quad \quad \text{else} \\
\quad \quad \quad \text{loopvariants}(\alpha, \beta); \\
\quad \quad \quad \text{rrvariants}(\alpha, \beta) \\
\quad \quad \text{dell}(\text{STORE, #EXTRA}) \\
\quad \text{end;} \\
\\text{a} \ \alpha = \text{<identifier>}, \ \beta = \#\text{TEMP} \ \text{a} \\

174 \texttt{rrvariants}(\alpha, \beta) = \\
\quad -1 \\
\quad \text{while} \ e \ k \ \beta \ \text{ineq} \ #V \ \text{do} \\
\quad \quad \text{rbackwar}(\beta); \\
\quad \quad \text{rforward}(\beta); \\
\quad \quad \text{if} \ e \ k (\#\text{EXTRA} \ \text{in} \ \text{STORE}) \ \& \ \text{L} \ \beta \\
\quad \quad \text{then} \\
\quad \quad \quad \text{rrselector}(\alpha, \beta) \\
\quad \text{else} \ \text{halt} \\
\\text{a} \ \alpha = \text{<identifier>}, \ \beta = \#\text{TEMP} \ \text{a}
175 rscalarorsubrangerel(\(\alpha, \beta, Y, \delta\)) =
    if \(Y = \text{eq}\) then
        scalarorsubrangevalueequal(\(\alpha, \beta, \delta\))
    else
        if \(Y = \text{eq}\) then
            scalarorsubrangevalueinequal(\(\alpha, \beta, \delta\))
        else
            if \(Y = \text{eq}\) then
                scalarorsubrangevalueless(\(\alpha, \beta, \delta\))
            else
                if \(Y = \text{eq}\) then
                    scalarorsubrangevaluelorgreater(\(\alpha, \beta, \delta\))
                else
                    if \(Y = \text{eq}\) then
                        scalarorsubrangevaluelessorequal(\(\alpha, \beta, \delta\))
                    else
                        scalarorsubrangevaluelorgreaterorequal(\(\alpha, \beta, \delta\))

\(\alpha = \text{prev(prev(#TEMP)))}, \beta = \text{prev(#TEMP)}\),
\(Y = \langle\text{relational operator}\rangle\), \(\delta = \text{#TEMP \&}\)

177 rsemi(\(\alpha, \beta\)) =
    rrfind(\(\alpha, \beta\))
    -1
    link(k \(\alpha \cap \text{ATTRIB}, \beta\))

\(\alpha = \langle\text{identifier}\rangle\), \(\beta = \text{#TEMP \&}\)

178 rset(\(\alpha, \beta, Y, \delta\)) =
    -1
    if \(k = \text{eq \#ANYSET of ATTRIB}\) then
        -1
        link(k \(\beta \cup \alpha\))
    else
        -1
        link(k \(\beta \cap \alpha\))
        if \(Y = \text{eq +}\) then
            si k \(\alpha, e k \beta \cup e k \delta\)
        else
            si k \(\alpha, e k \beta \cap e k \delta\)

\(\alpha = \text{prev(prev(#TEMP)))}, \beta = \text{prev(#TEMP)}\),
\(Y = \langle\text{additional operator}\rangle\), \(\delta = \text{#TEMP \&}\)
179 \( \text{rsetrel}(\alpha, \beta, \gamma, \delta) = \)

\[
\text{if } \gamma \text{ eq } = \\
\text{then } \\
\text{if}(k \ \beta \text{ subset } k \ \delta) \ \Delta (k \ \delta \text{ subset } k \ \beta) \\
\text{then} \\
\text{si } k \ \alpha, \text{true} \\
\text{else} \\
\text{si } k \ \alpha, \text{false} \\
\text{else} \\
\text{if } \gamma \text{ eq } () \\
\text{then } \\
\text{if}(k \ \beta \text{ subset } k \ \delta) \ \lor (k \ \delta \text{ subset } k \ \beta) \\
\text{then} \\
\text{si } k \ \alpha, \text{true} \\
\text{else} \\
\text{si } k \ \alpha, \text{false} \\
\text{else} \\
\text{if } \gamma \text{ eq } <= \\
\text{then } \\
\text{if } k \ \beta \text{ subset } k \ \delta \\
\text{then} \\
\text{si } k \ \alpha, \text{true} \\
\text{else} \\
\text{si } k \ \alpha, \text{false} \\
\text{else} \\
\text{if } \gamma \text{ eq } >= \\
\text{then } \\
\text{if } k \ \delta \text{ subset } k \ \beta \\
\text{then} \\
\text{si } k \ \alpha, \text{true} \\
\text{else} \\
\text{si } k \ \alpha, \text{false} \\
\text{else} \\
\text{if } e \ k \ \beta \ \in \ k \ \delta \\
\text{then} \\
\text{si } k \ \alpha, \text{true} \\
\text{else} \\
\text{si } k \ \alpha, \text{false} \\
\text{a } \alpha=\text{prev(prev(#STEMP)), } \beta=\text{prev(#STEMP),} \\
\text{Y= (relational operator), } \delta=#\text{STEMP} \ \& \ \\
\text{a } \alpha=\text{prev(#STEMP), } \beta=+,-, \gamma=#\text{STEMP} \ \&
\[ \text{rstrgrel}(\alpha, \beta, \gamma, \delta) = \]
\[ \begin{cases} 
  \text{if } e \ k \ \beta \text{eq } e \ k \ \gamma \\
    \text{then} \\
    \text{if } e \ k (\text{WIDTH} \ \text{STORE}) \ \text{ineq } 1 \\
    \text{then} \\
    s1 \ k (\text{WIDTH} \ \text{STORE}), e \ \text{at} (e \ k (\text{WIDTH} \ \text{STORE})) \\
    \text{prd} \\
    \text{forwards}(\beta, \delta) \\
    \text{rstrgrel}(\alpha, \beta, \gamma, \delta) \\
  \text{else} \\
  \text{rsalarorsubrangerel}(\alpha, \beta, \gamma, \delta) \\
  \text{else} \\
  \text{rsalarorsubrangerel}(\alpha, \beta, \gamma, \delta) \\
\end{cases} \]
\[ \alpha = \text{prev(prev(#STEMP))}, \beta = \text{prev(#STEMP)}, \gamma = (\text{relational operator}), \delta = \text{STEMP} \ \& \]

\[ \text{rterm}(\alpha, \beta, \gamma, \delta) = \]
\[ \text{rexplicits}(\alpha, \beta) \]
\[ \text{if } \gamma \ \text{eq and} \\
\text{then} \\
\text{rand}(\alpha, \beta, \delta) \\
\text{else} \\
\text{if } \gamma \ \text{eq mod} \\
\text{then} \\
\text{rmod}(\alpha, \beta, \delta) \\
\text{else} \\
\text{if } \gamma \ \text{eq div} \\
\text{then} \\
\text{rintdiv}(\alpha, \beta, \delta) \\
\text{else} \\
\text{if } \gamma \ \text{eq /} \\
\text{then} \\
\text{rrdiv}(\alpha, \beta, \delta) \\
\text{else} \\
\text{if } (e \ k \ \beta \ \text{IN}) \ \gamma \ (e \ k \ \beta \ \text{R}) \\
\text{then} \\
\text{rtimes}(\alpha, \beta, \delta) \\
\text{else} \\
\text{rintersect}(\alpha, \beta, \delta) \\
\]
\[ \alpha = \text{prev(prev(#STEMP))}, \beta = \text{prev(#STEMP)}, \gamma = (\text{multiplying operator}), \delta = \text{STEMP} \ \& \]
rtimes(\alpha, \beta, \gamma) =
\begin{align*}
\text{if } & (e \land \beta \in \text{IN}) \land (e \land \gamma \in \text{IN}) \\
& \text{then} \\
& \text{link(integer lm ATTRIB,} \alpha); \\
& \text{if } k \land \alpha, e \not\in (e \land \beta, e \land \gamma) \\
& \text{else} \\
& \text{link(real lm ATTRIB,} \alpha); \\
& \text{rinttor}(\beta); \\
& \text{rinttor}(\gamma); \\
& \text{if } k \land \alpha, e \not\in (e \land \beta, e \land \gamma) \\
\end{align*}
\alpha = \text{prev(prev(#STEMP))),} \beta = \text{prev(#STEMP)}, \gamma = \text{#STEMP e}
\]

scalarorsubrangetypeidentical(\alpha, \beta) =
\begin{align*}
\text{if } 1 \land \alpha, \text{ineq } 1 \land \beta \\
& \text{then halt} \\
\end{align*}
\alpha = \text{prev(#ATEMP),} \beta = \text{#ATEMP e}

scalarorsubrangevalueequal(\alpha, \beta, \gamma) =
\begin{align*}
\text{if } e \land \beta, e \not\in (e \land \gamma) \\
& \text{then } k \land \alpha, \text{true} \\
& \text{else} \\
& \text{else } k \land \alpha, \text{false} \\
\end{align*}
\alpha = \text{prev(prev(#STEMP))),} \beta = \text{prev(#STEMP)}, \gamma = \text{#STEMP e}

scalarorsubrangevaluegreater(\alpha, \beta, \gamma) =
\begin{align*}
\text{if } e \land \beta, e \in \text{copyr(DEF of k,} \beta, e \land \gamma) \\
& \text{then } k \land \alpha, \text{true} \\
& \text{else} \\
& \text{else } k \land \alpha, \text{false} \\
\end{align*}
\alpha = \text{prev(prev(#STEMP))),} \beta = \text{prev(#STEMP)}, \gamma = \text{#STEMP e}

scalarorsubrangevaluegreaterprequal(\alpha, \beta, \gamma) =
\begin{align*}
\text{if } e \land \beta, e \in \text{copyr(DEF of k,} \beta, \text{prev((e \land \gamma) of DEF of k,} \beta)) \\
& \text{then } k \land \alpha, \text{true} \\
& \text{else} \\
& \text{else } k \land \alpha, \text{false} \\
\end{align*}
\alpha = \text{prev(prev(#STEMP))),} \beta = \text{prev(#STEMP)}, \gamma = \text{#STEMP e}
188 \textit{scalarorsubrangevalueinequal}(\alpha, \beta; \gamma) = \\
\begin{align*}
\text{if } e_k \beta \text{ ineq } e_k \gamma & \\
\text{then } & \begin{cases} 
\alpha & \text{true} \\
\gamma & \text{false} 
\end{cases} \\
\text{else} & \begin{cases} 
\alpha & \text{false} \\
\gamma & \text{true} 
\end{cases}
\end{align*}
\begin{align*}
\alpha = \text{prev}(\text{prev}(\text{#TEMP})), \beta = \text{prev}(\text{#TEMP}), \gamma = \text{#TEMP} & \\
\text{if } e_k \beta \text{ ineq } e_k \gamma & \\
\text{then } & \begin{cases} 
\alpha & \text{true} \\
\gamma & \text{false} 
\end{cases} \\
\text{else} & \begin{cases} 
\alpha & \text{false} \\
\gamma & \text{true} 
\end{cases}
\end{align*}

189 \textit{scalarorsubrangevalueless}(\alpha, \beta; \gamma) = -1 \\
\begin{align*}
\text{if } e_k \beta & \text{ copyl(DEF of } k \beta, e_k \gamma) \\
\text{then } & \begin{cases} 
\alpha & \text{true} \\
\gamma & \text{false} 
\end{cases} \\
\text{else } & \begin{cases} 
\alpha & \text{false} \\
\gamma & \text{true} 
\end{cases}
\end{align*}
\begin{align*}
\alpha = \text{prev}(\text{prev}(\text{#TEMP})), \beta = \text{prev}(\text{#TEMP}), \gamma = \text{#TEMP} & \\
\text{if } e_k \beta & \text{ copyl(DEF of } k \beta, \text{ prev}(e_k \gamma) \text{ of DEF of } k \beta) \\
\text{then } & \begin{cases} 
\alpha & \text{true} \\
\gamma & \text{false} 
\end{cases} \\
\text{else } & \begin{cases} 
\alpha & \text{false} \\
\gamma & \text{true} 
\end{cases}
\end{align*}
\begin{align*}
\text{while } k \alpha \text{ ineq } k \beta & \\
\text{do } & \begin{cases} 
\text{link}(k \beta, k \alpha); \\
\text{forward}(\alpha); \\
\text{augr}(k \alpha, e_k, (+, e_k, \text{#FIXWIDTH of STORE})); \\
\text{sep} \\
\text{morewidth}(\text{#FIXWIDTH of STORE}, \gamma); \\
\text{forward}(\alpha); \\
\text{morewidth}(\text{#FIXWIDTH of STORE}, \gamma)
\end{cases}
\end{align*}
\begin{align*}
\alpha = \text{prev}(\text{prev}(\text{#ATEMP})), \beta = \text{prev}(\text{#ATEMP}), \gamma = \text{#ATEMP} & \\
\text{if } \#\text{ANYSET of ATTRIB } \not\in \{ k \alpha, k \beta \} & \\
\text{then } & \begin{cases} 
\text{adjsin}(\alpha, \beta); \\
\text{explicit}(\alpha, \beta); \\
\text{scalarorsubrangegetypeidentical}(\alpha, \beta)
\end{cases}
\end{align*}
\begin{align*}
\alpha = \text{prev}(\text{#ATEMP}), \beta = \text{#ATEMP} & \\
\text{if } \#\text{ANYSET of ATTRIB } \not\in \{ k \alpha, k \beta \} & \\
\text{then } & \begin{cases} 
\text{adjsin}(\alpha, \beta); \\
\text{explicit}(\alpha, \beta); \\
\text{scalarorsubrangegetypeidentical}(\alpha, \beta)
\end{cases}
\end{align*}
193 setrange(α, β) =
    \(1\)
    if ordinal(α DEF of k α) \in \(-1\)
    copyUINT, ordinal(β DEF of k α)
    then
        \(1\)
        si k α, {} 
    else
        \(1\)
        si k α, {copy(DEF of k α, prev(α), next(β))}
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
    \(1\)
197 spactual =
  if #LATER \not{\in} 1 k #STEMP
  then
    if 1 k #STEMP eq (x)
    then
      explicit(prev(#ATEMP));
      -1
    if e(k k prev(#ATEMP) \cap TYPE) \in \{scalar, subrange\}
    then
      -1
    link(#ATEMP, k (x of k #STEMP) \cap ATTRIB);
    explicit(#ATEMP);
    if k #ATEMP ineq real lm ATTRIB
    then
      typecheck(prev(#ATEMP), #ATEMP)
    else halt
    else halt
  else halt

198 standcall(\alpha, \beta) =
  rexplicit(\beta);
  if k \alpha \in \{read lm STORE, readln lm STORE\}
  then
    exread(\beta)
  else
    if k \alpha \in \{write lm STORE, writeln lm STORE\}
    then
      exwrite(\beta)
    else
      if k \alpha eq new lm STORE
      then
        exnew(\beta)
      else
        if k \alpha eq dispose lm STORE
        then
          edxdispose(\beta)
        else
          if k \alpha eq succ lm STORE
          then
            exsucc(\alpha, \beta)
          else
            if k \alpha eq pred lm STORE
            then
              expred(\alpha, \beta)
            else
              if k \alpha eq ord lm STORE
              then
                exord(\alpha, \beta)
              else
                exchr(\alpha, \beta)
            \end{cases}
  \end{cases}
staticlink=
    insertr(STORE,#STEMP,#STEMP);
    link(#STEMP,k prev(#STEMP));
    while e k #STEMP ineq $ do
      forward(#STEMP);
      link($ of STORE,k #STEMP);
      delr(STORE,#STEMP)

strgiden($,b) =
  if e($1 of $1 of k $) eq 1
    then
      if e($1 of $1 of k b) eq 1
        then
          if e($1 of $1 of k b) eq e($1 of $1 of k b)
            then
              adjsin($,b);
              forwards($,b);
              at(#ATEMP);
              link(#ATEMP,k prev($));
              at(#ATEMP);
              link(#ATEMP,k prev(prev($))); explicit($prev(#ATEMP)#ATEMP);
              strgindex(prev(#ATEMP),#ATEMP);
              trimr(ATTRIB,prev(#ATEMP));
              forwards($,b);
              explicit($,b);
            end
            if $ k $ TYPE $ "{scalar, subrange}"
              then
                -1
                if $ k $ TYPE $ "{scalar, subrange}"
                  then
                    adjsin($,b);
                    if $ k $ CHAR
                      then
                        if $ k $ CHAR
                          then
                            if $ k $ ineq $ k
                              then halt
                            else halt
                          else halt
                        else halt
                      else halt
                    else halt
                  else halt
                else halt
              else halt
            else halt
          else halt
        end
      else halt
    end
  else halt

@ $ = prev(#ATEMP); $ = #ATEMP @
201 \text{string} \text{index}(\alpha, \beta) = -1$

\text{if } e(k, k \in \alpha \cap \text{TYPE}) \text{ eg subrange then}
\text{adj \text{sin}(\alpha, \beta) ;}
\text{if } e(\text{1 } \text{of } k \in \alpha \text{ eg } +1 \\
\text{then }
\text{if } e(\text{1 } \text{of } k \in \beta \text{ eg } +1 \\
\text{then }
\text{if } l \in \alpha \text{ subset IN then }
\text{if } l \in \beta \text{ subset IN then }
\text{if } l \in \alpha \text{ ineq } l \in \beta \\
\text{then } \text{halt else } \text{halt else } \text{halt else } \text{halt else } \text{halt else } \text{halt}
\text{else } \text{halt}

@ \alpha = \text{prev(ATEMP)} ; \beta = \text{ATEMP} @

202 \text{string} \text{type}(\alpha) =
\text{if } e(\text{1 } \text{of } k \in \alpha \text{ eg } 1 \\
\text{then }
\text{adj \text{sin}(\alpha) ;
forward(\alpha) ;
\text{at(ATEMP) ;
link(#ATEMP, k \text{ prev(ATEMP)}) ;
explicit(#ATEMP) ;
in \text{dextypeofstrg(#ATEMP) ;
deir(ATTRIB, #ATEMP) ;
forward(\alpha) ;
explicit(\alpha) ;
\text{-1 \\
if } e(k, k \in \alpha \cap \text{TYPE}) \in \{\text{scalar, subrange} \\
\text{then }
\text{adj \text{sin}(\alpha) ;
\text{if } l \in \alpha \text{ subset CHAR then } \text{halt else } \text{halt else } \text{halt}
\text{else } \text{halt}
\text{else } \text{halt}

@ \alpha = \text{ATEMP} @

203 \text{subhead} =
\text{headcompile;}
\text{headrun}
204  subodd=
    log(\phi);
    at(\phi);
    \text{link}(\phi \text{ of } \text{ATTRIB}, \phi \text{ of } \text{STORE});
    \text{link}(\phi \text{ of } \text{STORE}, \phi \text{ 1m } \text{STORE});
    \text{function};
    \text{if}\ e^2 \not\equiv (e \text{k (x of } \text{STORE}), +2) \text{ eq } +2 \mod
    \text{then} \ s1 \text{k #OUTPUT FROM odd, true}
    \text{else} \ s1 \text{k #OUTPUT FROM odd, false};
    \text{funcval}(\text{odd});
    \text{trimr}(\text{ATTRIB, } \phi)
    \text{trimr}(\text{STORE, } \phi)

2048 \text{timesnum}(\alpha, \beta, \gamma) =
    \text{-1}
    \text{if}\ e(k \ k \ \beta \ \text{TYPE}) \ \in \ \{\text{scalar, subrange}\}
    \text{then}
    \text{-1}
    \text{if}\ e(k \ k \ \gamma \ \text{TYPE}) \ \in \ \{\text{scalar, subrange}\}
    \text{then}
    \text{adjisin}(\beta, \gamma);
    \text{if}(1 \ k \ \beta \ \text{subset IN}) \land (1 \ k \ \gamma \ \text{subset IN})
    \text{then}
    \text{link}(\alpha, \text{integer 1m } \text{ATTRIB})
    \text{else}
    \text{if}((1 \ k \ \beta \ \text{subset IN}) \land (1 \ k \ \gamma \ \text{eq R}) \lor
    ((1 \ k \ \beta \ \text{eq R}) \land (1 \ k \ \gamma \ \text{subset IN}) \lor
    ((1 \ k \ \beta \ \text{eq R}) \land (1 \ k \ \gamma \ \text{eq R}))
    \text{then}
    \text{link}(\alpha, \text{real 1m } \text{ATTRIB})
    \text{else} \text{halt}
    \text{else} \text{halt}
    \text{else} \text{halt}

\text{D } \alpha = \text{prev(prev(#ATEMP))), } \beta = \text{prev(#ATEMP)}, \gamma = \text{#ATEMP } \&
205 tottype(α) =
-1
    if e(k k α TYPE) ∉ {scalar, subrange}
    then
        (at(#ATEMP);
        link(#ATEMP, $1 of k prev(#ATEMP));
        trans(augl(ATTRIB, #IDTEMP));
        at(#ATEMP);
        link(#ATEMP, $1 of k prev(#ATEMP));
    )
    while k #ATEMP ineq Ø do
        (if n k #ATEMP eq #T
            then
                trans(joins(#IDTEMP $m ATTRIB, #ATEMP));
                typetranslate(#ATEMP)
            else
                -1
                if k #ATEMP ineq #T
                    then
                        trans(link(#IDTEMP $m ATTRIB, k #ATEMP));
                        while next(n k #ATEMP) ineq #T do
                            (forward(#ATEMP);
                            trans(forward(#ATEMP))
                        )
                    )
                delr(ATTRIB, #ATEMP)
                trans(del(ATTRIB, #IDTEMP))
            )
       )
α = #ATEMP α
206 transfer(α, β) =

-1

link(k α ∩ TYPE, k β);

if e(k α ∩ TYPE) e g type identifier
then

2

link(k β k α ∩ ATTRIB)
else

-1

if e(k α ∩ TYPE) ∈ {scalar, subrange}
then

augl(ATTRIB, #IDTEMP);
at(#ATEMP);
link(#ATEMP, $1 of k prev(prev(#ATEMP)));
at(#ATEMP);
link(#ATEMP, $1 of k prev(prev(#ATEMP)));
cattrans(#IDTEMP lm ATTRIB, prev(#ATEMP), #ATEMP);
trimr(ATTRIB, prev(#ATEMP));
dell(ATTRIB, #IDTEMP)

else

if #ATEMP = prev(#ATEMP) ∧ #ATEMP = #ATEMP ∧

207 translate types =

at(#ATEMP);

link(#ATEMP, $ of ATTRIB);
forward(#ATEMP);
while k #ATEMP ineg #ATEMP do

trans(at(α k #ATEMP));

forward(#ATEMP)
);

trans(at(#ATEMP);

link(#ATEMP, $ of ATTRIB);
forward(#ATEMP)
);

adjl(#ATEMP, k #ATEMP, next( $ of #ATEMP));
while k #ATEMP ineg ø do

(typetranslate(#ATEMP);
forward(#ATEMP);
trans(forward(#ATEMP))
);

dell(ATTRIB, #ATEMP);
trans(dell(ATTRIB, #ATEMP))
208 twochar(α, β) =
    if(α ∈ CHAR ∧ β ∈ CHAR)
    then
        s1 $k$ #STEMP, M is ();
        augr($k$ #STEMP, $\{α, β\}$)
        else halt;
    at($T$ is (DIMEN is (1),
        $T$ is (DEF is (+1,+2), WIDTH is (1)),
        $T$,
        WIDTH is ()
    )
); link(array of TYPE, $T$ of ATTRIB);
link($T$ of $T$ of ATTRIB, char lm ATTRIB);
link(type identifier of TYPE, $T$ of $T$ of ATTRIB);
link(subrange of TYPE, $T$ of $T$ of ATTRIB);
link($T$ of ATTRIB, $k$ #STEMP)
end

209 typecheck(α, β) =
    /explicit(α, β);
    if $k$ α eq real lm ATTRIB
    then
        -1
        if $e(k$ $k$ β ∩ TYPE) ∈ {scalar, subrange}
        then
            if (DEF of $k$ $k$ β subset IN) ∧ (DEF of $k$ $k$ β ineq R)
            then halt
        else
            if $k$ $k$ β eq #ANYSET of ATTRIB
            then
                -1
                (if $e(k$ $k$ α ∩ TYPE) ineq set
                then halt
            )
        else
            if $k$ $k$ β eq #ANYPOINTER of ATTRIB
            then
                -1
                (if $e(k$ $k$ α ∩ TYPE) ineq pointer
                then halt
            )
        else
            identical(α, β)
    end
end
end

210 \text{typeidtrans}(\alpha) = \\
\text{if } \text{ordinal}(k \alpha \cap \text{ATTRIB}) \text{ then} \\
\text{copyr UINT, ordinal($\bar{x}$ of ATTRIB)} \\
\text{else} \\
\text{trans(link($k \#\text{ATEMP}$, ($e(k \alpha \cap \text{ATTRIB})$ of ATTRIB)))} \\
\text{loc(\#STEMP);} \\
\text{2 backidtp($e(k \alpha \cap \text{ATTRIB})$);} \\
\text{link($k \text{prev}(\#\text{ATEMP}), k \#\text{ATEMP}$);} \\
\text{delt(ATTRIB, \#ATEMP);} \\
\text{delt(\text{STORE, \#STEMP})} \\
\text{\alpha = \#\text{ATEMP} \&} \\

211 \text{typesets}(\alpha, \beta, \gamma) = \\
\text{if } k \gamma \text{ eq \#ANYSET of ATTRIB} \text{ then} \\
\text{link(\alpha, k \gamma);} \\
\text{else} \\
\text{if } k \gamma \text{ eq \#ANYSET of ATTRIB} \text{ then} \\
\text{link(\alpha, k \beta);} \\
\text{else} \\
\text{at(\#ATEMP);} \\
\text{link(\#ATEMP, k prev(\beta))}; \\
\text{at(\#ATEMP);} \\
\text{link(\#ATEMP, k prev(prev(\gamma))}); \\
\text{setiden(prev(\#ATEMP), \#ATEMP);} \\
\text{trimr(ATTRIB, \text{prev}(\#ATEMP));} \\
\text{link(\alpha, k \beta)} \\
\text{\alpha = prev(prev(\#ATEMP)), \beta = prev(\#ATEMP), \gamma = \#ATEMP \&} \\

212 \text{typesunique}(\alpha, \beta) = \\
\text{at(\#ATEMP);} \\
\text{link(\#ATEMP, k prev(\alpha))}; \\
\text{at(\#ATEMP);} \\
\text{link(\#ATEMP, k prev(prev(\alpha))}); \\
\text{identical(prev(\#ATEMP), \#ATEMP);} \\
\text{trimr(ATTRIB, prev(\#ATEMP))} \\
\text{\alpha = prev(\#ATEMP), \beta = \#ATEMP \&}
213 \text{typetranslate}(\alpha) = \\
\quad \text{trans}(\text{link}(k \alpha \cap \text{TYPE}, k \#\text{ATEMP})); \\
\quad \begin{cases} \text{if } \varepsilon(k \alpha \cap \text{TYPE}) \equiv \text{eq type identifier} \\
\quad \text{typeidtrans}(\alpha) \\
\quad \text{else} \\
\quad \text{tohtype}(\alpha) \\
\quad \alpha = \#\text{ATEMP} \end{cases}

214 \text{valstandfunc} = \\
\quad \text{if } k \text{prev}(\#\text{STEMP}) \in \{\text{succ lm STORE, pred lm STORE}\} \\
\quad \text{then} \\
\quad \quad \text{explicit}(\#\text{ATEMP}); \\
\quad \quad \begin{cases} \text{if } \varepsilon(k \#\text{ATEMP} \cup \text{TYPE}) \in \{\text{scalar, subrange}\} \\
\quad \text{then} \\
\quad \quad \text{if } k \#\text{ATEMP} \text{ ineq real lm ATTRIB} \\
\quad \quad \text{then} \\
\quad \quad \quad \text{link}(\text{prev}(\text{prev}(\#\text{ATEMP}), k \#\text{ATEMP}) \\
\quad \quad \text{else} \quad \text{halt} \\
\quad \quad \text{else} \\
\quad \quad \text{else} \\
\quad \quad \text{if } k \text{prev}(\#\text{STEMP}) \equiv \text{ord lm STORE} \\
\quad \quad \text{then} \\
\quad \quad \quad \begin{cases} \text{if } \varepsilon(k \#\text{ATEMP} \cap \text{TYPE}) \in \{\text{scalar, subrange}\} \\
\quad \quad \text{then} \\
\quad \quad \quad \text{if } k \#\text{ATEMP} \text{ eq real lm ATTRIB} \\
\quad \quad \quad \text{then} \quad \text{halt} \\
\quad \quad \quad \text{else} \quad \text{halt} \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \text{typecheck}(\text{prev}(\#\text{ATEMP}), \#\text{ATEMP})
\end{cases}
\end{cases}
215 \text{valuepar} =
\begin{align*}
\text{if } & k \text{ prev}(\#STEMP) \in \text{STANDF} \cup \text{AB} \\
\text{then} & \text{valstandfunc} \\
\text{else} & \\
\text{if } & k \text{ prev}(\#STEMP) \in \text{STANDP2} \\
\text{then} & \text{writepar} \\
\text{else} & \\
\text{if } & \#\text{LATER} \in k \text{ prev}(\#STEMP) \\
\text{then} & \text{formalsub}(\text{prev}(\#STEMP)) \\
\text{else} & -1 \\
\text{if } & e(k, k \text{ #STEMP } \cap \text{OBJECT}) \text{ eq formal value} \\
\text{then} & \text{typecheck(} \text{prev}(\#ATEMP), \#ATEMP) \\
\text{trans(} \text{assign(} \text{prev}(\#STEMP), \#ATEMP) \text{)} \\
\text{else} & \text{halt}
\end{align*}

216 \text{valvarpar} =
\begin{align*}
\text{if } & k \text{ prev}(\#STEMP) \in \text{STANDP1} \\
\text{then} & \text{varstandproc} \\
\text{else} & -1 \\
\text{if } & e(k, k \text{ #STEMP } \cap \text{OBJECT}) \text{ eq formal variable} \\
\text{then} & \text{identical(} \text{prev}(\#ATEMP), \#ATEMP) \\
\text{trans(} \text{link(} k \text{ prev}(\#STEMP), k \text{ #STEMP) \text{)} \\
\text{else} & \text{valuepar}
\end{align*}

217 \text{varformalvalvar(\alpha) =}
\begin{align*}
\text{-1} & \text{link(} \#ATEMP, k \text{ \alpha of STORE } \cap \text{ATTRIB)}; \\
\text{-1} & \text{trans(} \text{link(} k \text{ \#STEMP } \cap \text{ATTRIB}, \#STEMP); \\
\text{adj1}(\#STEMP, k \text{ \#STEMP, } \#1 \text{ of } k \text{ \#STEMP) \\
\text{\@ \alpha = \{identifier\} \@}
\end{align*}
variants(\(\alpha, \beta\)) =

\[
\text{while } e \ k \ \alpha \ \text{eq} \ #V \ \text{do} \\
\\text{(forwards}(\alpha, \beta); \\
\text{if } l \ k \ \alpha \ \text{eq} \ l \ k \ \beta \\
\text{then} \\
\text{forwards}(\alpha, \beta); \\
\text{field}(\alpha, \beta); \\
\text{forwards}(\alpha, \beta) \\
\text{else} \ \text{halt} \\
) \\
\]
\[\text{\(\alpha = \text{prev}(\#\text{ATEMP})\)}, \text{\(\beta = \#\text{ATEMP}\) \(\alpha\)}\]

varpartiden(\(\alpha, \beta\)) =

\[
\text{forwards}(\alpha, \beta); \\
\text{if } e \ k \ \alpha \ \text{ineq} \ #\text{EVP} \\
\text{then} \\
\text{if } e \ k \ \beta \ \text{ineq} \ #\text{EVP} \\
\text{then} \\
\text{forwards}(\alpha, \beta); \\
\text{contvarpart}(\alpha, \beta) \\
\text{else} \ \text{halt} \\
\text{else} \\
\text{if } e \ k \ \beta \ \text{eq} \ #\text{EVP} \\
\text{then} \\
\text{forwards}(\alpha, \beta) \\
\text{else} \ \text{halt} \\
\]
\[\text{\(\alpha = \text{prev}(\#\text{ATEMP})\)}, \text{\(\beta = \#\text{ATEMP}\) \(\alpha\)}\]

varstandproc =

\[
\text{if } k \ \text{prev}(\#\text{STEMP}) \ \in \{\text{read } \text{lm} \ \text{STORE}, \text{readln } \text{lm} \ \text{STORE}\} \ \text{then} \\
\text{readpar}(\#\text{ATEMP}) \\
\text{else} \\
\text{explicit}(\#\text{ATEMP}); \\
-1 \\
\text{if } e(k \ k \ \#\text{ATEMP} \ \text{\(\&\text{TYPE}\}) \ \text{ineq pointer} \\
\text{then} \ \text{halt} \\
\]
221 \text{vcpf}(\alpha)=
\begin{align*}
&\text{if } \alpha \in n \text{ STORE} \\
&\quad \text{then} \\
&\quad \quad \text{trans}(\text{rref}(\alpha, \#\text{STEMP}))
\end{align*}
-1
\begin{align*}
&\quad \text{if } e(k, \alpha \text{ of STORE } \cap \text{ OBJECT}) \notin \\
&\quad \quad \{\text{variable, formal value, formal variable}\} \\
&\quad \quad \text{then} \\
&\quad \quad \quad \text{varformalvalvar}(\alpha) \\
&\quad \quad \text{else} \\
&\quad \quad \quad -1 \\
&\quad \quad \quad \text{if } e(k, \alpha \text{ of STORE } \cap \text{ OBJECT}) \equiv \text{ constant} \\
&\quad \quad \quad \quad \text{then} \\
&\quad \quad \quad \quad \quad \text{singleconst}(\alpha) \\
&\quad \quad \quad \quad \text{else} \\
&\quad \quad \quad \quad -1 \\
&\quad \quad \quad \quad \quad \text{if } e(k, \alpha \text{ of STORE } \cap \text{ OBJECT}) \notin \\
&\quad \quad \quad \quad \quad \{\text{function, formal function}\} \\
&\quad \quad \quad \quad \quad \text{then} \\
&\quad \quad \quad \quad \quad \quad \text{funcformalfunc}(\alpha) \\
&\quad \quad \quad \quad \quad \text{else} \\
&\quad \quad \quad \quad \quad \quad \text{halt} \\
&\quad \quad \quad \text{else} \\
&\quad \quad \quad \quad \quad \text{halt} \\
&\quad \quad \text{if } \alpha=\langle\text{identifier}\rangle \#\alpha
\end{align*}

222 \text{writeactual}=
\begin{align*}
&\text{if } \#\text{LATER } \notin n \quad l \quad k \quad \#\text{STEMP} \\
&\quad \text{then} \\
&\quad \quad \text{writeproc}
\end{align*}

223 \text{writelnactual}=
\begin{align*}
&\text{if } \#\text{LATER } \notin n \quad l \quad k \quad \#\text{STEMP} \\
&\quad \text{then} \quad \text{writelnproc}
\end{align*}

224 \text{writelnproc}=
\begin{align*}
&\text{if } l \quad k \quad \#\text{STEMP } \text{ineq} (\cdot) \\
&\quad \text{then} \\
&\quad \quad \text{writeparameters}
\end{align*}
225 writepar= explicit(#TEMP);
    -1
    if e(k k #TEMP ∩ TYPE) ∈ {scalar, subrange}
    then
        (if (1 DEF of k #TEMP subset CHAR) ∧
            (1 DEF of k #TEMP subset IN) ∧
            (1 DEF of k #TEMP ineq BOOL) ∧
            then halt)
    else
        -1
        if e(k k #TEMP ∩ TYPE) eq scalar
        then
            (if k #TEMP ineq real lm ATTRIB
                then halt)
        else
            -1
            if e(k k #TEMP ∩ TYPE) eq array
            then
                stringtype(#TEMP)
            else halt
    226 writeparameters=
        loc(#STEM)
        link(#STEM,$1 of k prev(#STEM))
        while k #STEM ineq Ø do
            (at(#STEM)
            -1
            link(#STEM,k k #STEM ∩ ATTRIB)
            writepar;
            delr(ATTRIB,#STEM);
            forward(#STEM)
            )
            delr(STORE,#STEM)
    227 writeproc=if 1 k #STEM eq ()
        then halt
        else
            writeparameters