THE DESIGN AND SEMANTIC ANALYSIS OF A PROGRAMMING LANGUAGE AND ITS COMPILER

P. HENDERSON

Ph.D. Thesis

September, 1970.

UNIVERSITY OF NEWCASTLE UPON TYNE

ACKNOWLEDGEMENTS

The author would like to thank Mr. M.J. Elphick for his supervision during the period of this research, in particular for the time he devoted to reading this thesis and for the helpful comments he made.

Thanks are also due to Miss H. Bell who typed the thesis and to my wife who did many odd jobs concerned with its production.

The research was supported initially by the Science Research Council and subsequently by the University of Newcastle upon Tyne.

ABSTRACT

A programming language ALEPH and its compiler have been designed under the constraint of being able to derive a functional description of the semantics of the language from the implementation. The way in which this approach developed is described, particularly the reason why it is considered as a design tool rather than a method of proving the equivalence of implementation and description. As a design tool, the approach proves to be very critical of irrational language and implementation features.

The language is described informally and then an implementation for a hypothetical computing device is given, by means of a simple translator. The translator associates with each source phrase, a sequence of commands in the machine language of the hypothetical computing device. A symbolic model of this device is described and the semantics of each command in the machine language is defined as a mapping of the internal states of this model. By compounding the effect of sequences of such mappings, the effect of a translated source phrase can be obtained. The recursive nature of the syntax of ALEPH necessitates an inductive approach to this analysis, where it is proved that a phrase has a certain property whenever its component phrases have. This property guarantees the integrity of the states of the model.

The derived functional description is used to answer some simple questions about ALEPH, to demonstrate its usefulness. The main problems and limitations of the approach and its possible extension are considered briefly. Appendices provide documentation of the author's experimental implementation.

CONTENTS

CHAPTER 1 — 1

introduction, outline of dissertation topic.

CHAPTER 2 — 16

ALEPH language, sample ALEPH programs, implementation of ALEPH for hypothetical computer.

CHAPTER 3 — 64

symbolic model of implementation, analysis of model, derivation of semantic functions for ALEPH.

CHAPTER 4 — 108

discussion and application of derived semantic functions, transfer of control in ALEPH.

CHAPTER 5 — 132

comments and conclusions.

APPENDIX 1 — 146

glossary of technical terms used to define ALEPH.

APPENDIX 2 — 159

the author's experimental implementation of ALEPH.

APPENDIX 3 177

sample ALEPH programs annotated and run under the author's implementation.

APPENDIX 4 204

references.

CHAPTER 1

1.0 The need for precise specification of high level programming languages stems from the exact interpretatation afforded to them by their implementations. An implementation itself can be considered as such a specification but it is unlikely to be in a form suitable for a language user to determine language features. An informal natural language specification on the other hand, does not have the necessary precision. A formal specification, which is more descriptive than an implementation, is required as a standard against which implementations and informal specifications can be checked.

For a language like ALGOL 60 however, providing such a specification is a major **problem**. Many of the features provided in ALGOL 60, although simple in themselves, may combine in the most complex fashion. It is this potential complexity which makes formal specification difficult. We must recognise however, that the constraints under which ALGOL 60 was designed, which include consideration for the programmer, for the field of application and for machine design, have been well balanced. ALGOL 60 is generally accepted as being a step in the right direction in programming language design.

The problem of providing a formal specification of a programming language like ALGOL 60 could be simplified if some of its features were rationalised or restricted in use. It was with this consideration in mind that the research reported in this dissertation was begun. How the research developed and the results it produced are described in this Chapter. The problem of providing an ad hoc formal description, even of a programming language over whose design one has control, turns out to be a difficult exercise. The possibility of deriving a descriptive formal specification from an implementation is the alternative investigated and this turns out to have the desirable bonus of establishing the "correctness" of the implementation.

In the following Sections, three alternative methods of programming language specification are exemplified. These are respectively an informal specification, an implementation and a formal specification. The possibility of establishing the consistency of three such specifications is discussed. This enables us to introduce some concepts and terminology. The way in which a programming language was defined and repeatedly redefined to simplify its description, its implementation and the equivalence of these two, is described. Support for this approach is however left until Chapter 5 where its effect can be more readily appreciated. Finally, this Chapter concludes with a description of the way these results are presented in the remainder of the dissertation.

1.1 Let us consider then a hypothetical programming language, similar to ALGOL in that it allows for the recursive nesting of statements and expressions. In particular, we shall examine an <u>iterating statement</u> which has the form

<u>while</u> B <u>do</u> S

where B is a Boolean expression and S is a statement. The effect of the iterating statement is to repeat statement S, zero or more times while B is true. More precisely, B is evaluated and if it is true then S is executed and the whole execution of while B do S is repeated. When B first evaluates to false, the execution is complete.

Thus we have defined the rule whereby an iterating statement may be written, if one knows how to write a Boolean expression and some alternative forms of statement. This is (a generative form of) the syntax of the iterating statement. Supplementary rules informally ascribe a meaning to the iterating statement once constructed, assuming again that we have access to rules defining how Boolean expressions and alternative forms of statements are to be assigned a meaning. These rules specify the semantics of the iterating statement in terms of the activities "evaluation" and "execution". A complete specification of the semantics of the language is needed to define these terms precisely. The recursive nature of the syntax specification necessarily means that this definition is also recursive. That is to say the "execution" of the iterating statement is defined in terms of "execution" of the general class of statements of which the iterating statement is a member.

In general then, the syntax identifies the structure of a program and the semantics determines its meaning. Certain rules are obviously syntactic or semantic in nature but this is not always the case. Syntactic rules may be qualified by semantic requirements or alternatively semantic rules may be specified by the syntactic constraints. For example, the way in which "type" is specified in ALGOL 60 is both by syntactic and semantic rules. We shall not be concerned with formal syntactic specification in this dissertation, nor shall we attempt to establish standards for semantic specification and hence this difficulty will not occur.

1.2 We turn our attention now to a simple scheme for language translation and interpretation and exemplify this scheme using the iterating statement of the previous Section. The iterating statement is part of the _source language_ which is mapped by the _translator_ to an equivalent program in some _object language_. The object language is structurally simpler than the source language in the sense that it is directly interpretable by some simple computing device. That is to say, the object language is an ordered sequence of _commands_ similar to a current day machine language. Such a language we shall refer to as _procedural_.

In order to specify the translator, we require an analytic form of the syntax. We shall use an abstract analytic syntax after McCarthy (1962a, 1964) which, in the case of statement $S \equiv \underline{\text{while}}\ B_1\ \underline{\text{do}}\ S_1$, defines three functions

$$\begin{aligned} \text{iswhile}\ (S) &\equiv \underline{\text{true}} \\ \text{condition}\ (S) &= B_1 \\ \text{statement}\ (S) &= S_1 \end{aligned}$$

The predicate "iswhile" is applicable to any statement S and is <u>true</u> only if S is an iterating statement. The selector functions "condition" and "statement" are applicable only to an iterating statement S and select the Boolean expression and qualified statement respectively.

Using this form of the analytic syntax, we may specify a function "translate (S,i)" which maps the source statement S to a command sequence in the object language. We shall use + as a concatenation operator between strings and "label (i)" as a function whose value is the string "Li" for some integer i. We have

```
translate (S,i) = ....
.... if iswhile (S) then
     begin  generate (label (i)  +  ":");
            translatexp (condition (S));
            generate ("if value ≡ true then begin");
            translate (statement (S), i + 1);
            generate ("goto  +  label (i)  + "end;")
     end else ....
```

The function "generate (t)" presents the string "t" to the object language processor and the function "translatexp (B)" translates expressions. Thus informally we see that the statement S ≡ <u>while</u> B_1 <u>do</u> S_1 translates into μS, where

```
    μS ≡ Li: μB₁
             if value ≡ true then
             begin μS₁; goto Li end;
```

The translation μB_1 of expression B_1 leaves the value of the expression in the variable "value". We see therefore that the effect of S, described earlier in Section 1.1, is obtained by this translation. Indeed, the translation can be considered as a procedural description of the semantics of the iterating statement.

1.3 Using the abstract analytic syntax of the previous Section, we can give a functional description of the semantics of the iterating statement. The technique is based on the work of McCarthy (1962a, 1964) and Landin (1963a). We define a recursive function "effect (S,ξ)" whose value is the <u>environment</u> that results when the statement X is executed in the environment ξ.
The environment is basically a collection of information concerning the accumulated state of the computation. The sort of information derivable from the environment is the current values of all declared variables. We use a complementary primitive semantic function "value (B,ξ)" which returns the value of the expression B in the environment ξ.

```
effect (S,ξ) = ....
.... if iswhile (S) then while (condition (S), statement (S), ξ)
         where while (B,S,ξ) =
                  if value (B,ξ) = true then while (B,S, effect (S,ξ)) else ξ
     else ....
```

Here we see that if "effect (S,ξ)" identifies the type of the statement as an iterating statement, the the effect is defined by the recursively specified function "while". This is the method described by McCarthy. Landin (1964, 1965b) is concerned with the production of an equivalent function rather than the exhibition of an interpreting function. Suppose that the value of "neffect (S)" is a function, applicable to an environment, which defines the semantics of S by updating the environment. If "nvalue (B)" yields a function which when applied to the environment ξ yields the value of B with respect to ξ, we may have

```
neffect (S) =
.... if iswhile (S) then while (nvalue(condition(S)),neffect(statement(S)))
       where while (B',S')(ξ) =
              if B'(ξ) then while (B',S')(S'(ξ)) else ξ
     else ....
```

Here we see that the value produced by "neffect(S)" is a function, tailored from the components of S, which defines the effect of S as an environment mapping function. The language in which the function "neffect" is defined is called the <u>semantic metalanguage</u> and the language in which the value of "neffect(S)" is defined is called the <u>semantic object language</u>. In Landin's work, the semantic metalanguage is called "applicative expressions" and is essentially the λ-calculus with some predefined primitives. His semantic object language is called "imperative applicative expressions" which includes applicative expressions as a proper part of it. It has imperative features which are evaluated for their "effect" and can be considered as a generalised subset of all Algol-like programming languages.

Strachey (1964) uses a semantic object language which has no imperative features. The semantic object language introduced in Chapter 3 of this dissertation is based directly on the work reported by Strachey, differing slightly in notation but substantially in domain of definition. The whole approach to semantic description used in this dissertation is based on the work reported by McCarthy, Landin and Strachey.

1.4 In the previous Sections, we have identified three different types of language specification. Firstly in Section 1.1, an informal language specification was exemplified which combined an informal generative syntax description with an equally informal semantic description. Section 1.2 specified the language with an (abstract) analytic form of the syntax and a translator function which defined the semantics by mapping the source language to a structurally simpler (procedural) object language. Similarly in Section 1.3, we saw how to associate a semantic describing function with a source language statement, again using an abstract analytic form of the syntax. Choosing to ignore the way in which a source language construct is written, we may leave the method of syntax description unspecified and identify each of the above specification techniques by one of the keywords: informal, procedural, functional. That is to say, we distinguish them by the way in which they describe the semantics.

Let us consider a language for which we have an informal, a procedural and a functional specification. An obvious question to ask is whether the three specifications are consistent. Before we look at this question however, let us consider how these specifications may have come about and who might use them.

An informal language specification is the most familiar: an example is the Algol report (Naur 1963). This is the form a user of the language finds most useful because it is the simplest to understand. It is also the form in which the language features are originally conceived and specified. As such, it acts as a problem specification for the implementer. It suffers from the imprecision of natural language and is commonly subjected to revision when it is discovered that an unfortunate interpretation can be put on it.

A procedural description of the semantics of the language is the second commonest and usually takes the form of a machine language program. Seldom is the procedural description very readable, an important exception however, is the original description of EULER provided by Wirth (1966). A procedural description intended also to be read would be of most use to a potential implementer. Any implementation effectively forms a procedural description which is as precise as the translator specification and object languages. Because of the simpler structure, this is presumably more precise than the informal source language specification.

A functional description of the semantics of a language is seldom provided, but a noteable exception is the work of the PL/1 group reported by Bandat (1968). Such a description is of most use to the language theorist or the language designer. It provides a formal basis for the derivation of language properties and a working language for the formulation of semantics for proposed language features.

Let us consider then the consistency of our three forms of description. Establishing consistency with the informal description is necessarily itself informal and subjective. Inconsistency with this description may however be due to an ambiguity in the informal description allowing for its misinterpretation. Consistency between the procedural and functional descriptions can be more precisely determined due to the formality of the two descriptions. Again, inconsistency arises because of errors in the descriptions. Thus we may expect to have to adjust all three specifications to obtain consistency.

This dissertation presents a language which is the result of an approach to language design based on establishing the above consistencies. The remainder of this Chapter is devoted to a description of this approach, why it was chosen and a preview of the results it produced. We begin with a description of the foundations upon which the language was built and the alternative approaches attempted. In particular, we describe an early experimental approach to language design which was found unsatisfactory mainly because of the imprecisely specified objectives.

1.5 Landin (1965b) has produced a formal specification of the semantics of ALGOL based on a semantic object language which is mainly functional but has certain imperative features. The complexity of this description is the direct result of the combination of features in ALGOL which conspire to complicate either the form of the semantic object description or the primitives of the semantic object language. This can be seen clearly when one studies Landin's model for programming languages, ISWIM (Landin 1966). A major conspiratorial feature is the control transfer (goto) in ALGOL and the freedom with which it may be used. An apparently simple control transfer can require the most complex environment updating process. ISWIM suggests however, that a judicious choice of source language constructs will considerably simplify the functional description without reducing expressive capability.

On the other hand, EULER is described by an implementation (Wirth 1966) and based on earlier ideas expressed by Wijngaarden (1962) and Wirth (1963). This implementation includes a very readable procedural description of the semantics of EULER. Wirth sees this description as a desirable alternative to Landin's functional method. He considers the purpose of a formal language specification as twofold: that it should provide a useable description for a programmer and that it should convince him of the reliability of an implementation to reflect this description. A readable procedural description certainly satisfies both these criteria.

Thus we have two language types, ISWIM and EULER, susceptible respectively to simple functional description and simple procedural description. The language ALEPH described in Chapter 2 is based on ideas taken from both these languages, where the ideas have apparently simplified description. For example, ALEPH has no control transfer because it does not mix well with functions, additionally assignment is performed by an expression rather than a statement to simplify the implementation. This latter device considerably simplifies the source language by removing the distinction between expression and statement.

1.6 Some earlier versions of ALEPH were substantially more sophisticated both in facilities and implementation to the version to be presented here. This was mainly because the earlier motivation for designing ALEPH had been "linguistic" in the sense that experiments were directed more towards the ways in which algorithms could be expressed than towards language description. Such an approach is unsatisfactory because there is no yardstick against which one can measure one's success. It has however had a beneficial affect upon ALEPH, in that the language has been designed to be discussed (on paper at least) and has thus acquired an individual terminology. This allows the informal specification of its syntax to be quite rigorous.
The familiar form of syntax specification, Backus-Naur Form, has not been used in this dissertation because it introduces too much superfluous information for our purposes. Finally with regard to this inception of ALEPH, let me say that its name is derived from an early title for the research which was "<u>a</u>lgorithmic <u>l</u>anguage <u>e</u>xperiments".

1.7 The original design of ALEPH was aided by a syntax directed compiler-compiler designed by the author and described in Appendix 2 where the final implementation of ALEPH is reviewed. The implementation was simple enough for a procedural description of the semantics to be abstracted from it. Next came the problem of providing a functional description of the semantics along the lines of Strachey (1964).

The provision of ad hoc semantic functions is not easy. They last just as long as it takes to find a counter-example, which is not a satisfactory state of affairs since one never considers them to be reliable. It soon became obvious that ALEPH still had some features which were not going to yield easily to functional description.

The problem of an unreliable functional description would be overcome if the semantic functions could be derived from the procedural description. This would also solve the problem of consistency, assuming every step in the derivation could be verified. If the informal interpretation of the functional description was consistent with the original language specification then the three-cornered consistency would have been established. Further we would have established the "correctness" of the implementation in those areas where the semantics are concerned.

It turned out to be necessary to redesign repeatedly both ALEPH and the implementation in order to make a success of this approach. The result consists of a symbolic model of the implementation and an analysis of this model to derive a functional description of the semantics of the source language from the semantics of the object language. The recursive nature of the syntax requires that this derivation takes the form of an inductive proof that the source language phrases each possess a certain property.

However, as we have pointed out above, ALEPH did not easily fit into this scheme and a continuous process of rationalisation of source language concepts and modification of implementation features was necessary to simplify both the derivation and the derived semantic functions. As such then the approach is a design tool in the sense that the source language and implementation are "improved" by this feedback process of proving the consistency of three specifications. It is not easy to give a concrete example of this feedback without discussing the details of the language itself first and so we shall leave this until Chapter 5. Let me say however, that as well as rationalising individual source language features, the feedback affected major design decisions such as the way a variable was to be associated with a value at run time. Unlike standard ALGOL implementations (Dijkstra 1961b, many others), where a variable is accessed by knowing its relative position within a block, it was necessary to implement a much simpler scheme. This scheme, which is simply to associate each name with a fixed location at execution time, is known to be inferior in certain major respects. This is not regarded however as a design fault because the functional description of the semantics predicts the ill effects that this can have in certain cases. We shall review this design decision in Chapter 5.

1.8 In Chapter 2 the informal specification of the final version of ALEPH is given. As we have said, this relies on an informal syntax description which introduces a technical vocabulary which is very important in the ensuing analysis. In this Chapter also an implementation of ALEPH is described by first specifying a hypothetical machine which will support ALEPH at run time and then defining the way in which ALEPH phrases are translated to command sequences for this machine.

In Chapter 3 a symbolic model of this run time machine is described and a functional description of the semantics of the machine language is informally attributed to it. By combining the effects of each of the commands in a sequence, the semantics of each source phrase can be derived from its translation. As we have said above, this requires an inductive proof that the translated phrases possess a certain property essential for the integrity of the interpretation. The derived semantics are acceptable only when this property has been proved. As such this is seen as a "correctness" proof of this small part of the implementation.

Chapter 4 discusses the derived functional description and some results about ALEPH which have been established using it. An example of how control can be exercised in a source language without an explicit control transfer is given. The way in which an explicit control transfer would have affected the language and implementation designs is described in Chapter 5. In this Chapter also, particular effects of the approach are discussed. General effects and the possibility of extending the approach, particularly to incorporate data structures, are also discussed. In retrospect the language ALEPH and the implementation described here are both seen to be unsatisfactory in certain respects. Much of this dissatisfaction however is a direct result of the criticality of this approach to language design.

CHAPTER 2

2.0 In the following informal description of the programming language ALEPH, certain common words will be used in a technical sense. Such words will be underlined on their first occurrence in the text and will subsequently only be used in their technical sense. No confusion should arise with the underlining of words used in the source language to form basic symbols, since in general, the former are nouns and the latter are conjunctions or verbs. A glossary of technical words is given in Appendix 1.

ALEPH will be described here as a programming language for computing with integers. This is however, quite an arbitrary choice of domain and could be replaced throughout this description by any other, provided only that there is a non-empty, distinguishable subset of the domain, upon membership of which conditional branches may be made. Thus for instance, where this dissertation refers to the set of integers and zero (the distinguished subset), it may well have referred to the set of real numbers and a neighbourhood of zero. In this sense, any discussion of specific operations upon integers is superfluous to the intent of the research. However such discussion lends substance to the results and provides an area of application from which examples may be obtained. In order to maintain this independence of the domain of operation, any member of the domain will be referred to as a value, and those values which are also intended to be distinguished as particular, will be referred to as zero. Note that in ALGOL, the domain of operation is the direct union of integers, reals and booleans and the distinguishable subset has only the single element, false.

2.1 In ALEPH, the user is particularly concerned with writing two types of phrase called *value phrases*. A value phrase may be either a *primary* or an *expression* and is complete, or self contained in the sense that it denotes a value (or the computation of a value) in terms of other values. A value phrase may also have a *side effect* and may well be evaluated for just this reason. The exact definition of a side effect will be given later. Thus, in order to specify the *semantics* of a particular type of value phrase, we must define how its value is computed and determine its side effect, if any. Since the syntactic structure of any particular value phrase is recursively defined in terms of other value phrases, the description of ALEPH semantics is necessarily recursive.

2.2 An expression is constructed from primaries and *basic operators*. The basic operators denote simple operations upon the elements of the domain, simple in the sense that they can have no side effect. Thus the side effect experienced when an expression is evaluated is defined by the order in which the constituent primaries are evaluated.

In the particular domain which is described here, the basic operators which are used are those usually denoted by the following symbols

$\times \quad \div \quad \underline{\text{mod}} \quad + \quad - \quad < \quad \leq \quad = \quad \neq \quad \geq \quad > \quad \underline{\text{and}} \quad \underline{\text{or}} \quad \underline{\text{not}} \quad \dot{-}$

These operators, taken in this order, refer to the infixed integer (fixed point) operators of multiplication, quotient of division, remainder of division, addition, subtraction, six relational operators, conjunction, disjunction, negation and unary minus. All but the last two are binary operators for which we use an infix notation and a priority scheme based on the following partition:

{↑, x, ÷, mod} {+, -} {< ≤ = ≠ ≥ >} {not, and} {or}

For example

prim1 + prim2 x prim3

is an expression when the primJ are primaries. The multiplication takes precedence over the addition. The relational operations yield the value -1 if the relation holds and 0 if not. The boolean operations similarly handle 0 and -1 as false and true respectively.

The metalinguistic device of using prim1, prim2 and prim3 as variables ranging over the class of all primaries, and referring to them collectively as primJ will be used through this chapter. In particular, we shall use

exprJ, nameJ, primJ

for representatives of the classes of expressions, names and primaries respectively. In the next chapter it will be necessary to use an abreviated form of metalinguistic variable. It will not be necessary however, to distinguish the various levels of metalinguistic abstraction in which we shall become involved. Already, for instance, we have primJ which ranges over

{prim1, prim2, prim3, ... }

each of which in turn ranges over the class of primaries.

The side effect experienced when an expression is evaluated is determined by the order of evaluation of the constituent primaries. This order is defined to be left to right as written. Thus in the above example, prim1 is evaluated first, prim2 second and prim3 third and the side effect, if any, is defined by this order of evaluation.

2.3 A particular type of primary directly connected with the domain is a constant. A constant is a denotation of a value from the domain, in the particular case of the domain of integers, a signed number consisting of an optional minus sign (±) followed by a non empty sequence of digits.

2.4 A variable is a primary and is defined to be a correspondence between a name and a value. A name is written as a sequence of letters. The value of a variable is the value corresponding to its name. The side effect of evaluating a primary is defined to be the effect of this evaluation upon these correspondences. In particular, the effect is null if all the existing correspondences remain unchanged and no new correspondences are defined. Thus the side effect of evaluating a constant or a variable is null.

2.5 The assignment primary is used to alter the value corresponding to a name. Thus if name1 and expr1 represent a name and an expression respectively, then

```
        name1 :=  expr1
```

is a primary whose value is the value of expr1 and whose side effect is to alter the value corresponding to name1 to the value of expr1.

As mentioned earlier, ALEPH does not distinguish statements and the assignment primary is the principal example of a construction which usually appears as a statement in a conventional programming language but appears as a primary in ALEPH. It will be seen that this imparts the usual meaning to common constructions using assignment. For example, all the following are valid as assignment primaries:

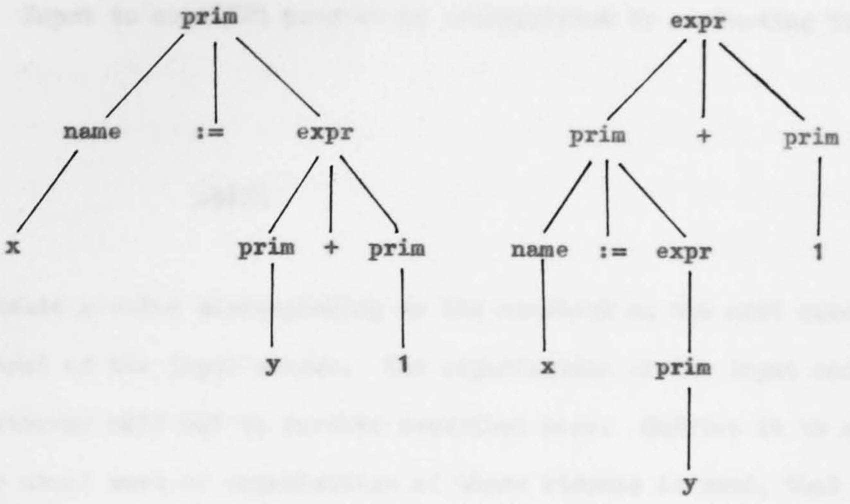
```
x := y + 1
x := y := 0
x := 1 + y := a
```

In the last example, the expression assigned to x is 1 + y := a which consists of the two primaries 1 and y := a. Thus y takes the value of a and x and value a + 1.

There is an ambiguity inherent in the syntax given for the assignment primary, in that since a primary can be a particular example of an expression, it is not possible, without further rules, to define the structure of some phrases. A simple example is the phrase

```
x := y + 1
```

which may be either a primary or an expression, as follows



The ambiguity is resolved by saying that the value assigned is the value of the longest expression to the right of the := sign, consistent with the remaining syntax of the language (e.g. closing parentheses).

2.6 A parenthesised expression is a primary. The value and side effect of this primary are the value and side effect of the enclosed expression. Thus the last example given above can be written

```
x := (1 + y := ■)
```

to distinguish it from

```
(x := 1) + (y := ■)
```

which is an expression whose value is 1 + ■ and whose side effect is to set x to 1 and y to ■.

2.7 Input to an ALEPH program is accomplished by evaluating the primary

input

which yields a value corresponding to the constant on the next record at the head of the input stream. The organisation of the input and output streams will not be further described here. Suffice it to say that the usual sort of organisation of these streams is used, that they are readable and that there is only one stream for input and one for output.

The primary

output prim1

where prim1 is a primary, puts a constant denotation of the value of prim1 into the output stream. Additionally, it may have the side effect of prim1. The value of the primary is the value of prim1. Formatting of the output stream can be accomplished but is not of interest here (see however, the examples in Appendix 3).

2.8 The block primary is used to introduce names and initialise them and such an introduction is called a declaration. The block primary has the form

let name1 = expr1   expr2

where name1 is a name and expr1 and expr2 are expressions. The value of this block primary is the value of expr2, evaluated in the context that name1 initially has the value of expr1. The side effect is the side effect of this same evaluation except that the value originally associated with name1 (in the greater context of the expression of which this primary is a component) is restored upon completion of the evaluation.

Thus for example

<u>let</u> y = u + v

y × y + y

is a block primary whose side effect is null and whose value is the same as (u+v) × (u+v) + u+v. Again the possible ambiguity involved in not having a delimiter for expr1 or expr2 is resolved by demanding that they each extend as far to the right as possible.

We define expr2 to be the <u>scope</u> of this declaration of name1. Any occurrence of name1 in expr2 is referred to as a declared occurrence, or alternatively name1 is said to be <u>declared</u>. Every variable which occurs as a primary in an ALEPH program, and every name on the left of an assignment primary must be declared. If the block primary occurs in the scope of some other declaration of name1, then any occurrence of name1 in expr2 refers to the second (innermost) block primary. However, at the conclusion of evaluating the inner block primary, the value associated with name1 in the outer block primary is restored.

For example

<u>let</u> x = a x + (<u>let</u> x = b x $\times$ x) + x

has the value $2a + b^2$ and null side effect.

The use of name1 in the first expression, expr1, is meaningless except in the case of a self referential occurrence in combination with the definition of a function (Section 2.16). We must however, include expr1 in the defined scope of name1 since any occurrence of name1 in expr1 is to be considered syntactically as a self reference. We shall see that the semantics of the author's implementation do not give a meaningful interpretation except in the case of a function definition.

2.9 In order to group expressions together to compound their side effects, ALEPH provides the <u>compound primary</u>,

<u>begin</u> expr1; expr2; ... ...; exprN <u>end</u>

where the exprJ are expressions. The value and side effect of the compound primary are the same as the value and side effect of evaluating exprN in the context of the side effect produced by evaluating the first N-1 expressions in the order from left to right as written. In effect the values of each exprJ, except exprN, are discarded. As was stated in Section 2.1, these expressions are evaluated simply for their side effects.

2.10 Alternation in the sequence of evaluation is provided for by the <u>conditional primary</u>, which has the form

<u>if</u> expr1 <u>then</u> expr2 <u>else</u> expr3

where the exprJ are expressions. The value and side effect of this primary are the value and side effect of evaluating one of expr2 or expr3 in the context of the side effect of evaluating expr1. The choice between expr2 and expr3 is based on the value of expr1, choosing expr3 if expr1 yields zero and choosing expr2 otherwise. Thus

<u>if</u> $x < 0$ <u>then</u> $x + y$ <u>else</u> $x - y$

has null side effect and corresponds to $x - y$ if $x \geq 0$, and to $x + y$ otherwise. The ambiguity problem introduced by expr3 is solved as before. There is no ambiguity involved with having a conditional primary between <u>then</u> and <u>else</u> since omitting the <u>else</u> is not allowed. In case the conditional primary is evaluated only for its side effect, then both arms must still be included. Any expression with null side effect may be used as a dummy in this instance, but for the sake of efficiency, a constant is suggested, 0 say. For example

<u>if</u> $a > 0$ <u>and</u> $a < 11$ <u>then</u> a:= a + 1 <u>else</u> 0

will increment a only if it originally lay in the range 1-10.

2.11 Repetition in the sequence of evaluation is provided for by the <u>iterating primary</u>, which has the form

<u>while</u> expr1 <u>do</u> expr2

where the exprJ are expressions. This has the effect of evaluating expr1 and expr2 alternately, beginning with expr1. The iteration terminates when expr1 first yields the value zero. The total side effect is then the side effect defined by this order of evaluation, and the value of the primary is the value yielded by expr2 on its last evaluation, or (conventionally) zero if expr2 is never evaluated. Thus

```
while  x > 0  do
begin  q := q + 1;  x := x - y  end
```

increases q by $x \div y$ (if $y > 0$ initially) and reduces x to x <u>mod</u> y. The value of the iterating primary is thus x <u>mod</u> y, or zero if x is initially negative. If y is initially negative, the value and side effect are undefined.

The ambiguity of the syntax given for the iterating primary is resolved as for the other primaries.

2.12 An ALEPH program is a primary in which every occurrence of a name is a declared occurrence.

Before going on to introduce functions and vectors, some simple examples of programs written in this kernel language are given. Results obtained from running these programs in the author's implementation are shown in Appendix 3.

*2.13 Consider how the factorial of a number on the input stream might be obtained. Obviously it is necessary to use the iterating primary to perform the necessary number of multiplications, n say. Thus after

```
while  i ≤ n  do
begin  s : =  s × i;  i := i + 1  end
```

s has as its value n!, if i and s are both initially 1. To form a complete program, it is necessary to declare the variables and initialise them and to output the result, thus

```
let s = 1
begin
        let n = input
        let i = 1
        while i ≤ n do
        begin s := s × i;  i := i + 1 end;

        output s
end
```

This is correct, however a rather more satisfying solution is the following

```
output  let n = input
        let s = 1
        let i = 0
        while (i := i + 1) ≤ n do s := s x i
```

although this is not correct for n = 0.

*2.14 Consider now Euclid's algorithm for the highest common factor of two integers. This may be written as

```
while  x mod y ≠ 0 do
begin a := x;
      x := y;
      y := a mod y
end
```

which reduces y to the highest common factor of x and y (note that if initially x < y, then the effect of the compound primary is to exchange x and y, since x mod y = x in this case). However the compound primary can be compacted to

```
y := x mod x := y
```

according to the definition given for the order of evaluation of primaries in an expression.

Thus

```
(let x = i
 let y = j
 while x mod y ≠ 0  do y := x mod x := y)
```

has as its value the highest common factor of i and j assuming initially that i ≠ j (note that it has the value 0 if i = j). Thus in order to compute Euler's Totient which, for given n, has as its value the number of positive integers less than n which are relatively prime to n, it is sufficient to write

```
output  let n = input
        let i = 0
        let c = 0
        while (i := i + 1) < n  do c := c +
        if (let x = n
            let y = i
            while x mod y ≠ 0  do
            y := x mod x := y) = 1 then 1 else 0
```

In fact the conditional primary can be removed, using the value of the relation (0 or -1) with which to count.

*2.15 Considering how we might form the binomial coefficient

$$^{n}C_{r} = \frac{n!}{r!(n-r)!} = \frac{n(n-1)\ \ldots\ (n-r+1)}{1\ .\ 2\ .\ \ldots\ .\ r}$$

we note that the division could not satisfactorily be performed (using integer arithmetic) until both the numerator and the denominator are complete. For this reason it is necessary to accumulate these values separately, as in

```
let s = 1
let t = 1
begin
        let i = 0
        while i < r do
        begin   s := s × (n - i := i + 1);
                t := t × i
        end;
        s÷t
end
```

This and the previous two examples are further explained in Appendix 3, Examples 1, 2 and 3.

2.16 In order to add functions to the kernel language, we extend the domain to include values known as <u>function references</u>. A function reference may be obtained by evaluating the <u>function definition primary</u> which has the form

<u>lambda</u> name1, name2, ..., nameN. expr0

where the nameJ are names and expr0 is an expression. The result of evaluating this primary is just a value from the domain. This value is of type function reference and although it is indistinguishable from a value of type integer, it is not well defined as an integer, in the sense that the semantics of the language do not interpret this value as a uniquely defined integer.

An example of a function definition primary is

<u>lambda</u> x, y. <u>if</u> x > y <u>then</u> x + y <u>else</u> y - x

which is intended to denote the function of x and y defined by the conditional primary.

In the general form, the list of names between <u>lambda</u> and dot (.), known as the formal parameter list, may be empty (N = 0). Like the block primary, the function definition primary introduces a new level of nomenclature, so that throughout expr0 any occurrence of the nameJ is considered to be a <u>declared</u> occurrence.

2.17 The only valid basic operation which may be performed upon a function reference is known as <u>application</u>. Through application, a correspondence is set up between the nameJ in the formal parameter list and a list of values in an actual parameter list. In order to define application, we must identify a subset of the primaries which we call the aprimaries (<u>aprimary</u> is short for applicable primary). The only primaries so far introduced which are also aprimaries are the parenthesised expression (Section 2.6) and the variable (Section 2.4). If aprim1 is an aprimary and the exprJ are expressions, then

```
aprim1  (expr1, expr2, ...  ..., exprM)
```

is an <u>application primary</u>. The application primary is another example of an aprimary.

If the application primary is such that aprim1 yields the function reference corresponding to the function definition primary of Section 2.16, then the value and side effect of evaluating the application primary is the same as evaluating expr0 in the context of each nameJ having the value exprJ, except that the values originally corresponding to the nameJ remain unchanged. In general, the number of formal parameters might not correspond to the number of actual parameters. In this case, each of the exprJ are evaluated in order, then from this list of values, the correspondences with the nameJ are set up. If there are too many exprJ, the excess values are ignored, if there are too few, then arbitrary (undefined) values are provided.

Consider for example

(<u>lambda</u> x, y. <u>if</u> x > y <u>then</u> x + y <u>else</u> y - x)(p,q)

which is a valid application primary which yields the value p + q or q - p according to whether p > q or not. Equivalently, if F had been assigned the value of the function reference defined by the above function definition primary, we could have written

F(p,q)

instead. This formulation of functions owes a great deal to a similar technique in EULER.

Consider how recursion might be introduced. The function reference assigned to hcf by

```
hcf := lambda x, y. if x mod y = 0 then y else hcf(y, x mod y)
```

will, if applied to two actual parameters, yield their highest common factor, provided only that the variable hcf retains its value until it is used in the function body. Similarly, names can be initialised to values which are function references by means of the block primary.

The method of parameter substitution described here is commonly referred to as "call by value". In particular, any assignment to the formal parameters within function body has a purely local effect. In the case of functions of a single variable, this can be stated more explicitly by an equivalence which will be established in Chapter 4. The following block and application primaries are interchangeable

```
let name1 = expr1  expr2
(lambda name1.  expr2)(expr1)
```

where name1 is a name and the exprJ are expressions, provided only that expr1 does not contain a reference to name1.

The introduction of recursion by functions has ramifications upon the implementation, in particular upon the generality of the block primary. Consider for example the above recursive definition of hcf.

The following alternative

```
hcf := lambda x, y.  let ꞩ = x mod y
                     if ꞩ  = 0 then y else hcf(y,ꞩ)
```

introduces the concept of dynamically nested blocks. In particular, if we consider the function

```
g := lambda x, y.    let ꞩ = x mod y
                     if ꞩ  = 0 then y else g(y,ꞩ) + ꞩ
```

(which has as its result the sum of all the remainders produced when Euclid's algorithm is applied), then it would fail if we used a static allocation of storage for variables. It was necessary to mention this only to motivate the actual implementation of the block primary to be described in Section 2.34.

*2.18   Using functions, we can rewrite the program for Euler's Totient in the following way

```
let hcf = lambda x, y. while x mod y ≠ 0 do y := x mod x := y
let phi = lambda n.  let i = 0
                     let c = 0
                     while (i:=i+1)<n do
                     if hcf(n,i) = 1 then c := c+1 else c
output  phi (input)
```

*2.19 The author's implementation of ALEPH provides none of the standard functions usually available in a high level language. For the next example we require the square root of an integer, which we define to be the integer part of its real square root. We can tackle this in various ways. An initial attempt might be

```
sqrt := lambda x. let i = 0
                  while (i := i + 1) × i ≤ x do i
```

Alternatively, we might hope to speed up the convergence by using an iterative technique such as "bisection" or "Newton Raphson".

However, we shall make do with the above version since we shall only require values of sqrt(x) for x < 100, approximately.

The problem which we wish to solve in this Section is fully defined in Appendix 3 (Example 5). Suffice it to say that, in the following program, the value of circle (n,rsq) is the number of integer lattice points, on an n-dimensional grid, contained within the n-dimensional hypersphere whose radius squared is given by the parameter rsq.

```
let  sqrt = lambda x.  let i = 0
            while i × i < x do i := i + 1
let circle = lambda n, rsq.  if n = 0 then 1 else
            circle (n - 1, rsq) + 2×
            let c = 0
            let i = sqrt (rsq) + 1
            while (i := i - 1) > 0 do
                  c := c + circle (n - 1, rsq - i × i)
output  circle (input, let k = input k×k)
```

2.20 Two alternative forms for the block primary serve to introduce the concept of a <u>vector</u>. Like a function, a vector is considered to be a member of an extended domain, this time extended to include <u>vector references</u>. An n-vector is an ordered set of n + 1 values numbered 0,1,......,n. By means of the block primary

<u>let</u> name1 = <u>row</u> expr1 expr2

where name1 is a name and the exprJ are expressions, storage is allocated for a vector. The value of expr1 (n, say) determines that the vector shall be of length n and the value of the 0th element is initialised to n, the remaining values of the vector elements are undefined. The value assigned to name1 is a vector reference, providing access to the elements of the vector. The storage exists only throughout the evaluation of expr2. An alternative form of the block primary serves to initialise the elements of the n-vector.

<u>let</u> name1 = <u>row</u> expr1 <u>each</u> expr3 expr2

This has the same effect as the block primary described above except that, after expr1 has been evaluated, then expr3 is evaluated. Each of the elements of the n-vector except the 0th is initialised to the value of expr3, the 0th element still being set to the value of expr1. Note that each of the expressions is evaluated once and once only.

2.21 The only meaningful basic operation which may be performed on a vector reference is <u>subscripting</u>. If the aprimary aprim1 yields a reference to an n-vector, then the aprimary

```
        aprim1 @ prim1
```

is meaningful, if and only if, the value of prim1 (i, say) is in the range 0 to n. In this case, the i th value is selected from the n-vector referred to by aprim1. The syntactic ambiguity introduced by prim1 not being delimited is resolved by demanding that prim1 shall extend as far to the right as possible.

2.22 In order to alter the values of an n-vector, an alternative form of the assignment primary is used. Thus

```
        aprim1 @ prim1 := expr1
```

where aprim1 is an aprimary, prim1 a primary and expr1 an expression, and where prim1 and expr1 extend as far to the right as possible, causes the value of expr1 to be assigned to the prim1'th element of the n-vector referred to by aprim1. If prim1 is not in the range 0 to n, then the value and side effect of this primary are undefined.

*2.23 Consider then

```
        if a @ j < a @ i then 0 else a @ (j):= a @ (j) + 1
```

which compares the i th and j th elements of the vector a and increases the j th element by 1 if the j th is less than the i th. On the right of the assignment primary it is perfectly correct to write

```
a @ j + 1
```

since j + 1 is an expression, not a primary. The j + 1st element of a must be denoted by

```
a @ (j + 1)
```

Lehmer's lexicographic method for the mth permutation of the integers 1, .., n requires the factorial expansion of the number m defined by

$$m = a_n (n - 1)! + a_{n-1} (n - 2)! + \dots + a_2 .1!$$

If initially q = n! then the following piece of ALEPH evaluates these

```
let j = n + 1 while (j := j - 1) > 1 do
begin     a @ (j) := m ÷ q;  m:= m mod q;
          q := q ÷ (j - 1)
end
```

which is evaluated for its side effect only. The mth permutation is then generated in the vector a by the following code

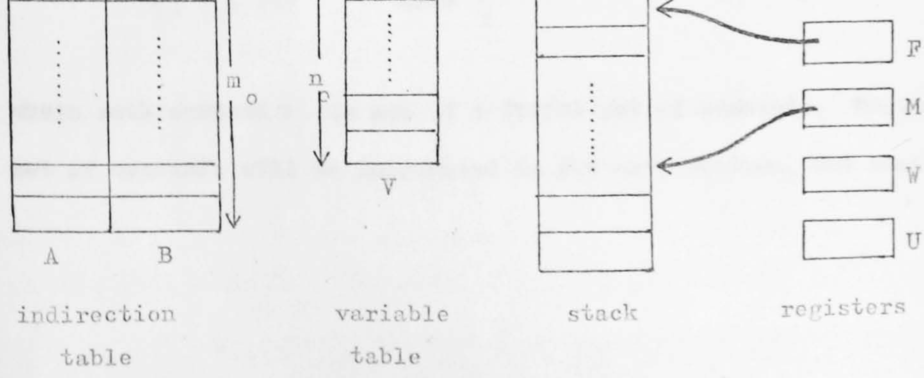
```
a @ 1 := 0;

let i = 1 while (i:= i + 1) ≤ n do
let j = 0 while (j:= j + 1) ≤ i - 1 do
if a @ j = a @ i then 0 else a @(j) := a @ j + 1
```

which is also evaluated for its side effect only. For a complete discussion of this problem and for other examples of the use of vectors, see Appendix 3. Note that it may be preferable to always put the primary after @ in parentheses. This gives a correspondence, [$\leftrightarrow$ @(,]$\leftrightarrow$) between this notation and more usual notations.

An implementation of ALEPH

2.24 The remainder of this chapter is devoted to a description of the author's implementation of ALEPH. Firstly, a particular hypothetical run time machine is described and then, for each value phrase, a unique translation to a macro language, which expands to a program for this machine. The run time machine is abstract, in the sense that no real machine exists which has exactly this specification, however the detail of Appendix 2 may serve to illustrate to anyone familiar with the machine languages of the IBM 360 and the IBM 1130 of the proximity of the run time machine to (a certain mode of operation of) current day machines. It would seem that some more recent central processing units (e.g. PDP 11) are even closer to this hypothetical device than the examples of Appendix 2.

2.25 The particular organisation of the run time machine is illustrated below



The storage area of the run time machine is divided into three sections, referred to as the indirection table, the <u>variable table</u> and the <u>stack</u>. Each element of the storage area is able to represent a value. The indirection table has two fields, called A and B respectively. The length of the two tables is variable but is fixed before the computation begins, in a manner to be described. The elements of the tables and the stack may be considered to be addressed independently by the integers from some arithmetic progressions. These indexes are themselves (internal) values which are indistinguishable from the values manipulated within the computation and can therefore be stored in the elements of storage. There is a set of three special purpose registers and (nominally) two working registers. The register P always points to (holds the index of) the last element put on the stack. Register T contains either the next element to be stacked or the last element removed from the stack, in general. The register M, the <u>marks pointer</u> usually holds the index of an earlier stacked element, and is used in the evaluation of application primaries. Registers W and U are used as working registers in the sense that the number of instructions, during the execution of which they are expected to retain a particular value, is fixed.

A program for the run time machine is a finite sequence of commands

$$c_1, c_2, \ldots \qquad \ldots, c_k$$

where each command c_i is one of a finite set of commands. Most of this set of commands will be introduced in the next Section, but some will be introduced when they are first required. The two fields of the indirection table are set at translate time to contain the indexes of predetermined elements of the command sequence. They remain unaltered throughout the computation. The use of the indirection table is described in Section 2.27.

2.26 In the following description of the types of command available for the run time machine, use will be made of some simple meta-concepts. Namely X and Y will denote registers, while y denotes a constant and x is either a register or a constant.

The command

X = x

resets register X to contain the value denoted by x, which is either a self defined value or the content of another register. As examples of this command we have

T = 127 and F = M.

The command

X = cY

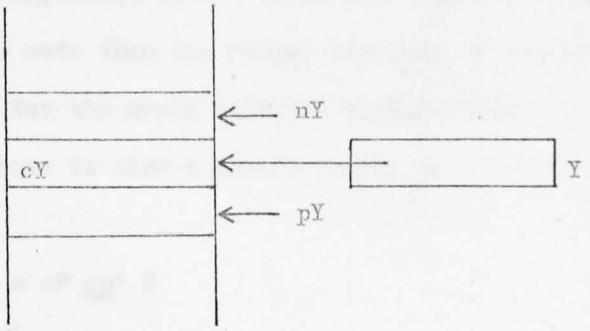
resets register X to contain the contents of the stack element pointed to by register Y. Thus for example, T = cF copies the element on top of the stack into register T. Correspondingly, the command

cX = Y

copies the contents of register Y to the element of the stack pointed to by register X. A stack pointer can be moved by use of one of the commands

X = nY   or   X = pY

where nY refers to the <u>n</u>ext element and pY to the <u>p</u>revious element.



After X = nY the register X points to the next element to that pointed to by Y. For example, F = nF moves the stack top pointer up.

The command X↑Y has the same effect as the sequence (Y = nY, cY = X) and the command X↓Y as the sequence (X = cY, Y = pY). Usually register F is used as the Y field of these commands in which case they can be interpreted respectively as stacking and unstacking register X. The elements of the variable table are accessed by the two commands

$$X = V@x \quad \text{and} \quad V@x = X$$

which respectively load and store the xth item of the table, where x is an integer in the range 1 to n_o. The fields of the indirection table can be used to effect jumps by the two commands

$$\rightarrow B@x \qquad \text{and} \qquad (X=0) \Rightarrow \rightarrow A@x$$

which respectively implement an unconditional transfer of control and a transfer dependant on the content of register X being zero. If a transfer is made then the normal sequence of control is abandoned in preference for the newly selected control index.

There is also a finite number of instructions of the form

T = cF <u>op</u>* T

where <u>op</u>* is the run time machine equivalent of the basic operator <u>op</u> (which in turn stands for any of the basic operators used to construct expressions from primaries). The values denoted by cF and T are combined by <u>op</u>* to form a new value for T.

2.27 In order to simplify the description of the translation from ALEPH to the machine language of the run time machine, we extend the latter by the addition of a macro definition facility. A macro definition may have one of two forms. A simple name for a sequence of commands can be introduced by a definition of the form

$$\text{mname} : (c_1 , c_2 , \ldots, c_k)$$

where the c_i are basic commands, and mname is the name to be given to the sequence. As an example we may have

$$\text{add} : (T = cF + T,\ F = pF)$$

The second form of macro definition has the format

$$\text{mname (i)} : (c_1 , \ldots\ , c_k)$$

where the macro has a name and a parameter. The parameter may be used within the c_j as a constant, and may thus be used to index the tables or as a self defining term. For example we may have

$$\text{load (i)} : (T{\uparrow}F,\ T = V@i)$$

The definition of a program must now be modified to include macro calls as valid replacements for subsequences of commands where the program is understood to be obtained by a simple expansion of the macro calls.

In macros of the second type, any of the basic commands c_j may be labelled

$$A@i : c_j \quad \text{or} \quad B@i : C_j$$

subject to the restriction that when every macro in the program is expanded, there should be exactly one occurrence of each type (A and B) of label for every element of the indirection table. These labellings effectively initialise the indirection table entries. It should be noted at this point that the use of the indirection table effectively makes both translation and assembly single pass (left to right).

In the following Sections, the specific nature of the translations of the value phrases defined in Sections 2.2 to 2.23 will be described.

2.28 An expression is constructed from primaries and basic operators in the form

$$\text{prim1}\ \underline{\text{op}}_1\ \text{prim2}\ \underline{\text{op}}_2\ \dots\ \underline{\text{op}}_{n-1}\ \text{primN}$$

where a priority has been defined for each operator. The priority of these operators can be made explicit by adding to the expression a pair of parentheses for each operator, to enclose the two operands of the operator. The expression then takes on the form

$$(\text{rand1}\ \underline{\text{op}}\ \text{rand2})$$

where randJ (operand) is either one of the original primaries or is also of this form. The translation of

$$(\text{rand1}\ \underline{\text{op}}\ \text{rand2})$$

then has the form

μ rand1, μ rand2, μ <u>op</u>

where μ randJ is the translation of randJ (as a sequence of macros) and μ <u>op</u> is the macro for the basic operator <u>op</u>. Thus for example, if we have

$a + b \times c$

which in parenthesised form is

$(a + (b \times c))$

then this translates to

μ 'a + b x c' = μ 'a', μ 'b', μ 'c', mul, add

where mul $\equiv \mu$ 'x'

add $\equiv \mu$ '+'

The translation of variables has not yet been defined. Finally, the translation of expressions is completed by displaying the macros for the basic operators, these all have the form

μ<u>op</u> : (T = cF <u>op</u>* T, F = pF)

where <u>op</u>* is the run time machine equivalent of <u>op</u>. Inductively, let us presume that the result of evaluating μrand1 is placed in cF and that of evaluating μ rand2 is placed in T. Then μ<u>op</u> leaves the result of μ '(rand1 <u>op</u> rand2)' in T, and moves the pointer down. Intuitively, this will effect the correct evaluation of the whole expression.

So far in this Section, a number of metalinguistic devices have been used without being introduced. Just as the primJ and the randJ are elements of the syntactic metalanguage, being names for sections of ALEPH text, so quoted pieces of text are elements of the same language, being names for the text quoted. The translator function μ which maps sections of ALEPH text to the tanslation (of that text) in the macro language is defined (recursively) by application to elements of the syntax metalanguage. We shall in general, avoid quotes by using basic symbols autonomously. Further, throughout the remainder of this dissertation, square brackets will be used algebraically in whatever language they appear, in that they will only be used to exemplify the syntax. Thus we may write the above result as

μ [rand1 <u>op</u> rand2] = μ rand1, μ rand2, μ<u>op</u>

where the randJ are the operands of the operator <u>op</u> (as defined by the syntax) and where μ<u>op</u> is the macro implementing the operator <u>op</u>.

2.29 If prim1 is a constant, then we define

μ prim1 = num (prim1)

where num(i) : (T↑F, T=i).

The effect of executing the sequence of commands obtained from the translation of a value phrase is always to save the value of T on the stack and to leave the result of the evaluation in T. The translation of an expression as described in the previous Section will be shown in Chapter 3 to also have this property, as will all value phrases.

2.30 In order to describe the translation of a variable, it is necessary to describe the overall translation of a program. In a program there are a certain number of declarations of names either in block primaries or in function definition primaries. If these occurrences are numbered from left to right (as written) with integers from 1 to n_o, then each variable has thus associated with it an index. This index defines the position reserved for it in the variable table. Note that there is no attempt to correlate different declarative occurrences of the same name. Thus we may refer to the index (or index into the variable table) associated with a particular occurrence of a name, either as a primary or in an assignment primary or in the above instances of declarative occurrences.

If prim1 is a variable occurring as a primary and μ name1 is its associated index, then we define

μ prim1 = get (μname1), val

where

get(i) : (T↑F, T=i) $1 \leq i \leq n_o$

val : (T = V⊗T)

Thus we see that get(i) has the property of saving T and replacing T by i. Immediately T is replaced by the content of the variable table associated with the name occurring as a primary. Further, we see that the variable table is used to store the current value associated with the variable.

2.31 Equivalently the assignment primary is translated as follows

μ [name1 := expr1] = get(μ name1), μ expr1, ass

where μ name1 is the index associated with name1 and

ass :(W = cF, V@W = T, F = pF)

The effect is to save the index of the variable on the stack, while expr1 is evaluated. The value of expr1 is left in T while the index is in cF. The macro 'ass' performs the required assignment and removes the index from the stack.

2.32 Trivially we have, for the parenthesised expression primary

μ [(expr1)] = μ expr1

which needs no explanation.

2.33 μ [<u>input</u>] = inp

μ [<u>output</u> prim1] = μ prim1, out

These two definitions effectively avoid the issue of input/output by handing over responsibility for the definition to the macro language.

The macro language in turn defines these in terms of new primitives.

```
inp : (T↑F, T = inp)
out : (out T)
```

where T = <u>inp</u> is a new basic command used to redefine the content of register T according to an input value. The machine operation <u>out</u> T is similarly undefined except to say that the value of T is passed to the output stream.

2.34 The block primary

<u>let</u> name1 = expr1 expr2

is translated into the sequence of macro calls

μ expr1, let (μ name1), μ expr2, tel (μ name1)

where μ name1 is the index associated with name1 by the translator. The macros are defined by

```
let(i) : (W = V@i, V@i = T, T = W)
tel(i) : (W = cF, V@i = W, F = pF)
```

The previous value of the variable is saved in T (and subsequently on the stack) by the first macro, which also initialises the variable to the value in T. This value is of course, the value of expr1. The second expression is evaluated in the light of this new value for the variable and the result of the evaluation is placed in T. Now the previous value of the variable is retrieved from the stack by 'tel' and restored and the stack pointer is moved down.

2.35 The compound primary has a simple translation defined as follows

$$\mu[\underline{\text{begin}}\ e_1;e_2;\ldots\ ;e_n\ \underline{\text{end}}] = \mu e_1,\text{off},\mu e_2,\text{off},\ldots\ \ldots,\text{off},\mu e_n$$

where e_j $(1\le j\le n)$ are expressions and

off : (T↓F)

As each result is obtained for the e_j in T, it is thrown away by 'off', except for the last expression which provides the result of the whole primary.

2.36 A further property of the translator is that it numbers (from left to right, starting at one) each occurrence of a conditional, iterating or function definition primary. This numbering is independent of that defined for names. The number is assigned when the prefix (<u>if</u>, <u>while</u> or <u>lambda</u>) is met in passing from left to right over the source. This number is called the associated index and indexes the corresponding element in the indirection table.

The translation of the conditional primary is defined by

μ[<u>if</u> expr1 <u>then</u> expr2 <u>else</u> expr3] = μ expr1, thn(x), μ expr2, els(x),

μ expr3, ndc(x)

where x is the index associated with this occurrence of the conditional primary. The macros are defined by

```
thn (i) : ((T = 0) ⇒ → A@i, T↓F)
els (i) : (→ B@i, A@i : T↓F)
ndc  ~~mdc~~ (i) : (B@i:)                    1 ≤ i ≤ m_0
```

It is easily seen that the value of expr1 is tested and according to whether it is zero or not, one of the sequences

T↓F, μexpr2

T↓F, μexpr3

is evaluated. In either case, evaluation resumes at B@x, thus ensuring that one and only one of the above sequences is evaluated.

2.37 Evaluation of the iterating primary is very similar.

μ [<u>while</u> expr1 <u>do</u> expr2] = whl(x), μexpr1, dow(x), μexpr2, gow(x)

where x is the associated index and where

```
        whl(i) : (T↑F, T=0, B@i:)
        dow(i) :((T=0) ⇒ → A@i, F=pF, T↓F)
        gow(i) : (→ B@i, A@i: T↓F)
```

First the value of T is saved and T is set to zero, a tentative value for the value of the whole primary. The first expression is evaluated and its value tested. Should this value be zero, then the current value at the top of the stack is accepted as the value of the whole primary. In case the test yields a non-zero value, the previously saved value of the second expression is removed and replaced by a new value, by evaluating the expression again. In this way the loop is performed and the value of the iterating primary is the value obtained when the second expression is last evaluated.

2.38 The function definition primary of Section 2.16 has the form

prim1 ≡ lambda name1, name2, ..., nameN. expr0

where the nameJ are names and expr0 is an expression. This translates as follows

μ prim1 = fun(x), for(μ name1), for(μ name2), ...
 ..., for(μ nameN), chk, μ expr0 , dot, rof(μ nameN), ...
 ..., rof(μ name2), rof(μ name 1), nuf(x)

where x is the index associated with the function definition primary, and μ nameJ the index associated with this declaration of nameJ.

The macros are defined by

```
        fun(i) : (T↑F, T = A@i, → B@i, A@i : F = nnM)
        for(i) : (F = nF, T = cF, W = V@i, V@i = T, cF = W)
        chk : (T↓F)
        dot : (cM = T, T↓F)
        rof(i) : (V@i = T, T↓F)
        muf(i) : (W = cnM, →W, B@i :)
```

Certain new basic commands have been introduced, all of which should be self explanatory. It is pointless to try and explain the operations performed here before the application primary has been translated.

2.39 The application primary of Section 2.17 has the form

```
        prim1 ≡ aprim1 (expr1, expr2, ...  ..., exprM)
```

which is translated into

```
μprim1  =  μaprim1, mrk, μexpr1, μexpr2, ...  ..., μexprM, cll, unm
```

where

```
        mrk : (T↑F, T = M, M = F, F = nF)
        cll : (T↑F, cnM = $, → cM, $ :)
        unm : (F = M, M = T, T↓F)
```

The macro "mrk" (mark) saves T and establishes a three element "mark" consisting of the function reference of the function to be called, which is the value of aprim1 and is held in T, an empty location for the return address and the address of the previous mark. It is intended that the function reference of the called function shall be overwritten by the value of the function. As each expression in the actual parameter list is evaluated, its value is placed on the stack, then "cll" places the value of exprM on the stack, the return address into the second word of the mark and branches to the function.

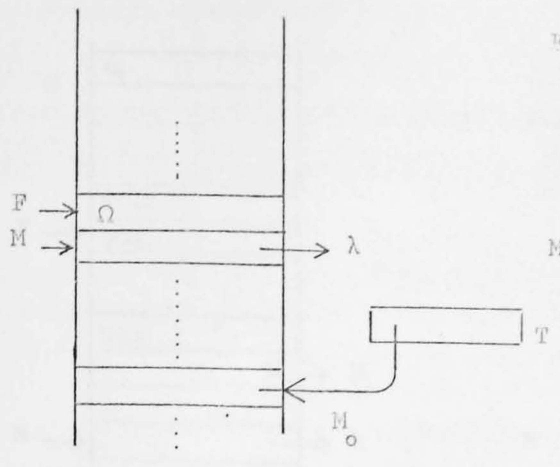
The function is entered at A@i and F is reset to a position just below the list of evaluated parameters. Each occurrence of "for" moves F up the stack exchanging the value of the actual parameter and the previous value associated with the name. The macro "chk" simply resets the contents of T to be significant. The body of the function is evaluated, its result appearing in register T. The macro "dot" copies this result back to the appropriate (i.e. the latest) mark (selected by M) and then each of the "rof" restores the values of the bound variables. Finally, the macro "nuf" causes a branch to the return address, whereupon "unm" (unmark) removes the mark, re-establishing the old mark and setting the value of the evaluated body as the value of the application primary.

It is easily observed that the value of the function definition primary is just the indirection table entry (A field) which (as a value) is referred to as a function reference.

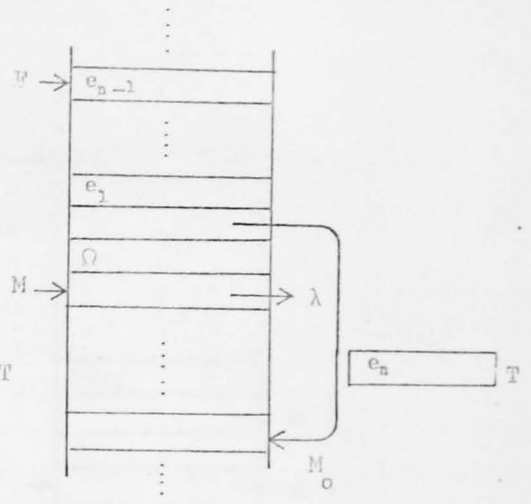
Because of the importance of the way functional application is handled, the description is repeated below pictorially. In these diagrams, the following notation is used:

λ	function reference value of function being called
\$	return address
M_0	previous value of marks pointer
Ω	undefined
e_j	value yielded by j^{th} actual parameter
x_j	index of j^{th} formal parameter in variable table
e_o	value yielded by body of function

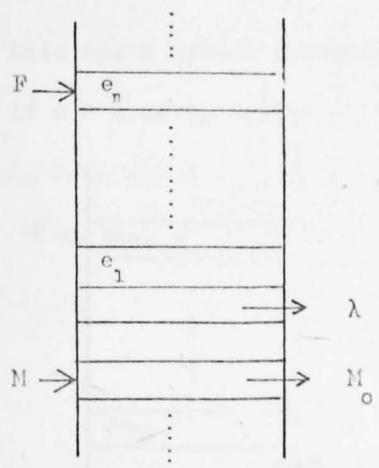
The point reached in the command sequence is shown below each diagram by an arrow indicating just after the last command executed.



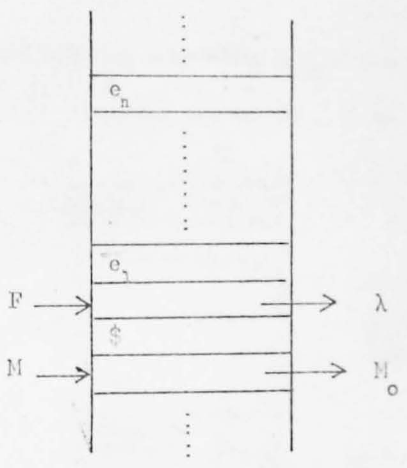
... mrk, ↓expr1,...



... exprN, ↓cll,...

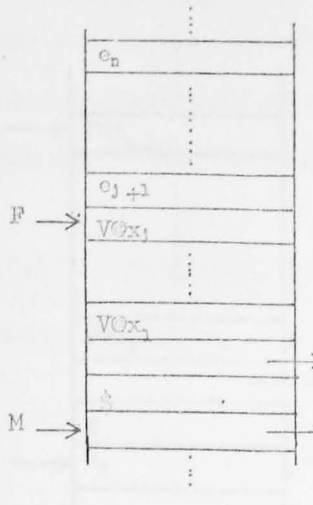


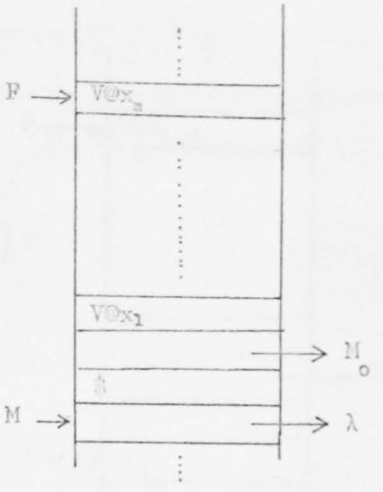
... cll, ↓mrk,...



... fun(i), ↓for(x_1),...

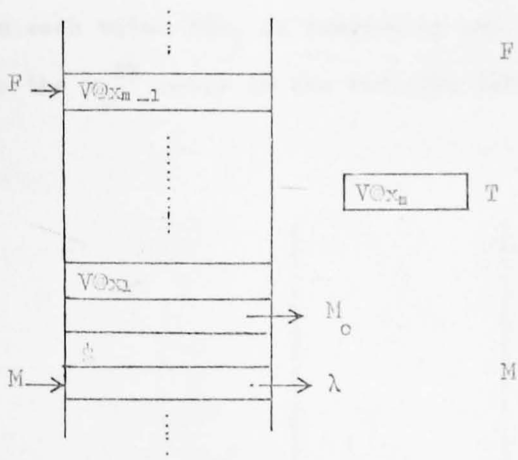
At this stage, neither the value of T nor the value of F is correct in the sense of their specially defined roles. Indeed T is not in use and F is being used to run up the actual parameter list. After executing for (x_j), each of the elements V@x_1, ... , V@x_j has been reset to contain e_1, ...,e_j respectively, while the state of the stack is as follows:

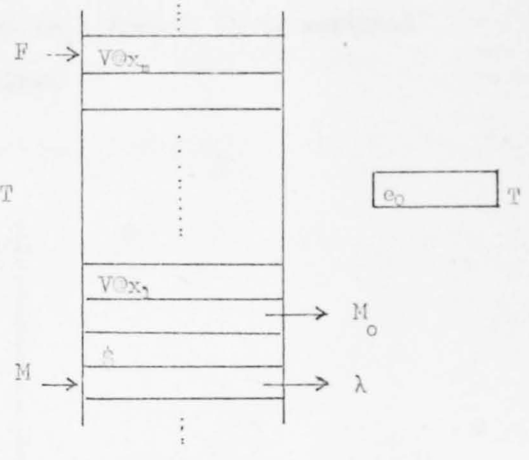




... $for(x_j), {}^{\downarrow}for(x_{j+1}),$... ... $for(x_m), {}^{\downarrow}chk,$...

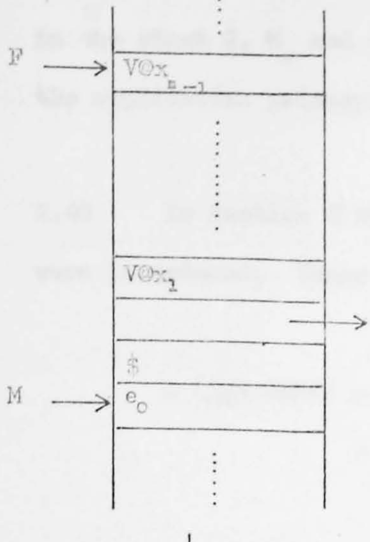
Note how m actual parameters are obtained by ignoring e_{m+1}, .., e_n if $n > m$ or by taking arbitrary values from the stack if $n < m$.



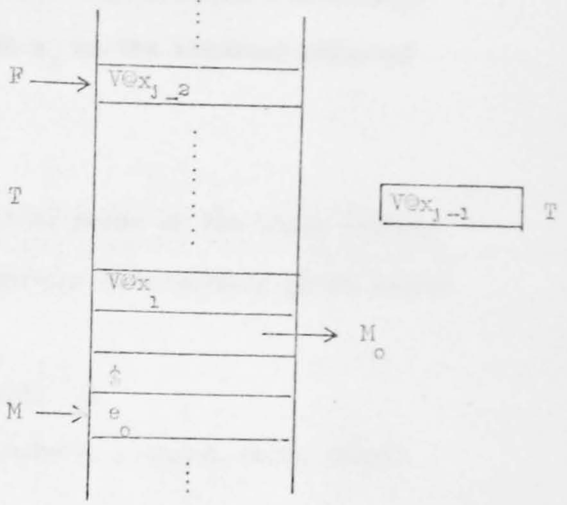


... chk, ${}^{\downarrow}\mu$ expr0,... ... μ expr0, ${}^{\downarrow}$dot,...

The value of the function body, expr0 is left in register T after evaluation. Note that this may have involved arbitrarily many function calls, nesting the marks, but that the integrity of the information stored here will have been retained (if vectors are used incorrectly however, it is possible to interfere with this information since the subscript is not checked for range).

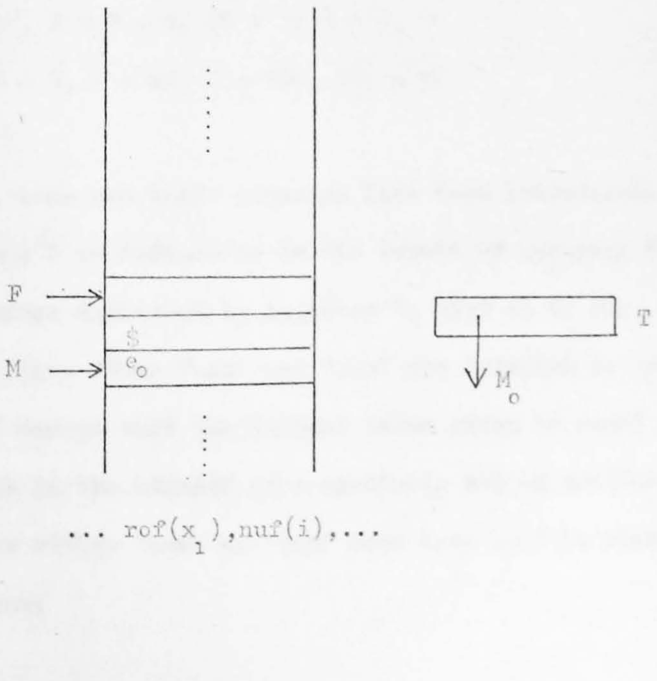


$\ldots$ dot, $^{\downarrow}$rof(x_n), $\ldots$



$\ldots$ rof(x_j), $^{\downarrow}$rof(x_{j-1}), $\ldots$

As each value $V@x_j$ is removed by rof (x_j) from T, it is restored to the x_j th entry in the variable table



$\ldots$ rof(x_1),nuf(i), $\ldots$

The combination of "huf(i)" and "unm" now use the information available in the stack $\$$, M_o and e_o to establish e_o as the returned value of the application primary.

2.40 In Section 2.20, two alternative forms of the block primary were introduced. These have the respective translations given below:

μ [<u>let</u> name1 = <u>row</u> expr1 expr2] =

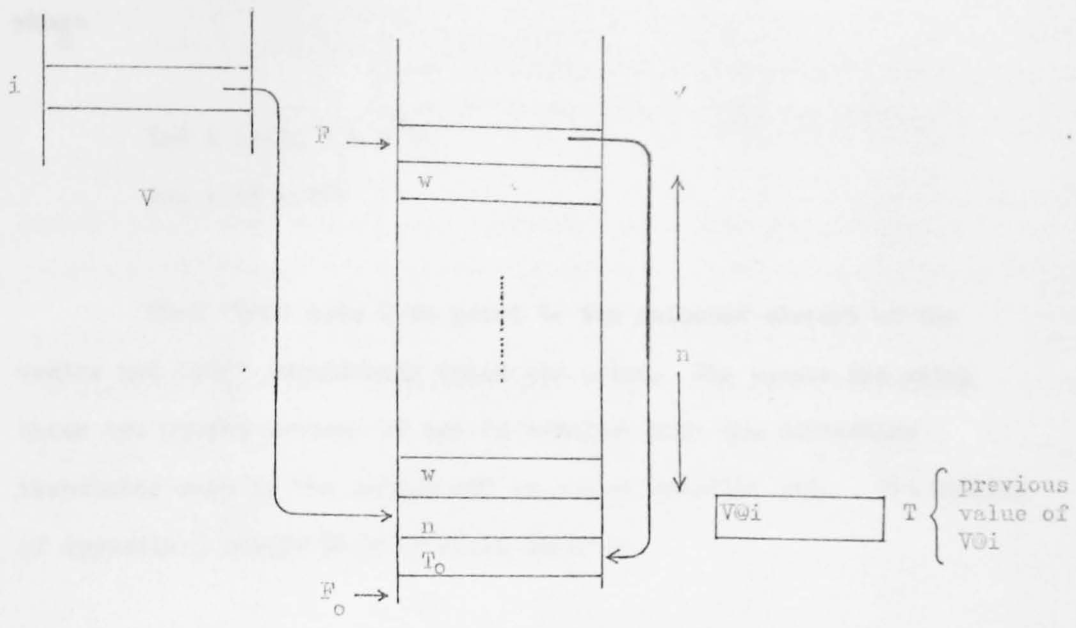
μ expr1, arr(μ name1), μ expr2, wor(μ name1)

μ [<u>let</u> name1 = <u>row</u> expr1 <u>each</u> expr3 expr2] =

μ expr1, μ expr3, row(μ name1), μ expr2, wor(μ name1)

```
arr(i) : (W = F, T↑F, F = n^T F, W↑F, W = nW, T = V@i, V@i = W)
wor(i) : (W↓F, V@i = W, F = cF)
row(i) : (W = F, U = cF, W = pW, U = U + 1,
          $ : T↑F, U = U - 1, (U ≠ 0) ⇒ → $,
              cF = W, W = nW, T = V@i, V@i = W)
```

Again, some new basic commands have been introduced. The command $F = n^T F$ is defined to be the result of applying $F = nF$ the number of times indicated by register T, that is to say, a simple address calculation. Thus "arr" and "wor" are intended to operate like "let" and "tel" except that the initial value given to name1 is a vector reference, which is the address of a specially set up section of stack. The set up after either "arr" or "row" have been used is shown in the following diagram:



T_0 and F_0 are the initial values of T and F, n is the value of expr1 and w the value of expr3 (or undefined in case the vector is not to be initialised). Note how a link is placed on the stack after the vector in order to "down date" correctly. This is in order not to have to rely on the value of V@i which might be changed by assignment.

2.41 Subscripting is implemented by the syntax

aprim1 @ prim1

which translates into

μ [aprim1 @ prim1] = μ aprim1, μ prim1, ind, iva where

ind : (W↓F, T = n^TW)

iva : (T = cF)

Thus "ind" sets T to point to the selected element of the vector and "iva" immediately takes its value. The reason for using these two macros instead of one is involved with the particular translator used by the author and is not of interest here. The details of Appendix 2 should however clear this up.

2.42 Assignment to a vector is accomplished by the syntax (Section 2.22)

aprim1 @ prim1 := expr1

which translates into

μ aprim1, μ prim1, ind, μ expr1, ias

where

ias : (W↓F, cW = T)

The values of aprim1 and prim1 are compounded (by "ind") to form a reference to the element to be altered and "ias" assigns the value of expr1 to this element.

2.43 This concludes the description of ALEPH translation as implemented by the author. Many of the devices used, as mentioned in Chapter 1, are mainly to simplify the proofs of Chapter 3 and should not be criticised until the reader has satisfied himself, after reading Chapter 3, that the method described here is too restricted, in the sense that a more general feature can be established with equivalent simplicity. In Chapter 5, the implementation will be scrutinised when the results of Chapters 3 and 4 are discussed.

CHAPTER 3

3.0 Each value phrase is defined in terms of certain component value phrases. Thus for instance, the value phrases expr1 and expr2 are the component phrases of the following two primaries:

```
while  expr1  do  expr2

let  name1  =  expr1  expr2
```

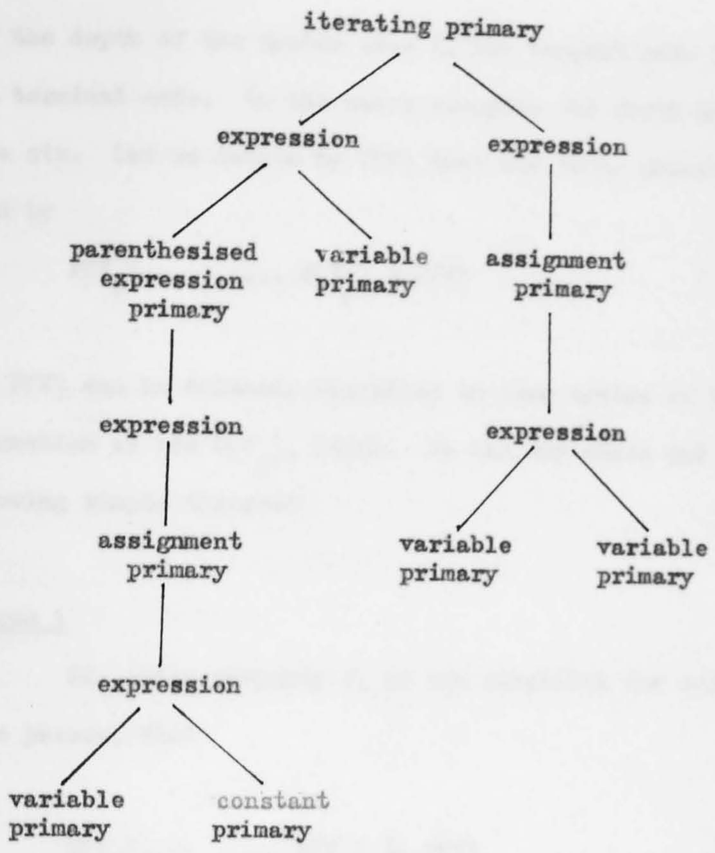
For each value phrase V therefore, we can define a syntax tree as follows:

1. If the value phrase V has no component value phrases, then the root of the syntax tree is labelled with the name of V and the tree has no branches.
2. If the value phrase V has k component value phrases $V_1, \dots \dots, V_k$, then the root of the syntax tree is labelled V and there are k branches from the root, the i^{th} branch being attached to the root of the syntax tree for V_i.

Thus the value phrase

```
while  (i := i + 1) ≤ n  do  s := s + i
```

has the syntax tree:



Note that wherever the syntactic component "name" occurs in the definition of a value phrase, it is ignored for the purposes of constructing the syntax tree.

Given a value phrase V, let us denote by δV, the depth of its syntax tree defined as follows:

1. If V has no component value phrases, then $\delta V = 0$
2. If V is constructed from $V_1, \ldots \ldots, V_k$, then

   $$\delta V = 1 + \max_{1 \leq i \leq k} \delta V_i$$

Thus the depth of the syntax tree is the longest path from the root to a terminal node. In the above example, the depth is easily seen to be six. Let us denote by P[V] that the value phrase V has property P and by

$$P[V_1], \ldots \ \ldots, P[V_k] \vdash P[V]$$

that P[V] can be deduced, according to some system of logic, from the conjunction of the $P[V_j]$, $1 \le j \le k$. We can now state and prove the following simple theorem:

Theorem 1

If, given property P, we can establish for each type of value phrase, that

$$P[V_1], \ldots \ \ldots, P[V_k] \vdash P[V]$$

where V is the value phrase and $V_1, \ldots \ \ldots, V_k$ its components, then we have established

$$\vdash P[V]$$

for any value phrase V. That is to say, V has property P absolutely.

Proof

Proof is by induction on the depth of the syntax tree for V. For consider when V has a syntax tree of depth 0, then from the definition of δV we see that V has no component value phrases. Thus $\vdash P[V]$ is given by the conditions of the theorem.

Our inductive hypothesis then is that we have established $\vdash P[V]$ for all value phrases V for which $\delta V < n$. Consider now the case $\delta V = n$. Let $V_1, \ldots \ldots, V_k$ be the component value phrases of V, $n>0$ implies $k>0$. Further, if $1 \leq i \leq k$ then $\delta V_i < n$ by reductio ad absurdum. For $\delta V_i \geq n$ implies $\delta V \geq n + 1$.

Thus we have:

$$
\begin{array}{ll}
\vdash P[V_1] & \delta V_1 < n \\
\vdash P[V_2] & \delta V_2 < n \\
\vdots & \vdots \\
\vdash P[V_k] & \delta V_k < n
\end{array}
$$

and $P[V_1], \ldots \ldots, P[V_k] \vdash P[V]$ is given

and so we can deduce $\vdash P[V]$. ||

This theorem enables us to break down the proof of a certain property P into a series of lemmas, each of which has the form

"if the value phrases $V_1, \ldots \ldots, V_k$ have the property P then so does the value phrase V which is constructed from them"

In Section 3.3 we shall introduce a property which we wish to establish for the value phrases of Chapter 2 and in the subsequent Sections, we shall prove a series of lemmas of this type.

3.1 If f is a function and x an argument within the domain of f, we denote by fx the result of applying f to its argument. Application associates to the left, thus fxy means [fx]y where fx has as its result a function which is applied to y. In general a sequence of applications is denoted by $f_1 f_2 \ldots f_n$ where each f_i is either a primitive function or a sequence of this type enclosed in brackets. The value of such a combination can be deduced in many ways, but in particular by the following procedure:

1. If n = 1 then the value is that of f_1, either as a primitive evaluation, or by invoking this procedure to remove any substructure.
2. If n > 1 then evaluate f_n as above and then evaluate $f_1, \ldots \ldots, f_{n-1}$ by this procedure. Finally apply the result of $f_1, \ldots \ldots, f_{n-1}$ to f_n.

Thus if the sequence $f_1, \ldots \ldots, f_n$ is defined at all, then it is defined by the above process which in particular demands evaluation of each of the f_i before any of the applications are made.

The function "dot" product f.g, where f and g are functions over appropriate domains is defined by:

$$[f.g]x \equiv f[gx]$$

in the sense that f.g has the value f[gx] at x or is undefined if f[gx] is undefined. The function "dot" product has a lower binding power than application and hence [fx.gy]z is interpreted as [fx][gyz]. Note also that this product is associative and that we shall write [f.g.h]x for f[g[hx]].

By means of the notation of the λ -calculus, we may write the definition of f.g as

$$f.g \equiv \lambda x f[gx]$$

In a λ-expression, the λ is followed immediately by the bound variable and then the body which extends as far to the right as is consistent with the bracketing. The λ-expression, λxA denotes the function, whose value for argument B, denoted by [λxA]B, is obtained by evaluating the expression obtained when B is substituted for all free occurrences of x in A. To remain consistent with the above procedure, we shall assume that B is evaluated before substitution and that any clash of variables (if the result of B contains any λ-expressions with free variables) is catered for by renaming of bound variables.

We denote by $<a_1,\ldots \ldots,a_n>$ the n-tuple consisting of the n elements $a_1,\ldots \ldots,a_n$. We consider an n-tuple as a function from $\{1,2,\ldots \ldots,n\}$ to the set over which the a_i range. Thus if i is an integer between 1 and n, then

$$<a_1,\ldots \ldots,a_n>i = a_i$$

This device enables us to avoid lowering the line for complex subscripts. A useful function defined for an n-tuple is the update function U_n,

$$U_n<a_1,\ldots \ldots,a_n>ib \equiv <a_1,\ldots \ldots,a_{i-1},b,a_{i+1},\ldots \ldots,a_n>$$

which reads "update $<a_1, \ldots \ldots, a_n>$ so that the ith element contains b". It is undefined unless $1 \leq i \leq n$. In general, if no confusion can arise, we shall write U for U_n.

Suppose we have E defined by

$$Eij = \begin{cases} \lambda x \lambda yx & \text{if } i = j \\ \lambda x \lambda yy & \text{if } i \neq j \end{cases}$$

where i and j are integers, then we can supply the following equivalences:

i) $Uabcd \equiv Ebdc[ad]$

ii) $Uab[ab] \equiv a$

iii) $U[Uabc]de \equiv Ebd[Uade][U[Uade]bc]$

That is to say the dth element of Uabc is c if b = d, or the dth element of a otherwise. Secondly, updating a so that the bth element contains ab leaves a unchanged. Finally, the result of updating Uabc so that the dth element contains e is the same as Uade if b = d, otherwise the two updates may be reversed. These equivalences are taken from McCarthy (1962).

Consider a function f, defined by an equation of the form

$$f = \epsilon_f$$

where ϵ_f is a combination of functions which contain an occurrence of f. Then, if such a function f exists, we shall write it as

$$Y\lambda f\epsilon_f$$

Y is called the paradoxical combinator (Curry & Feys 1958) or the "fixed-point" operator (Landin 1963). Effectively it allows us to name the function f independently of the letter used to designate it. As an example, we have the function

$$Y\lambda f\lambda x E\underline{0}x[\lambda x1][Mx.f.N]x$$

which is applicable to an integer. If Nx yields x - 1 and Mxy yields x * y, then this is an expression of the factorial function.

We shall require a characteristic property of Y, which is

$$YF = F[YF]$$

In the above formulation, we have

$$F = \lambda f \varepsilon_f$$

and that YF is the solution of

$$f = \varepsilon_f$$

Thus $\quad YF = \varepsilon_{YF} = F[YF]$

which demonstrates the property of Y expressed above. This property is useful in inductive arguments involving functions of the form YF.

A stack is a particular type of structure which we distinguish with a special notation. Thus we write

$$(s_0, (s_1, (s_2, \ldots \ldots, (s_n, \Lambda)\ldots)))$$

to denote a stack with n + 1 elements, the top element being s_0 and the bottom element being s_n. Λ denotes the empty stack and if s is a stack, then s^+ denotes the top element and s^- the stack that remains after the top element has been removed.

$$(s_0, (s_1, \ldots \ \ldots, (s_n, \Lambda)))^+ = s_0$$
$$(s_0, (s_1, \ldots \ \ldots, (s_n, \Lambda)))^- = (s_1, (s_2, \ldots \ \ldots, (s_n, \Lambda)))$$

The result of putting s_0 on top of the stack s is denoted by (s_0, s) and hence we have

$$s = (s^+, s^-)$$

for any non-empty stack s. It will be necessary later to introduce partitioning of the stack and equivalence of stack elements.

3.2 When we consider the run time machine, there are various elements which can alter as the result of executing a command. Specifically these elements are the registers T, F and M, the variable table and the input and output streams. Although the contents of the working registers W and U may also change, the contents have purely local significance, in the sense that their contents must remain constant only during the execution of a fixed number of commands. This is obvious when the macros of Chapter 2 are considered as partitioning the command sequence. The content of register W is not significant between any two adjacent elements of the partitions. That is to say, within a macro, the first reference to W always assigns a value to it. The current state of all the elements whose contents may change is referred to as the <u>environment</u> ξ. We write

$$\xi = [t,f,m,v,i,o]$$

where

t	is the current value held in register T,
f	is the stack of values identified by register F,
m	is the stack of values identified by register M,
v	is the n-tuple of values held in the variable table,
i	is the stack of values yet to be input,
o	is the stack of values so far output.

The semantics of the machine language can therefore be defined by the effect each command would have upon ξ. Thus, for example

$$[t,f,m,v,i,o] \xrightarrow{T\uparrow F} [t,(t,f),m,v,i,o]$$

$$[t,f,m,v,i,o] \xrightarrow{T\downarrow F} [f^{+},f^{-},m,v,i,o]$$

$$[t,f,m,v,i,o] \xrightarrow{T=V@x} [vx,f,m,v,i,o]$$

$$[t,f,m,v,i,o] \xrightarrow{V@x=T} [t,f,m,Uvxt,i,o]$$

$$[t,f,m,v,i,o] \xrightarrow{cF=T} [t,(t,f^{-}),m,v,i,o]$$

$$[t,f,m,v,i,o] \xrightarrow{F=nF} [t,(\Omega,f),m,v,i,o]$$

$$[t,f,m,v,i,o] \xrightarrow{T=\underline{inp}} [i^{+},f,m,v,i^{-},o]$$

$$[t,f,m,v,i,o] \xrightarrow{\underline{out}T} [t,f,m,v,i^{-},(t,o)]$$

Thus we see that the semantics of the machine language are fairly straightforward and hence a complete list is not given here. A more extensive list is to be found in Appendix 2 where each command is also correlated with its equivalent in the author's IBM 360 implementation of ALEPH. By composing the effect of each command with the result of its predecessor in a sequence, we get a semantic description of the effect of the sequence, thus:

$$\begin{aligned}
[t,f,m,v,i,o] &\xrightarrow{T\uparrow F} [t,(t,f),m,v,i,o] \\
&\xrightarrow{T=x} [x,(t,f,),m,v,i,o] \\
&\xrightarrow{W=cF,\ V@T=W} [x,(t,f,),m,Uvxt,i,o] \\
&\xrightarrow{F=pF} [x,f,m,Uvxt,i,o] \\
&\xrightarrow{T=V@z} [[Uvxt]z,f,m,Uvxt,i,o]
\end{aligned}$$

Where a derivation sequence such as the one above does not refer to individual components of the original environment, as the above does not refer to m,i,and o, then the components will be left out of the sequence. Thus the above sequence is written:

$$[t,f,v] \xrightarrow{T\uparrow F,\ T=x} [x,(t,f),v] \xrightarrow{W=cF,\ V@T=W} [x,(t,f),Uvxt]$$

$$\xrightarrow{F=pF} [x,f,Uvxt] \xrightarrow{T=V@z} [[Uvxt]z,f,Uvxt]$$

The letters used to denote the components of the first element in the sequence determine those components present. Occasionally, when no confusion can arise, we shall not be quite so explicit about which components are present, which should be obvious from the context.

3.3, We noted in Chapter 2 that, when a value phrase is evaluated, the value is left in register T and the previous value of register T has been saved on the stack. Further, this evaluation may have a side effect upon the contents of the variable table and upon the input and output streams. These observations are incorporated in the property we are to establish for all value phrases in ALEPH.

Property A

The value phrase V is said to possess property A, if and only if, the result of evaluating V in the environment $\xi = [t,f,m,v,i,o]$ is the environment $\xi' = [t',(t,f), m,v',i,o']$ where $t' = gv$, $v' = hv$, $o' = kvo$, for some functions g,h,k determined by the phrase V.

Thus we see that, in particular, the original value of T is saved and that of M is restored. The input stream remains unaltered and thus a value phrase which contains an evaluated occurrence of the primary input cannot possess property A. We have noted in Chapter 1 that the results to be obtained here are incomplete in that they do not give a general approach to input and output. We shall discuss later the reasons why a complete coverage has not been given.
It should be obvious how much more complicated property A would become if input and output were to be treated fully. The principal feature of property A is that the value of the phrase V and the side effect upon the variable table can be expressed in terms of the original contents of the variable table. The function g is called the <u>value function</u> associated with V and h the associated <u>side effect function</u>.
We postulate the existence of three functions φ, ψ and η which map a given value phrase V to the respective associated functions, thus

$$g = \varphi V \qquad h = \psi V \qquad k = \eta V$$

Therefore we can write the derivation due to a value phrase possessing property A as

$$[t,f,m,v,i,o] \quad \overset{\mu V}{\rightarrow} \quad [\varphi Vv,(t,f),m,\psi Vv,i,\eta Vvo]$$

The function k has a form such that kvo is the result of pushing some values, dependant only on v, down onto the stack o. We shall not concern ourselves further with the functions k or η, nor distinguish k with a special name. However, the functions φ and ψ play a large role in the following analysis and it is as well to understand them now. Given a value phrase V, then φV and ψV are functions which determine the semantics of V, by application to an n-tuple representing the current values of the variables occurring in V. In particular φVv is a value in the domain, "the value of V" and ψVv is an n-tuple of assignments, "the side-effect of V".

It is our purpose in the remainder of this Chapter to establish the lemmas which, in conjunction with Theorem 1 will establish property A for all value phrases. It should become obvious that property A provides for the harmonious co-operation of the respective translations of the value phrases.

In order to keep this analysis as short as possible, discussion of the functional forms obtained for φ and ψ is left until Chapter 4. This is appropriate also because these forms are not acceptable until property A has been established. Where possible, the discussion of Chapter 4 has been made independant of the derivation performed here. This is possible because there are many ways in which the functional form of the semantics could have been obtained and this Chapter presents only one of them.

3.4 Lemma 1

If e is an expression, constructed from the primaries $p_1, \ldots \ldots, p_k$ in that order and if each p_j, $1 \leq j \leq k$ has property A then so does e.

Proof

In Chapter 2 we showed how e can be written in the form

$$e \equiv r_1 \underline{op}\, r_2$$

if $k \geq 2$, where each of r_1 and r_2 contains at least one of the original primaries. If $k = 1$ then $e \equiv p_1$. Further r_1 and r_2 have the same inductive structure. Thus since $k = 1$ implies e has property A trivially, we take as our inductive hypothesis that r_1 and r_2 both have property A since they have less than k primaries each. Constructing a derivation sequence for

$$\mu[r_1 \underline{op}\, r_2] = \mu r_1, \mu r_2, \mu\underline{op}$$

we have

$$[t,f,v] \xrightarrow{\mu r_1} [\varphi r_1 v, (t,f), \psi r_1 v] \quad \text{since} \quad A[r_1]$$

$$\xrightarrow{\mu r_2} [\varphi r_2[\psi r_1 v], (\varphi r_1 v, (t,f)), \psi r_2[\psi r_1 v]]$$

$$\xrightarrow{T=cF\underline{op}^*T, F=pF} [[\varphi r_1 v] \underline{op} [\varphi r_2[\psi r_1 v]], (t,f), \psi r_2[\psi r_1 v]]$$

and thus e has property A. ||

Corollary 1

Since $\psi[r_1 \underline{op}\, r_2] = \psi r_1 \cdot \psi r_2$ and the function dot product is associative, we have

$$\psi e = \psi p_k \cdot \psi p_{k-1} \cdot \ldots \quad \ldots \cdot \psi p_1$$

which establishes a left to right evaluation rule for the composite side effect of primaries in an expression.

Corollary 2

If $e \equiv p_1 \; \underline{op}_1 \; p_2 \; \underline{op}_2 \; \ldots \; \ldots \underline{op}_{k-1} \; p_k$ and we take the operations into the semantic metalanguage with the same priorities then it follows from the above that

$$\varphi e = \lambda v \, [\alpha_1 v] \; \underline{op}_1 \; [\alpha_2 v] \; \underline{op}_2 \ldots \; \ldots \underline{op}_{k-1} [\alpha_k v]$$

where $\alpha_i = \lambda v \, \varphi p_i \, [\psi p_{i-1} [\ldots \; \ldots [\psi p_1 v] \ldots \; \ldots]]$

$$= \varphi p_i \cdot \psi p_{i-1} \cdot \ldots \cdot \; \ldots \cdot \psi p_1$$

Thus the value of the ith primary is dependant upon the side effect of the first i-1 primaries.

3.5 Lemma 2

The constant primary has property A.

Proof

If $p \equiv c$ is a constant primary then

$$\mu p = \text{num } (c)$$

and we have

$$[t,f] \rightarrow [c,(t,f)]$$

This derivation establishes that p has property A with

$$\varphi p = \lambda v c$$

$$\psi p = \lambda v v$$ ||

3.6 Lemma 3

The variable primary has property A.

Proof

If $p \equiv y$, is a variable primary, let αy be the index of y in the variable table, then

$$\text{get}(x) : (T \uparrow F,\ T = x)$$

$$\text{val} : (T = V @ T)$$

and we have

$$[t,f,v] \xrightarrow{\text{get}(\alpha y)} [\alpha y,\ (t,f),v]$$

$$\xrightarrow{\text{val}} [v[\alpha y],\ (t,f),v]$$

Thus p has property A with

$$\varphi p = \lambda v v[\alpha y]$$

$$\psi p = \lambda v v \qquad ||$$

3.7 Lemma 4

If the expression e has property A then so does the assignment primary $p \equiv y := e$ where y is a declared name.

Proof

Let αy be the index of y in the variable table.

We have

$$\mu[y := e] \;=\; get(\alpha y),\ \mu e,\ ass$$

where

$$ass : (W{\downarrow}F,\ V@W = T)$$

thus

$$[t,f,v] \xrightarrow{get(\alpha y)} [\alpha y,\ (t,f),v]$$

$$\xrightarrow{\mu e} [\varphi ev,\ (\alpha y,\ (t,f)),\ \psi ev] \quad \text{since} \quad A[e]$$

$$\xrightarrow{ass} [\varphi ev,\ (t,f),\ U[\psi ev][\alpha y][\varphi ev]]$$

which establishes that p has property A. We have

$$\varphi[y := e] \;=\; \varphi e$$

$$\psi[y := e] \;=\; \lambda v U[\psi ev][\alpha y][\varphi ev] \qquad ||$$

3.8 Lemma 5

If the expression e has property A then so does the parenthesised expression primary $p \equiv (e)$.

Proof

Trivial, since $\mu p \;=\; \mu e$. ||

3.9 Lemma 6

If the primary p_2 has property A, then so does the output primary $p_1 \equiv$ output p_2.

Proof

$$[t,f,v,o] \xrightarrow{\mu p_2} [\varphi p_2 v,\ (t,f),\ \dagger p_2 v, o] \quad \text{since} \quad A[p_2]$$

$$\xrightarrow{\underline{\text{out}}T} [\varphi p_2 v,\ (t,f),\ \dagger p_2 v,\ (\varphi p_2 v, o)]$$

Hence p has property A and $\varphi p_1 = \varphi p_2$

$$\dagger p_1 = \dagger p_2$$

||

Although the primary p ≡ <u>input</u> does not have property A, the following derivation

$$[t,f,i] \xrightarrow{T\dagger F,\ T=\underline{\text{inp}}} [i^+,\ (t,f),\ i^-]$$

establishes that it behaves very like the constant primary, except that the value it produces is not determined by inspecting the source program or the variable table.

3.10 <u>Lemma 7</u>

If the expressions e_1 and e_2 have property A, then so does the block primary

$$p \equiv \underline{\text{let}}\ y = e_1 e_2$$

where y is a name.

Proof

$$\mu[\underline{\text{let}}\ y = e_1 e_2] = \mu e_1,\ \text{let}(\alpha y),\ \mu e_2,\ \text{tel}(\alpha y)$$

where

$$\text{let}(x) \quad : (W = V@x,\ V@x = T,\ T = W)$$

$$\text{tel}(x) \quad : (W{\downarrow}F,\ V@x = W)$$

Thus

$$[t,f,v] \xrightarrow{\mu e_1} [\varphi e_1 v,\ (t,f),\ \psi e_1 v] \quad \text{since} \quad A[e_1],$$

$$\xrightarrow{\text{let}(\alpha y)} [\psi e_1 v[\alpha y],\ (t,f),\ U[\psi e_1 v][\alpha y][\varphi e_1 v]]$$

$$\xrightarrow{\mu e_2} [\varphi e_2 v',\ (\psi e_1 v[\alpha y],(t,f)),\psi e_2 v'] \quad \text{since} \quad A[e_2],$$

$$\text{where} \quad v' = U[\psi e_1 v][\alpha y][\varphi e_1 v]$$

$$\xrightarrow{\text{tel}(\alpha y)} [\varphi e_2 v',(t,f),\ U[\psi e_2 v'][\alpha y][\psi e_1 v[\alpha y]]]$$

and thus the block primary p has property A, with

$$\varphi\ [\underline{\text{let}}\ y = e_1 e_2] = \lambda v \varphi e_2 [U[\psi e_1 v][\alpha y][\varphi e_1 v]]$$

$$\psi\ [\underline{\text{let}}\ y = e_1 e_2] = \lambda v U[\psi e_2 [U[\psi e_1 v][\alpha y][\varphi e_1 v]]][\alpha y][\psi e_1 v[\alpha y]] \qquad ||$$

3.11 Lemma 8

If the expressions $e_1, e_2, \ldots \ldots, e_k$ possess property A, then so does the compound primary

$$p \equiv \underline{\text{begin}}\ e_1; e_2; \ldots \ldots; e_k\ \underline{\text{end}}$$

Proof

We prove, by induction on the index k, that if

$$\beta_k = \mu[\underline{\text{begin}}\ e_1;\ldots\ \ldots;e_k\ \underline{\text{end}}]$$

then

$$[t,f,v] \overset{\beta}{\rightarrow}{}^{k} [[\varphi e_k.\psi e_{k-1}.\ \ldots\ \ldots\ .\psi e_1]v,(t,f),[\psi e_k.\ \ldots\ \ldots\ .\psi e_1]v]$$

for if $k = 1$, $\beta_1 = \mu[\underline{\text{begin}}\ e_1\ \underline{\text{end}}] = \mu e_1$ and hence

$[t,f,v] \overset{\beta}{\rightarrow}{}^{1} [\varphi e_1 v,(t,f),\ \psi e_1 v]$ since $A[e_1]$ and so hypothesis is established for $k = 1$. Consider now $k > 1$, we have

$$\mu[\underline{\text{begin}}\ e_1;\ldots\ \ldots;e_k\ \underline{\text{end}}] = \mu e_1,\text{off},\mu e_2,\text{off},\ldots\ \ldots,\text{off},\mu e_k$$

$$= \mu[\underline{\text{begin}}\ e_1;\ldots\ \ldots;e_{k-1}\ \underline{\text{end}}],\text{off},\mu e_k \text{ and hence}$$

$$[t,f,v] \overset{\beta}{\rightarrow}{}^{k-1}[[\varphi e_{k-1}.\psi e_{k-2}.\ \ldots\ .\psi e_1]v,(t,f),[\psi e_{k-1}.\ \ldots\ .\psi e_1]v]$$

by the inductive hypothesis

$$\overset{\text{off}}{\rightarrow}\ [t,f,[\psi e_{k-1}.\ \ldots\ \ldots\ .\psi e_1]v]$$

$$\overset{\mu e}{\rightarrow}{}^{k}\ [[\varphi e_k.\psi e_{k-1}.\ \ldots\ \ldots\ .\psi e_1]v,(t,f),[\psi e_k.\ \ldots\ \ldots\ .\psi e_1]v]$$

and hence the hypothesis is established for all $k > 0$.

The compound primary p thus has property A as required and

$$\varphi[\underline{\text{begin}}\ e_1;\ldots\ \ldots;e_k\ \underline{\text{end}}] = \varphi e_k.\psi e_{k-1}.\ \ldots\ .\psi e_1$$

$$\psi[\underline{\text{begin}}\ e_1;\ldots\ \ldots;e_k\ \underline{\text{end}}] = \psi e_k.\psi e_{k-1}.\ \ldots\ .\psi e_1$$

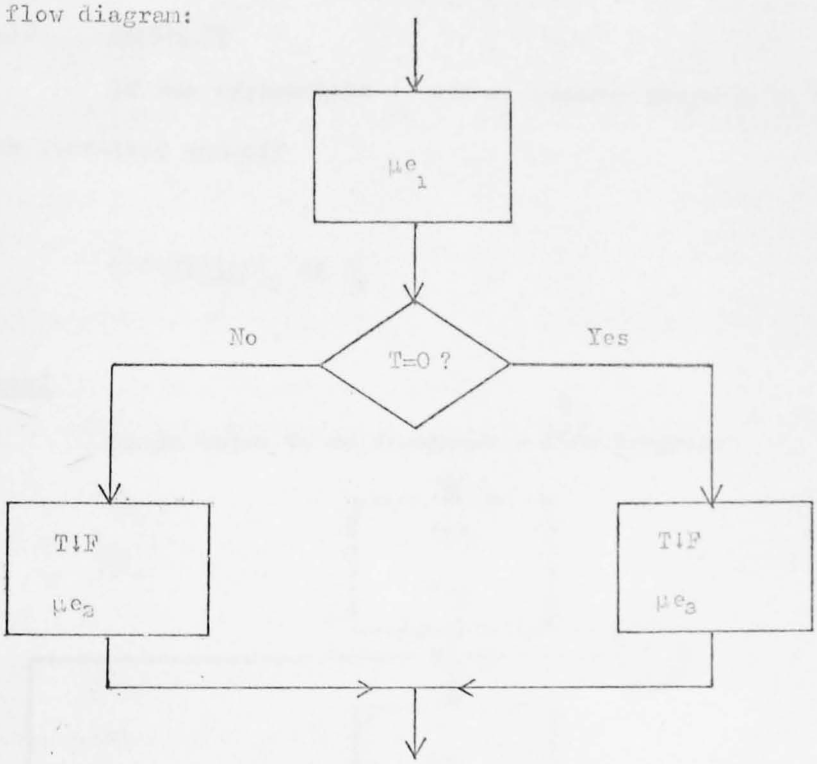
3.12 Lemma 9

If the expressions e_1, e_2 and e_3 possess property A then so does the conditional primary

$$p \equiv \underline{if}\ e_1\ \underline{then}\ e_2\ \underline{else}\ e_3$$

Proof .

We can write the translation given in Chapter 2 in a symmetric form, which excludes reference to the indirection table, by constructing a flow diagram:



Noting the derivation, for some e with A[e],

$$[t,f,v] \overset{T\downarrow F}{\rightarrow} [f^+, f^-, v]$$

$$\overset{\mu e}{\rightarrow} \quad [\varphi ev, f, \psi ev] \quad \text{since} \quad A[e],$$

and that the condition in the above diagram selects between two alternative functions to apply to the environment, rather than two alternative updated environments, we have

$$[t,f,v] \xrightarrow{\mu e_1} [\varphi e_1 v, (t,f), \psi e_1 v] \quad \text{since} \quad A[e_1],$$

$$\rightarrow \quad [\underline{EO}[\varphi e_1 v][\varphi e_3][\varphi e_2][\psi e_1 v], (t,f), \underline{EO}[\varphi e_1 v][\psi e_3][\psi e_2][\psi e_1 v]]$$

and hence p has property A with

$$\varphi[\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3] = \lambda v \underline{EO}[\varphi e_1 v][\varphi e_3][\varphi e_2][\psi e_1 v]$$

$$\psi[\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3] = \lambda v \underline{EO}[\varphi e_1 v][\psi e_3][\psi e_2][\psi e_1 v]$$

||

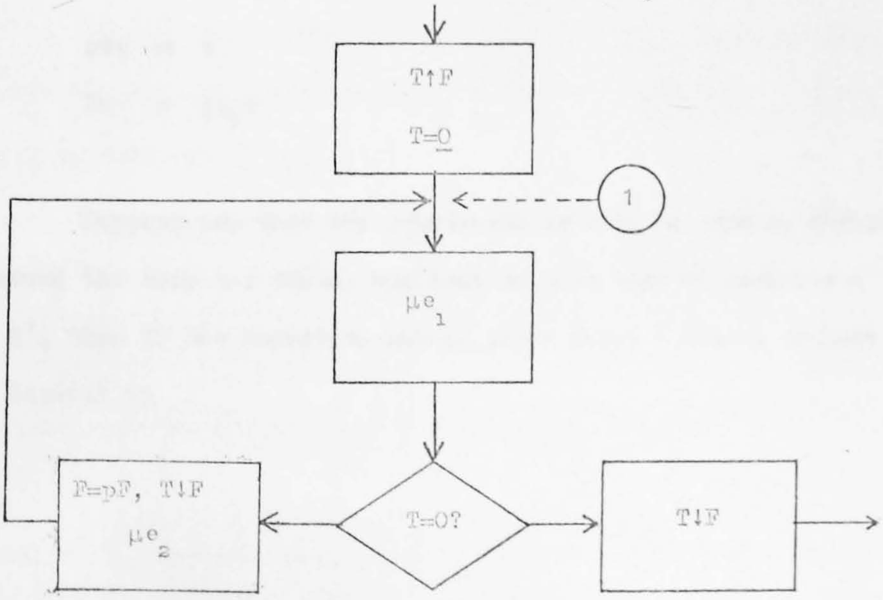
3.13 Lemma 10

If the expressions e_1 and e_2 possess property A, then so does the iterating primary

$$p \equiv \underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2$$

Proof

As in Lemma 9, we construct a flow diagram:



We postulate that there exist functions α and β with the property that, if we enter the diagram at 1 with $\xi = [t,f,v]$ then we exit (if at all) with the environment

$$\xi' = [\alpha tv,\ f,\ \beta v]$$

and we prove this by induction on k, the number of times the condition ($T=0$) proves false.

If the condition never proves false ($k=0$) then we have the derivation:

$$[t,f,v] \xrightarrow{\mu e_1} [\varphi e_1 v,\ (t,f),\ \psi e_1 v] \quad \text{since} \quad A[e_1]$$

$$\xrightarrow{T\downarrow F} [t,f,\psi e_1 v]$$

and hence hypothesis is true with

$$\alpha tv = t$$

$$\beta v = \psi e_1 v$$

Suppose now that the hypothesis is true in case we should go round the loop k-1 times, and that in this case we have $\alpha = \alpha'$, $\beta = \beta'$, then if the condition should prove false k times, we have the derivation

$$[t,f,v] \xrightarrow{\mu e_1} [\varphi e_1 v,\ (t,f),\ \psi e_1 v] \quad \text{since} \quad A[e_1]$$

$$\xrightarrow{F=pF,\ T\downarrow F} [f^+,\ f^-,\ \psi e_1 v]$$

$$\xrightarrow{\mu e_2} [\varphi e_2[\psi e_1 v],\ f,\ \psi e_2[\psi e_1 v]] \quad \text{since} \quad A[e_2]$$

$$\rightarrow [\alpha'[\varphi e_2[\psi e_1 v]][\psi e_2[\psi e_1 v]],\ f,\ \beta'[\psi e_2[\psi e_1 v]]\ ]$$

by inductive hypothesis, since condition proves false after k-1 more loops. Now this establishes existence of α and β with the form

$$\alpha t v = \alpha'[\varphi e_2[\psi e_1 v]][\psi e_2[\psi e_1 v]]$$

$$\beta v \ = \beta'[\psi e_2[\psi e_1 v]]$$

However these are not definitional equations, rather identities satisfied by α and β in case (e_1 and e_2 are such that) the condition does not prove true on the first loop.

To get the correct definitions of α and β, consider the partial derivations, corresponding to the two arms of the diagram chosen by the condition $T=0$,

i) $[t,f,v] \xrightarrow{T\downarrow F} [f^+,\ f^-,\ v]$ for the right branch, and

ii) $[t,f,v] \xrightarrow{F=pF,\ T\downarrow F} [f^{-+},\ f^{--},\ v]$

$$\xrightarrow{\mu e_2} [\varphi e_2 v,\ f^-,\ \psi e_2 v] \quad \text{since} \quad A[e_2]$$

$$\rightarrow [\alpha[\varphi e_2 v][\psi e_2 v],\ f^-,\ \beta[\psi e_2 v]\ ]$$

for the left branch.

Employing these in a derivation, where we consider entry to the diagram at 1

$$[t,f,v] \xrightarrow{\mu e_1} [\varphi e_1 v,\ (t,1),\ \psi e_1 v]$$

$$\rightarrow [E\underline{O}[\varphi e_1 v][\lambda vt][\lambda v\alpha[\varphi e_2 v][\psi e_2 v]][\psi e_1 v], f,$$

$$E\underline{O}[\varphi e_1 v][\lambda vv][\lambda v\beta[\psi e_2 v]][\psi e_1 v]\ ]$$

which defines α and β by

$$\alpha tv = E\underline{O}[\varphi e_1 v][\lambda vt][\lambda v\alpha[\varphi e_2 v][\psi e_2 v]][\psi e_1 v]$$

$$\beta v = E\underline{O}[\varphi e_1 v][\lambda vv][\beta\cdot\psi e_2][\psi e_1 v]$$

Thus

$$\alpha = Y\lambda\alpha\lambda t\lambda vE\underline{O}[\varphi e_1 v][\lambda vt][\lambda v\alpha[\varphi e_2 v][\psi e_2 v]][\psi e_1 v]$$

$$\beta = Y\lambda\beta\lambda vE\underline{O}[\varphi e_1 v][\lambda vv][\beta\cdot\psi e_2][\psi e_1 v]$$

Now to establish property A, we have simply to construct a derivation for the whole diagram

$$[t,f,v] \xrightarrow{T\uparrow F,\,T=\underline{O}} [\underline{O},\ (t,f),\ v]$$

$$\rightarrow [\alpha\underline{O}v,\ (t,f),\ \beta v]$$

and hence p has property A and we have

$$\varphi p = \lambda v \alpha \underline{O} v = \alpha \underline{O}$$

$$\psi p = \lambda v \beta v \ = \beta$$

which, when expanded, give

$$\varphi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2] \ = \ [Y\lambda\alpha\lambda t\lambda vE\underline{O}[\varphi e_1 v][\lambda vt][\lambda v\alpha[\varphi e_2 v][\psi e_2 v]][\psi e_1 v]]\underline{O}$$

$$\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2] \ = \ Y\lambda\beta\lambda vE\underline{O}[\varphi e_1 v][\lambda vv][\beta.\psi e_2][\psi e_1 v] \qquad ||$$

3.14 Before we can state and prove the requisite lemmas about the application primary and the function definition primary, we must introduce some alternative notation for handling the overlap of the sections of stack identified by F and M. First note that if we simply allowed the following derivation

$$[t,f,m,v] \ \overset{M=F}{\rightarrow} \ [t,f,f,v]$$

then the essential information correlating the elements pointed to by M and F has been lost. If the derivation were to continue

$$\overset{cM=T}{\rightarrow} \ [t,f,\ (t,f^-),\ v]$$

then the resulting environment is incorrect, and should be

$$[t,\ (t,f^-),(t,f^-),\ v]$$

This latter form is only slightly better since it is open to the same mistake again.

Our solution is to write

$$[t,f,m,v] \overset{M=F}{\rightarrow} [t,\Lambda,f,v] \quad F \equiv M$$

where we have employed two devices. Firstly, we make a note of the equivalence of M and F, and secondly we retain only one copy of the information originally identified by F, choosing to consider F as identifying a newly empty stack. The equivalence allows us to interpret any reference to F which affects the information identified by M (this fact being established automatically by its direct inapplicability to F) to correctly adjust the model. Thus the above continuation becomes

$$\overset{cM=T}{\rightarrow} \quad [t,\Lambda,(t,f^{-}),v] \quad M \equiv F$$

Note that the equivalence is not altered by this operation. Effectively, when an operation is found inapplicable to an environment, the equivalences are inspected to see if an alternative interpretation is justified. Usually we shall anticipate the operations to be applied and use the equivalences to adjust the model appropriately.

Thus we consider F and M as two independant stacks in the model, but with a restricted mode of access to M. The equivalences can be generalised to include, for example $F \equiv n^k M$ with an obvious meaning.

Consider the following derivation

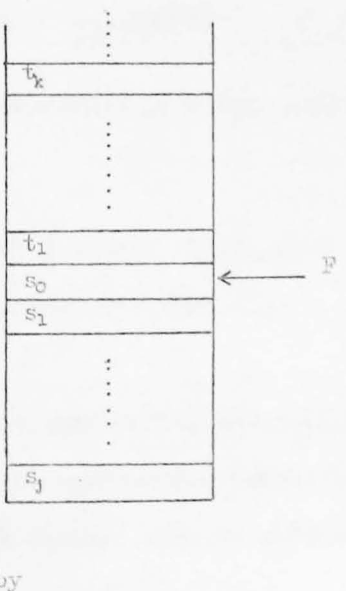
$$[t,f,m,v] \xrightarrow{T=M} [m,f,\Lambda,v] \qquad T \equiv M$$

$$\xrightarrow{M=F} [m,\Lambda,f,v] \qquad F \equiv M$$

$$\xrightarrow{T\uparrow F} [\Lambda,(m,\Lambda),f,v] \qquad F \equiv nM, \quad T \equiv cF$$

where we have manipulated the equivalences in an obvious way

3.15 A second device which we require, which will be obvious from a glance at the diagrammatic representation of function evaluation in Chapter 2, is to be able to identify information stacked <u>above</u> a pointer (M or F), via that pointer. In particular, we must be able to collect the actual parameter from an area of the stack which the current model would consider undefined. Consider the stack shown below



which we shall denote by

$$f = (t_k, t_{k-1}, \ldots \ldots, t_1 ; (s_0, (s_1, \ldots \ldots, (s_j, \Lambda)\ldots))\ldots)$$

where, for example, applying the operation $F = nF$ results in

$$f = (t_k, t_{k-1}, \ldots \ldots, t_2 ; (t_1, (s_0, \ldots \ldots, (s_j, \Lambda)\ldots))\ldots)$$

The sequence $\{t_i\}$ can be considered as semi-infinite where, if $i > k$ then $t_i = \Omega$ which is undefined.

3.16 Lemma 11

The function definition primary

$p \equiv$ lambda $y_1, \ldots \ldots, y_k . e$

where $k \geq 0$ has property A.

Proof

We have

$\mu p = \text{fun}(x), \ldots \ldots, \text{nuf}(x)$

where x is the index associated with this occurrence of the primary, and where

fun(i) : (T↑F, T = A@i, → B@i, A@i : F = nnM)

nuf(i) : (W = cnM, → W, B@i :)

so we see that p has the same effect as a constant primary, except that the value is found in the indirection table. Since we consider this value to be an element of the domain, then we conclude that p has property A.

Further

$$\phi[\underline{\text{lambda}}\ y_1, \dots\ \dots, y_k \cdot e] = \lambda vv$$

however, we do not choose to represent

$$\varphi[\underline{\text{lambda}}\ y_1, \dots\ \dots, y_k \cdot e]$$

in the metalanguage, always referring to it in this form. $||$

3.17 Lemma 12

If the aprimary a, the expressions e and $e_1, \dots\ \dots, e_s$ have property A, and if

$$\varphi a = \varphi[\underline{\text{lambda}}\ y_1, \dots\ \dots, y_k \cdot e]$$

where $y_1, \dots\ \dots, y_k$ are distinct names, then for some k and s,

$$p \equiv a(e_1, \dots\ \dots, e_s)$$

also has property A.

Proof

We note that

$$\mu p = \mu a,\ \ \text{mrk},\ \ \mu e_1, \dots\ \dots, \mu e_s\ \ \text{cll},\ \ \text{unm}$$

where

```
mrk    :  (T↑F, T=M, M=F, F=nF)
cll    :  (T↑F, cnM=$, → cM, $:)
unm    :  (F=M, M=T, T↓F)
```

and that

$$\mu[\underline{\text{lambda}}\ y_1,\ldots\ \ldots,y_k.e] = \text{fun}(x),\ \text{for}(\alpha y_1),\ldots\ \ldots,\text{for}(\alpha y_k),\text{chk},$$
$$\mu e,\text{dot},\text{rof}(\alpha y_k),\ldots\ \ldots,\text{rof}(\alpha y_1),\text{nuf}(x)$$

where

```
fun(i)  :  (T↑F, T=A@i, → B@i, A@i : F=nnM)
for(i)  :  (F=nF, T=cF, W=V@i, V@i=T, cF=W)
chk     :  (T↓F)
dot     :  (cM=T, T↓F)
rof(i)  :  (V@i=T, T↓F)
nuf(i)  :  (W=cnM, → W, B@i:)
```

We observe also that execution of 'cll' causes a branch to the function reference returned by a, which corresponds to A@x. The sequence of instructions which are executed is thus well defined but of length determined by s and k. Because of this, our derivation will also be of a length determined by s and k, and hence the proof will require inductive reasoning to establish the correct form of the environment on certain occasions.

Where this need arises we shall rely upon an informal induction which consists of exhibiting the first, i^{th} and $i + 1^{st}$ (general) and last steps in a sequence of similar derivation steps and checking that the notation denotes correctly the special case of no (zero) elements in the sequence. We begin then by constructing a derivation as follows

$$[t,f,m,v] \xrightarrow{\mu a} [\varphi av, (t,f),m,\psi av] \quad \text{since } A[a],$$

$$\xrightarrow{T\uparrow F, T=M} [m, (\varphi av,(t,f)),\Lambda,\psi av] \; T \equiv M$$

$$\xrightarrow{M=F} [m,\Lambda,(\varphi av,(t,f)),\psi av] \; M \equiv F$$

$$\xrightarrow{F=nF} [m,(\Omega,\Lambda), (\varphi av,(t,f)),\psi av] \; M \equiv pF$$

Using the equivalence we can write this as

$$[m,\Lambda,(\Omega;(\varphi av,(t,f))),\psi av] \; M \equiv pF$$

$$\xrightarrow{\mu e_1} [\varphi e_1[\psi av],(m,\Lambda),(\Omega;(\varphi av,(t,f))),\psi e_1[\psi av]] \; M \equiv ppF \quad \text{Since } A[e_1]$$

......

$$\xrightarrow{\mu e_i} [r_i,(r_{i-1},\dots \quad \dots,(r_1,(m,\Lambda)) \dots),(\Omega;(\varphi av,(t,f))),$$
$$[\psi e_i \bullet \dots \quad \dots \bullet \psi e_1 \bullet \psi a]v] \; M \equiv p^{i+1}F$$

$$\text{where} \quad 1 \le j \le i \Rightarrow r_j = [\varphi e_j \bullet \psi e_{j-1} \bullet \dots \quad \dots \bullet \psi e_1 \bullet \psi a]v$$

$$\xrightarrow{\mu e_{i+1}} [r_{i+1},(r_i,\dots,(r_1,(m,\Lambda)) \dots),(\Omega;(\varphi av,(t,f))),$$
$$[\psi e_{i+1} \bullet \dots \bullet \psi e_1 \bullet \psi a]v] \; M \equiv p^{i+2}F$$

......

$$\xrightarrow{\mu e}{}^{s} \quad [r_s,(r_{s-1},\ldots \quad \ldots,(r_1,(m,\Lambda))\ \ldots\ ),(\Omega;(\varphi av,(t,f))),[\psi e_s.\ \ldots\ .\psi e_1.\psi a]v] \quad M \equiv p^{s+1}F$$

$$\xrightarrow{T\uparrow F,\ cnM = \$} \quad [\Omega,(r_s,\ldots \quad \ldots,(r_1,(m,\Lambda))\ \ldots\ ),(\$;(\varphi av,(t,f))),v'] \quad M \equiv p^{s+2}F$$

where $v' = [\psi e_s.\ \ldots \quad \ldots\ .\psi e_1.\psi a]v$

We must check that this form correctly denotes the environment in case $s = 0$, which, when substituted yields

$$[\Omega,(m,\Lambda),(\$;(\varphi av,(t,f))),\psi av] \quad M \equiv p^2F$$

which is correct. We continue now at the entry point, to which transfer has been made.

$$[\Omega,(r_s,\ldots \quad \ldots,(r_1,(m,\Lambda))\ \ldots\ ),(\$;(\varphi av,(t,f))),v'] \quad M \equiv p^{s+2}F$$

$$\xrightarrow{F=nnM} \quad [\Omega,(r_s,\ldots \quad \ldots,r_1;(m,\Lambda)),(\$;(\varphi av,(t,f))),v'] \quad M \equiv p^2F$$

We can rewrite this as

$$[\Omega,(r_k,\ldots \quad \ldots,r_1;(m,\Lambda)),(\$;(\varphi av,(t,f))),v'] \quad M \equiv p^2F$$

where if $i > s$ then $r_i = \Omega$ is undefined. What this means is that we anticipate the need for k actual parameters by making sure that k are available. Our extended definition of the r_i would therefore be

$$1 \leq i \leq s \Rightarrow r_i = [\varphi e_i . \psi e_{i-1} . \ldots \ \ldots . \psi e_1]v$$

$$s < i \leq k \Rightarrow r_i = \Omega$$

to which we would give the correct interpretation in case $s \geq k$.
The derivation continues

$$\xrightarrow{for(x_1)} [\Omega,(r_k,\ldots\ \ldots,r_2;(v'x_1,(m,\Lambda))),m',Uv'x_1 r_1]\ M \equiv p^3 F$$

where $m' = (\$;(\varphi av,(t,f)))$

......

$$\xrightarrow{for(x_i)} [\Omega,(r_k,\ldots\ \ldots,r_{i+1};(v'x_i,\ldots\ \ldots,(v'x_1,(m,\Lambda))),m',$$

$$U[\ \ldots\ [Uv'x_1 r_1]\ \ldots\ ]x_i r_i]\ M \equiv p^{i+2} F$$

(next we split up "for(x_{i+1})" into its component parts)

$$\xrightarrow{F=nF,T=cF} [r_{i+1},(r_k,\ldots\ \ldots,r_{i+2};(r_{i+1},(v'x_i,\ldots\ \ldots,(v'x_1,(m,\Lambda)))),m',$$

$$U[\ \ldots\ [Uv'x_1 r_1]\ \ldots\ ]x_i r_i]\ M \equiv p^{i+3} F$$

$$\xrightarrow{W=V@x_{i+1},V@x_{i+1}=T,cF=W} [r_{i+1},(r_k,\ldots\ \ldots,r_{i+2};(v'x_{i+1},\ldots\ \ldots,(v'x_1,(m,\Lambda)))),m',$$

$$U[\ \ldots\ [Uv'x_1 r_1]]x_{i+1} r_{i+1}]\ M \equiv p^{i+3} F$$

which follows since $y_1,\ldots\ \ldots,y_k$ all distinct implies $x_1,\ldots\ \ldots,x_k$ are distinct and hence $[U[\ \ldots\ [Uv'x_1 r_1]\ \ldots\ ]x_i r_i]x_{i+1} = v'x_{i+1}$.
We prefer to consider the value of t as now undefined.

......

$$\xrightarrow{\mathrm{for}(x_k)} [\Omega,(v'x_k,\ldots\ \ldots,(v'x_1,(m,\Lambda))\ \ldots\ ),(\$;(\varphi av,(t,f))),$$
$$\tilde{U}v'<x_1,\ldots\ \ldots,x_k><r_1,\ldots\ \ldots,r_k>]\ M \equiv p^{k+2}F$$

where $\tilde{U}v<x_1,\ldots\ \ldots,x_k><r_1,\ldots\ \ldots,r_k> = U[U\ \ldots\ [Uvx_1r_1]\ \ldots\ x_{k-1}r_{k-1}]x_kr_k$

Again we must check that our notation takes care of the case **k** = 0, which it can be seen to do trivially. The derivation continues

$$\xrightarrow{\mathrm{chk},\mu e} [\varphi ev'',(v'x_k,\ldots\ \ldots,(v'x_1,(m,\Lambda))\ \ldots\ ),(\$;(\varphi av,(t,f))),\psi ev'']$$
$$M \equiv p^{k+2}F \text{ since } A[e]$$

where $v'' = \tilde{U}v'<x_1,\ldots\ \ldots,x_k><r_1,\ldots\ \ldots,r_k>$

$$\xrightarrow{\mathrm{dot}} [v'x_k,(v'x_{k-1},\ldots\ \ldots,(v'x_1,(m,\Lambda))\ \ldots\ ),(\$;(\varphi ev'',(t,f))),\psi ev'']$$
$$M \equiv p^{k+1}F$$

$$\xrightarrow{\mathrm{rof}(x_k)} [v'x_{k-1},(v'x_{k-2},\ldots\ \ldots,(v'x_1,(m,\Lambda))\ \ldots\ ),(\$;(\varphi ev'',(t,f))),$$
$$U[\psi ev'']x_k[v'x_k]]\ M \equiv p^{k+1}F$$

......

$$\xrightarrow{\mathrm{rof}(x_{i+1})} [v'x_i,(v'x_{i-1},\ldots\ \ldots,(v'x_1,(m,\Lambda))\ \ldots\ ),(\$;(\varphi ev'',(t,f))),$$
$$U[\ \ldots\ [U[\psi ev'']x_k[v'x_k]]\ \ldots\ ]x_{i+1}[v'x_{i+1}]]\ M \equiv p^{i+2}F$$

$$\xrightarrow{\mathrm{rof}(x_i)} [v'x_{i-1},(v'x_{i-2},\ldots\ \ldots,(v'x_1;(m,\Lambda))\ \ldots\ ),(\$;(\varphi ev'',(t,f))),$$
$$U[\ \ldots\ [U[\psi ev'']x_k[v'x_k]]\ \ldots\ ]x_i[v'x_i]]\ M \equiv p^{i+1}F$$

......

$$\xrightarrow{\mathrm{rof}(x_1)} [m,\Lambda,(\$;(\varphi ev'',(t,f))),\tilde{U}[\psi ev'']<x_k,\ldots\ \ldots,x_1><v'x_k,\ldots\ \ldots,v'x_1>]$$
$$M \equiv p^2F$$

Again the form is correct if $k = 0$ is substituted. Execution is now transferred to the return address \$, which has been correctly preserved in cnM, and we return to execute the "unm" corresponding to the original "mrk".

$$\xrightarrow{F=M} \quad [m, (\varphi e v'', (t,f)), \Lambda, \widetilde{U}[\psi e v'']\langle x_k, \ldots \ \ldots, x_1 \rangle \langle v' x_k, \ldots \ \ldots, v' x_1 \rangle] \quad F \equiv M$$

$$\xrightarrow{M=T,\,T\downarrow F} \quad [\varphi e v'', (t,f), m, \widetilde{U}[\psi e v'']\langle x_1, \ldots \ \ldots, x_k \rangle \langle v' x_1, \ldots \ \ldots, v' x_k \rangle]$$

where reversing the k-tuples in the last element is justified because $x_1, \ldots \ \ldots, x_k$ are all distinct. This form of the environment finally establishes property A for the application primary p, since v' and v'' are defined solely in terms of v.

$$v' = [\psi e_s \cdot \ldots \ \ldots \cdot \psi e_1 \cdot \psi a]v$$

$$v'' = \widetilde{U} v' \langle x_1, \ldots \ \ldots, x_k \rangle \langle r_1, \ldots \ \ldots, r_k \rangle$$

$$r_i = [\varphi e_i \cdot \psi e_{i-1} \cdot \ldots \ \ldots \cdot \psi e_1 \cdot \psi a]v$$

$$\text{if } 1 \le i \le s$$

If we write $\sigma_i = \psi e_i \cdot, \ldots \ \ldots, \cdot \psi e_1 \cdot \psi a$ which is the accumulated side effect after the ith actual parameter has been evaluated, then $v' = \sigma_s v$; if $1 \le i \le s$ then

$$r_i = [\varphi e_i \cdot \sigma_{i-1}]v = \varphi e_i [\sigma_{i-1} v]$$

$$\varphi[a(e_1,\ldots\ \ldots,e_s)] = \lambda v\varphi e[\widetilde{U}[\sigma_s v]<\alpha y_1,\ldots\ \ldots,\alpha y_k><\varphi e_1[\sigma_0 v],\ \ldots\ ,\varphi e_k[\sigma_{k-1} v]>]$$

$$\psi[a(e_1,\ldots\ \ldots,e_s)] =$$

$$\lambda v\widetilde{U}[\psi e[\widetilde{U}[\sigma_s v]<\alpha y_1,\ldots\ \ldots,\alpha y_k><\varphi e_1[\sigma_0 v],\ \ldots\ ,\varphi e_k[\sigma_{k-1} v]>]]<\alpha y_1,\ldots\ \ldots,\alpha y_k>$$

$$<[\sigma_s v]x_1,\ \ldots\ ,[\sigma_s v]x_k>$$

where, if $1 \leq i \leq s$ then $\sigma_i = \psi e_i \cdot \ldots\ \ldots \cdot \psi e_1 \cdot \psi a$

$$\sigma_0 = \psi a$$

if $i > s$ then $\sigma_i v$ is undefined. ||

As we have pointed out in Chapter 1, the symbolic model which we have developed is unable to handle the concept of a vector properly. The analyses however are included here in order to exemplify the difficulties involved with this concept.

3.18 Lemma 13

If the expressions e_1 and e_2 possess property A, then so does the first alternative block primary

$$p \equiv \underline{let}\ y = \underline{row}\ e_1\ e_2$$

where y is a name.

Proof

We have

$$\mu p = \mu e_1, arr(\alpha y), \mu e_2, wor(\alpha y)$$

with

$$\mathrm{arr}(i) : (W{=}F,\ T{\uparrow}F,\ F{=}n^{T}F,\ W{\uparrow}F,\ W{=}nW,\ T{=}V@i,\ V@i{=}W)$$

$$\mathrm{wor}(i) : (W{\downarrow}F,\ V@i{=}W,\ F{=}cF)$$

and hence we can construct a derivation as follows

$$[t,f,v] \xrightarrow{\mu e_1} [\varphi e_1 v,(t,f),\psi e_1 v]$$

$$\xrightarrow{W=F} [\varphi e_1 v,(t,f),\psi e_1 v]\ W \equiv F$$

$$\xrightarrow{T\uparrow F} [\varphi e_1 v,(\varphi e_1 v,(t,f)),\psi e_1 v]\ W \equiv pF$$

$$\xrightarrow{F=n^{T}F} [\varphi e_1 v,(\Omega_k,(\Omega_{k-1},\ldots\ \ldots,(\Omega_1,(k\ ,(t,f)))\ \ldots\ )),\psi e_1 v] W \equiv p^{k+1}F$$

where $k = \varphi e_1 v$ and Ω_1 is undefined.

$$\xrightarrow{W\uparrow F,W=nW} [\varphi e_1 v,((t,f),(\Omega_k,\ldots\ \ldots,(\Omega_1,(k,\Lambda)))),\psi e_1 v] W \equiv p^{k+1}F,\ cF \equiv pW$$

By the equivalences, we note that

$$w = (\Omega_k,\ldots\ \ldots,\Omega_1;(k,\Lambda))$$

and hence

$$\xrightarrow{T=V@\alpha y,V@\alpha y=W} [\psi e_1 v[\alpha y],((t,f),\Lambda),U[\psi e_1 v][\alpha y](\Omega_k,\ldots\ \ldots,\Omega_1;(k,\Lambda))]V@\alpha y{\equiv}p^{k+1}F$$

$$\xrightarrow{\mu e_2} [\varphi e_2 v',(\psi e_1 v[\alpha y],((t,f),\Lambda)),\psi e_2 v'] \quad \text{since} \quad A[e_2]$$

$$\text{where } v' = U[\psi e_1 v][\alpha y](\Omega_k,\ldots\ \ldots,\Omega_1;(k,\Lambda)),$$

note however that we have necessarily dropped the equivalence.

$$\xrightarrow{W{\downarrow}F,V@\alpha y=W} \quad [\varphi e_2 v',((t,f),\Lambda),U[\psi e_2 v'][\alpha y][\psi e_1 v[\alpha y]]\ ]$$

$$\xrightarrow{F=cF} \quad [\varphi e_2 v',(t,f),U[\psi e_2 v'][\alpha y][\psi e_1 v[\alpha y]]\ ]$$

which provəs property A for the first alternative block primary. However, the last step in the derivation F=cF also results in a deallocation of storage which is not depicted here. Had it been possible to determine the effect of μe_2 upon the equivalence $V@\alpha y \equiv p^{k+1}F$ then the resulting equivalences would have noted references to the storage represented by $(p^{k+1}F,k)$ which could then have been deleted. The danger arises with the incorrectly specified side effect of a primary such as

(<u>let</u> a = <u>row</u> n b:= a)

However, for the record, we note the semantics which we have derived above.

$$\varphi[\underline{\text{let}}\ y = \underline{\text{row}}\ e_1\ \ e_2] = \lambda v \varphi e_2[U[\psi e_1 v][\alpha y](\Omega_k, \ldots\ \ \ldots,\Omega_1;(k,\Lambda))]$$

$$\psi[\underline{\text{let}}\ y = \underline{\text{row}}\ e_1\ \ e_2] = \lambda v U[\psi e_2[U[\psi e_1 v][\alpha y](\Omega_k, \ldots\ \ \ldots,\Omega_1;(k,\Lambda))]][\alpha y][\psi e_1 v[\alpha y]] \ ||$$

The second form of the block primary is neglected here since its analysis cannot add any substantial information concerning the difficulties discovered here.

3.19 Lemma 14

If the aprimary a and the primary p' possess property A, and if φa returns a vector reference $(x_k, \ldots\ \ \ldots, x_1;(k,\Lambda))$ for some k, and if $\varphi p'$

returns a value i, in the range $0 \leq i \leq k$ then the subscripted variable **primary**

$$p \equiv a@p'$$

also possesses property A.

<u>Proof</u>

We have

$$\mu p \;=\; \mu a,\ \mu p',\ \text{ind},\ \text{iva}$$

where

$$\begin{aligned} &\text{ind} : (W{\downarrow}F,\ T{=}n^{T}W) \\ &\text{iva} : (T{=}cT) \end{aligned}$$

and so we construct a derivation as follows

$$[t,f,v] \xrightarrow{\mu a} [\varphi av,(t,f),\psi av] \quad \text{since} \quad A[a]$$

$$\xrightarrow{\mu p'} [[\varphi p'.\psi a]v,(\varphi av,(t,f)),[\psi p'.\psi a]v]$$

$$\xrightarrow{W\downarrow F} [[\varphi p'.\psi a]v,(t,f),[\psi p'.\psi a]v] \quad W \equiv (x_k,\ldots\ \ldots,x_1;(k,\Lambda))$$

$$\xrightarrow{T=n^{T}W} [(x_k,\ldots\ \ldots,x_{i+1};(x_i,\ldots\ \ldots,(k,\Lambda)\ldots)\ldots),(t,f),$$
$$[\psi p'.\psi a]v] \quad W \equiv p^{i}T$$

where $i = [\varphi p'.\psi a]v$

$$\xrightarrow{T=cT} [x_i,(t,f),[\psi p'.\psi a]v]$$

and hence p has property A. If we let Ixy denote the operation of taking the y th element of the vector reference x then we have

$$\varphi[a@p'] = \lambda v I[\varphi a v][\varphi p'[\psi a v]]$$

$$\psi[a@p'] = \psi p' . \psi a$$

||

3.20 Lemma 15

If a and p' satisfy the conditions of Lemma 14 and if e is an expression with property A, then the subscripted assignment primary

$$p \equiv a@p' := e$$

has property A.

Proof

Since

$$\mu[a@p' := e] = \mu a, \mu p', \text{ind}, \mu e, \text{ias}$$

where

$$\text{ias} : (W{\downarrow}F,\ cW{=}T)$$

we can continue the derivation from Lemma 14.

$$[(x_k, \ldots \ \ldots, x_{i+1}; (x_i, \ldots \ \ldots, (k, \Lambda) \ldots) \ldots), (t,f), [\psi p' . \psi a]v]$$

μe
$\rightarrow$ $[[\varphi e.\psi p'.\psi a]v,((x_k,\ldots \;\; \ldots,x_{1+1};(x_1,\ldots \;\; \ldots,(k,\ )\ \ldots\ )\ \ldots\ ),(t,f)),$
$[\psi e.\psi p'.\psi a]v]$

ias
$\rightarrow$ $[[\varphi e.\psi p'.\psi a]v,(t,f),A[\psi e.\psi p'.\psi a][\varphi av][\varphi p'[\psi av]][[\varphi e.\psi p'.\psi a]v]]$

where Aabcd updates the variable table a so that the c th element of the vector b contains d. This is only an approximation however since not <u>all</u> references to the vector b will be in the variable table. For example

a@(a@(1):= 1)

would stack the value of a, then alter a_1 and then access a_1. A model which can take care of all aspects of sharing of data is highly desirable.

However, we have established property A and the result

$$\varphi[a@p':= e] \;=\; \varphi e.\psi p'.\psi a$$ ||

It seems that the aspects of sharing which arise when vectors are handled in this way present substantial problems for our model. It is difficult to imagine a model which does not have difficulties in this quarter and this suggests an approach through language design. Sufficiently powerful constraints upon the user could probably keep the multiplicity of references to vectors small but this would exclude such devices (as the example above shows) as subscripted variables occurring as indices, and many others. We shall not further concern ourselves with the semantics we have derived here for vectors, other than to discuss the difficulties in Chapter 5.

3.21 In the next Chapter we shall discuss the semantics we have derived here and apply them to obtain some results about ALEPH. To conclude this Chapter, let us summarise these results. The fifteen Lemmas combine via Theorem 1 to establish property A absolutely for each value phrase. Further, for most of the value phrases we have been able to derive an acceptable form of the semantics as functions applicable to an n-tuple of values, representing the variables in the phrase. If these semantic functions should prove intuitively acceptable, as a representation of the semantics, as the reader has derived such, informally, from Chapter 2, then the analysis of this Chapter is a proof of correctness of the run time machine. Alternatively, the semantic functions can be considered as a tool to be used in determining the definition of the language recognised by the run time machine. In this sense, the description of Chapter 2 is considered mere speculation. It is to be hoped that the applications of the next Chapter will do much to reinforce the reader's acceptance of the semantics of ALEPH as correct in one of these senses.

CHAPTER 4

4.0 This Chapter presents some results which use the functional description of ALEPH semantics derived in Chapter 3. In an attempt to make these results as independant as possible of that Chapter and of the way the functional description was derived, certain conceptual details will be repeated here. However, the notation introduced in Section 3.1 will not be repeated. Certain of the functional descriptions themselves will be reviewed in order to give them the informal commentary we avoided in Chapter 3. The semantics associated with vectors however, will not be reviewed until the next Chapter.

We use the functional description to show the equivalence of the block primary and a certain application primary, and then to analyse the role of the assignment primary in expressions. This latter analysis leads to the introduction of a special operator which allows for a restricted form of parallel assignment. Next we analyse the iterating primary, giving three alternative forms of the semantics. Firstly we establish its equivalence with a replacement text, secondly a simpler form of the functional description is given and finally we prove that a certain form of proposition is unchanged by the iteration. In a somewhat different vein, we turn to the problem of control transfer in ALEPH, in particular the problem of encoding an arbitrary flow diagram and two alternative solutions are given. All the results presented in this Chapter are intended to demonstrate some features of the functional description or of ALEPH which are relevant to a discussion of the design of ALEPH which we undertake in the next Chapter.

4.1 A value phrase V has associated with it two functions φV and ψV which together describe its semantics: φV is the value function, and ψV is the side effect function. Both of these functions are applicable to an n-tuple of values which represents the current assignment of values to names occurring in V. If y is a name in V, then αy is the index of y in the n-tuple v, and $v[\alpha y]$ is the current value associated with y. The way in which α associates the index with y is as follows. If V occurs in the program p then p is scanned from left to right as written, and each name occurring directly in a block primary or a function definition primary is numbered, starting at one. The index thus assigned to y is αy and the maximum index assigned is n.

For a given n-tuple v, φVv is the value of the value phrase V evaluated with this assignment of values and ψVv is the side effect of evaluating V, that is to say an n-tuple derived from v. The primitive functions used to define φV and ψV include U_n. $U_n vix$ is the n-tuple which differs from v only in that the i^th^ element contains x. We abbreviate U_n to U when no confusion can arise. We have also

$$\tilde{U}_{nk}\, v\langle i_1, \ldots\ \ldots, i_k\rangle\langle x_1, \ldots\ \ldots, x_k\rangle =$$

$$U_n[\ \ldots\ U_n[U_n v i_1 x_1] i_2 x_2\ \ldots\ ] i_k x_k$$

which we abbreviate to $\tilde{U}$. Finally we have the equality function E which is defined so that Eijxy yields x if i and j are equal as values and yields y otherwise.

In the remainder of this Chapter we shall use the following notational conventions:

$e, e_1, e_2, \ldots \ldots$ are expressions,

$a, p, p_1, p_2, \ldots \ldots$ are primaries, a is an aprimary,

$y, y_1, y_2, \ldots \ldots$ are names,

$r, r_1, r_2, \ldots \ldots$ are operands, i.e. components of an expression

The next six Sections repeat the functional descriptions, described in the Section of Chapter 3 used to identify them, and then briefly interpret them informally.

4.2(3.4) If $e \equiv p_1 \,\underline{op}_1\, p_2 \,\underline{op}_2\, \ldots \ldots \underline{op}_{k-1}\, p_k$ then

$$\varphi e = \lambda v[\alpha_1 v]\underline{op}_1[\alpha_2 v]\underline{op}_2 \ldots \ldots \underline{op}_{k-1}[\alpha_k v]$$

$$\psi e = \psi p_k . \psi p_{k-1} . \ldots . \ldots . \psi p_1$$

where

$$\alpha_i = \varphi p_i . \psi p_{i-1} . \ldots . \ldots . \psi p_1 \qquad 1 \le i \le k$$

The principal semantic feature of an expression is that the side effect is determined solely by the constituent primaries and the order in which they appear, independantly of the actual operations performed upon them and the priorities of these operations.

The side effect of the expression is the side effect of evaluating the constituent primaries in order from left to right as written. The value of the i th constituent primary is affected by the side effect of the first i-1 constituent primaries and the value of the expression as a whole is then obtained by combining the values of the constituent primaries according to the priorities and primitive definitions of the operators appearing.

4.3 (3.7) $\varphi[y := e] = \varphi e$

$\psi[y := e] = \lambda v U[\psi e v][\alpha y][\varphi e v]$

The value of the assignment primary is the value of the expression assigned. The side effect is to install this value as the new current value corresponding to the name y. However, the side effect of evaluating e is experienced before the assignment.

4.4 (3.10) $\varphi[\underline{\text{let}}\ y = e_1\ e_2] = \lambda v \varphi e_2 [U[\psi e_1 v][\alpha y][\varphi e_1 v]]$

$\psi[\underline{\text{let}}\ y = e_1\ e_2] = \lambda v U[\psi e_2 [U[\psi e_1 v][\alpha y][\varphi e_1 v]]][\alpha y][\psi e_1 v[\alpha y]]$

The expression e_2 is evaluated in the environment obtained by assigning the value of e_1 to the name y. The value of the block primary is the value of e_2, thus evaluated. The side effect of the block primary is the side effect of this evaluation except that the original value corresponding to y is restored. Strictly, the above interpretation tells us that the restored value is that value of y after e_1 has been evaluated. Previously we had imposed the informal restriction that the only form e_1 could take, if it contained a legitmate occurrence of y, was a function definition primary, in which case $\psi e_1 = \lambda vv$. The above definition of the side effect of the block primary allows us to state this restriction more formally as: e_1 is such that for any v, $\psi e_1 v[\alpha y] \equiv v[\alpha y]$. That is to say, the side effect of e_1 is transparent to y. This ensures that the correct value is restored; however, since the value of y may be undefined until the assignment due to the block primary takes place, an additional restriction is to say that e_1 may not contain an evaluated occurrence of y.

4.5 (3.11) $\varphi[\underline{\text{begin}}\ e_1;\ldots\ \ldots;e_k\ \underline{\text{end}}] = \varphi e_k . \psi e_{k-1} . \ldots\ \ldots . \psi e_1$

$\psi[\underline{\text{begin}}\ e_1;\ldots\ \ldots;e_k\ \underline{\text{end}}] = \psi e_k . \psi e_{k-1} . \ldots\ \ldots . \psi e_1$

The compound primary provides for the evaluation of an expression e_k in the light of the side effect of evaluating the expressions $e_1,\ldots\ \ldots,e_{k-1}$. The value and side effect of the compound primary are just those yielded by this evaluation.

4.6 (3.12) $\varphi[\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3] = \lambda v \underline{\text{EO}}[\varphi e_1 v][\varphi e_3][\varphi e_2][\psi e_1 v]$

$\psi[\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3] = \lambda v \underline{\text{EO}}[\varphi e_1 v][\psi e_3][\psi e_2][\psi e_1 v]$

Depending upon the value produced when e_1 is evaluated, one of the expressions e_2 or e_3 is evaluated, in the light of the side effect of the evaluation of e_1. The value and side effect of the conditional primary are the value and side effect of this evaluation. Note that the expression evaluated is e_2 if e_1 yields a non-zero value, and e_3 otherwise. This is consistent with our choice of 0 as false and -1 as true. However, it does show that <u>if</u> $x \neq 0$ <u>then</u> .... would be given the same interpretation as <u>if</u> x <u>then</u> ..... The former is to be preferred because it is explicit about the condition being tested.

4.7 (3.13) $\varphi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2] = [Y\lambda f\lambda x\lambda vE\underline{O}[\varphi e_1 v][\lambda vx][\lambda vf[\varphi e_2 v][\psi e_2 v]][\psi e_1 v]]\underline{O}$

$\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2] = [Y\lambda g\lambda vE\underline{O}[\varphi e_1 v][\lambda vv][g.\psi e_2][\psi e_1 v]$

Direct interpretation of this description is a little difficult because of the recursive form. Unlike the previous descriptions, it does not correspond to the informal description of Chapter 2 in an obvious way. The value of the iterating primary is tentatively set to zero. The expression e_1 is evaluated and if this yields zero, then the tentative value is accepted. Otherwise e_2 is evaluated as a new tentative value for the iterating primary and the whole process is repeated. The side effect is defined by this iterated evaluation of e_1 followed by e_2, where e_1 is evaluated at least once and exactly one more time than e_2. In Section 4.11 we shall give a proof of these statements.

4.8 (3.17) If $\varphi a = \varphi[\underline{\text{lambda}}\ y_1, \ldots\ \ldots, y_k . e]$ where $y_1, \ldots\ \ldots, y_k$ are distinct names, then

$$\varphi[a(e_1,\ldots\ \ldots,e_s)] = \lambda v \varphi e[\widetilde{U}[\sigma_s v] < \alpha y_1,\ldots\ \ldots,\alpha y_k >< \varphi e_1[\sigma_0 v],\ldots\ \ldots,\varphi e_k [\sigma_{k-1} v]>]$$

$$\psi[a(e_1,\ldots\ \ldots,e_s)] = \lambda \widetilde{U}[\psi e[\widetilde{U}[\sigma_s v] < \alpha y_1,\ldots\ \ldots,\alpha y_k >< \varphi e_1[\sigma_0 v],\ldots\ \ldots,\varphi e_k [\sigma_{k-1} v]>]] < \alpha y_1,\ldots\ \ldots,\alpha y_k >< [\sigma_s v][\alpha y_1],\ldots\ \ldots,[\sigma_s v][\alpha y_k]>$$

where, if $1 \leq i \leq s$

$$\sigma_i = \varphi e_i . \psi e_{i-1} . \ldots\ \ldots . \psi e_1 . \psi a$$

otherwise $\sigma_i v$ is undefined.

The value and side effect of the application primary is that of evaluating the function body with the value of each actual parameter assigned to the corresponding name. The side effect is however transparent to the names occurring in the actual parameter list in the sense that their original values are restored on exit. Each actual parameter is evaluated in the light of the side effect of evaluating the aprimary and then all the preceding actual parameters in order from left to right. Similarly, the side effect of evaluating the actual parameters affects the evaluation of the function body.

Names other than $y_1,\ldots\ \ldots,y_k$ which occur in e have values which are current at the time of function calling, rather than function definition time. The way α associates an index with each name in a program is important at this point. Since the correlation between names is lost, the name referred to from e includes e in its scope.

This seems to be a desirable effect of α. Consider for example

$$\underline{\text{let}}\ y_1 = e_1$$

$$\underline{\text{let}}\ y_2 = \underline{\text{lambda}}.e_2$$

$$\underline{\text{let}}\ y_3 = e_3$$

$$y_2()$$

when y_1 and y_3 are the same name. Any occurrence of this name in e_2 refers to y_1. Had the name correlation been retained, the ability to update y_3 from e_2 would have been obtained.

4.9 If we consider the case of an application primary which has only a single parameter, we get a useful subcase of the results of Section 4.6. If $\psi a = \lambda vv$ and $\varphi a = \varphi[\underline{\text{lambda}}\ y.e]$ then

$$\varphi[a(e_1)] = \lambda v\varphi e[\widetilde{U}[\psi e_1 v]\langle\alpha y\rangle\langle\varphi e_1 v\rangle] \text{ from Section 4.6}$$

$$= \lambda v\varphi e[U[\psi e_1 v][\alpha y][\varphi e_1 v]] \text{ from defn. of } \widetilde{U}$$

$$= \varphi[\underline{\text{let}}\ y = e_1 e] \qquad \text{from Section 4.3}$$

$$\psi[a(e_1)] = \lambda v\widetilde{U}[\psi e[\widetilde{U}[\psi e_1 v]\langle\alpha y\rangle\langle\varphi e_1 v\rangle]\langle\alpha y\rangle\langle\psi e_1 v[\alpha y]\rangle \text{ from Section 4.6}$$

$$= \lambda vU[\psi e[U[\psi e_1 v][\alpha y][\varphi e_1 v]][\alpha y][\psi e_1 v[\alpha y]] \text{ from defn. of } \widetilde{U}$$

$$= \psi[\underline{\text{let}}\ y = e_1 e] \qquad \text{from Section 4.3}$$

We have established therefore, that in case y does not occur in e_1 then (<u>lambda</u> y.e)(e_1) and (<u>let</u> y = e_1 e) are interchangeable. The case when y is assigned the value of a recursive function and thus y occurs in e_1 cannot be handled in this way but requires the explicit programming of the paradoxical combinator Y. How this can be done in a particular case is the subject of Example 10 in Appendix 3. This equivalence of two combinations of ALEPH phrases is an instance of the application of the functional form of the semantics to establish a result about ALEPH. The remaining Sections of this Chapter are devoted to similar applications.

4.10 Consider now the expression

$$e \equiv y_1 := y_2 \ \underline{op} \ y_2 := y_1$$

for some basic operator <u>op</u> and names y_1 and y_2. We have

$$\varphi e = \varphi[y_1 := y_2 \ \underline{op} \ y_2 := y_1]$$

$$= \varphi[y_2 \ \underline{op} \ y_2 := y_1]$$

$$= \lambda v[\varphi y_2 v] \ \underline{op} \ [\varphi[y_2 := y_1]v]$$

$$= \lambda v[v[\alpha y_2]] \ \underline{op} \ [\varphi y_1 v]$$

$$= \lambda v[v[\alpha y_2]] \ \underline{op} \ [v[\alpha y_1]]$$

$$\begin{aligned}
\psi e &= \psi[y_1 := y_2 \ \underline{op}\ y_2 := y_1] \\
&= \lambda v U[\psi[y_2 \ \underline{op}\ y_2 := y_1]v][\alpha y_1]\varphi[y_2 \ \underline{op}\ y_2 := y_1]v] \\
&= \lambda v U[[\psi[y_2 := y_1].\psi y_2]v][\alpha y_1][v[\alpha y_2] \ \underline{op}\ v[\alpha y_1]] \\
&= \lambda v U[U[\psi y_1 v][\alpha y_2][\varphi y_1 v]][\alpha y_1][v[\alpha y_2] \ \underline{op}\ v[\alpha y_1]] \\
&= \lambda v U[Uv[\alpha y_2][v[\alpha y_1]]][\alpha y_1][v[\alpha y_2] \ \underline{op}\ v[\alpha y_1]]
\end{aligned}$$

From these results, we see that y_2 takes the original value of y_1 and y_1 takes the value of $y_2 \ \underline{op}\ y_1$ in terms of the original values of y_1 and y_2. Thus if we were to include in the set of basic operators, an operator <u>et</u>, say with the property

$$r_1 \ \underline{et}\ r_2 \ = \ r_1$$

that is to say, it discards its second operand, then the side effect of $e \equiv y_1 := y_2 \ \underline{et}\ y_2 := y_1$ is

$$\psi e = \lambda v U[Uv[\alpha y_2][v[\alpha y_1]]][\alpha y_1][v[\alpha y_2]]$$

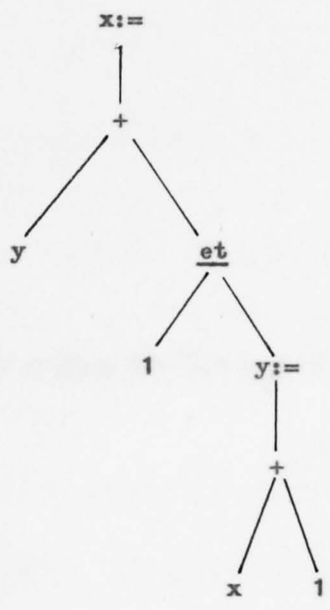
which exchanges the original values of y_1 and y_2. If <u>et</u> were considered an arithmetic (or logical) operator, it would be given a lower priority than <u>or</u>. This would mean that combination of its operands was performed last in expression evaluation. This gives a reasonable interpretation to

$$y_1 := r_1 \ \underline{et}\ y_2 := r_2 \ \underline{et} \ldots \ \ldots \underline{et}\ y_k := r_k$$

which is to assign the "expressions" $r_1, \ldots \ \ldots, r_k$ to the names $y_1, \ldots \ \ldots, y_k$, where each r_i is evaluated with the original values of $y_1, \ldots \ \ldots, y_k$. For example, the assignment

$$x := y + 1 \ \ \underline{et}\ \ y := x + 1$$

is constructed as



and hence the values of x and y are incremented and exchanged. By assuming that each r_i does not contain an occurrence of __et__, the above statement could be proved by a simple proof such as the one given at the beginning of this Section. An operator such as __et__ is useful for transforming some recursive functions to iterative ones. For example, the function

```
f = lambda y_1,... ...,y_k .
    if e_1 then f(r_1,... ...,r_k) else e_2
```

can be written as

```
f = lambda y_1,... ...,y_k .
    begin
          while e_1 do
          y_1:= r_1 et ... et y_k:= r_k;
          e_2
    end
```

In particular, we have

```
hcf = lambda a,b.
    if a mod b ≠ 0 then hcf(b,a mod b) else b
```

and

```
hcf = lambda a,b.
    begin
          while a mod b ≠ 0 do
          a:= b et b:= a mod b;
          b
    end
```

These forms of the highest common factor function should be compared with the one introduced in Section 2.18 which makes use of the result given at the beginning of this Section.

4.11 Turning our attention now to the iterating primary, a useful result is that

$$p_2 \equiv \underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2$$

and

$$p_1 \equiv \underline{\text{if}}\ e_1\ \underline{\text{then}}\ \underline{\text{begin}}\ e_2\ ;\ \underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2\ \underline{\text{end}}\ \underline{\text{else}}\ 0$$

have the same side effect. Trivially, they do not have the same value (for consider the case when e_2 is evaluated exactly once). Considering the side effect, we expand as follows

$$\psi p_1 = \lambda v E\underline{O}[\varphi e_1 v][\lambda vv][\psi[\underline{\text{begin}}\ e_2\ ;\ \underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2\ \underline{\text{end}}]][\psi e_1 v]$$

$$= \lambda v E\underline{O}[\varphi e_1 v][\lambda vv][\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2].\psi e_2][\psi e_1 v]$$

$$\psi p_2 = Y\lambda g\lambda v E\underline{O}[\varphi e_1 v][\lambda vv][g.\psi e_2][\psi e_1 v]$$

$$= \lambda v E\underline{O}[\varphi e_1 v][\lambda vv][[Y\lambda g\lambda v E\underline{O}[\varphi e_1 v][\lambda vv][g.\psi e_2][\psi e_1 v]].\psi e_2][\psi e_1 v]$$

by characteristic property of Y, $YF = F[YF]$, thus

$$\psi p_2 = \lambda v E\underline{O}[\varphi e_1 v][\lambda vv][\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2].\psi e_2][\psi e_1 v]$$

$$= \psi p_1$$

This form of replacement definition for the iterating primary is sufficient to define the semantics (side effect only) of that primary. Effectively

$$\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2] = \psi[\underline{\text{if}}\ e_1\ \underline{\text{then}}\ \underline{\text{begin}}\ e_2;\ \underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2\ \underline{\text{end}}\ \underline{\text{else}}\ 0]$$

says that the side effect of while e_1 do e_2 is just the side effect of evaluating e_1, if the value of e_1 is zero. Otherwise it is the side effect of evaluating e_1, e_2 and while e_1 do e_2 in that order. The same effect as this replacement can be obtained by making the recursion explicit at run time as follows

$$p_3 \equiv (\underline{\text{let}}\ y = \underline{\text{lambda}}.\ \underline{\text{if}}\ e_1\ \underline{\text{then}}\ \underline{\text{begin}}\ e_2;\ y()\ \underline{\text{end}}\ \underline{\text{else}}\ 0$$
$$y())$$

which can also be shown to have the same side effect as p_1 if y does not occur in e_1 or e_2. The primary

$$p_4 \equiv (\underline{\text{let}}\ y_1 = \underline{\text{lambda}}\ y_2.\underline{\text{if}}\ e_1\ \underline{\text{then}}\ y_1(e_2)\ \underline{\text{else}}\ y_2$$
$$y_1(0))$$

has the same value and side effect as while e_1 do e_2, if y_1 and y_2 are names which do not occur in e_1 or e_2. However showing this is complicated by the use of y_2 to hold the tentative result of the iterating primary. A somewhat more directly useful result about the iterating primary is deduced in the next Section.

4.12 Let us denote by f^n the product $f. \ldots \ldots .f$, where f occurs $n \geq 0$ times. Thus, we have

$$f^n = f.f^{n-1}, \quad n > 0$$
$$f^0 = \lambda xx$$

If, given v_0, there exists $n \geq 0$ such that

i) $[\varphi e_1 .[\psi e_2 .\psi e_1]^n]\, v_0 = \underline{0}$

and ii) if $0 \leq k < n$ then $[\varphi e_1 .[\psi e_2 .\psi e_1]^k]\, v_0 \neq \underline{0}$

then we can prove

i)′ $\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2]\, v_0 = [\psi e_1 .[\psi e_2 .\psi e_1]^n]\, v_0$

ii)′ $\varphi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2]\, v_0 = \begin{cases} \underline{0} & ,n = 0 \\ [\varphi e_2 .[\psi e_1 .\psi e_2]^{n-1}][\psi e_1 v_0] & ,n \geq 1 \end{cases}$

To prove (i)′ requires an inductive argument as follows. Suppose $n = 0$, then $\varphi e_1 v_0 = \underline{0}$. From the definitions of $\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2$ given in Section 4.5, we have

$\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2]\ v_0 = [Y\lambda g\lambda v\underline{\text{EQ}}[\varphi e_1 v][\lambda vv][g.\psi e_2][\psi e_1 v]]\ v_0$

$= [\lambda v\underline{\text{EQ}}[\varphi e_1 v][\lambda vv][\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2].\psi e_2][\psi e_1 v]]\ v_0$ since $YF = F[YF]$

$= \underline{\text{EQ}}[\varphi e_1 v_0][\lambda vv][\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2].\psi e_2][\psi e_1 v_0]$

$= \psi e_1 v_0$ since $\varphi e_1 v_0 = \underline{0}$

We take as our inductive hypothesis that if $n < n_0$ then $\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2]\ v_0 = [\psi e_1.[\psi e_2.\psi e_1]^n]\ v_0$ where n is the least integer such that $[\varphi e_1.[\psi e_2.\psi r_1]^n]\ v_0 = \underline{0}$ and consider the case $n = n_0 > 0$. We have

$\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2]v_0$

$= \underline{\text{EQ}}[\varphi e_1 v_0][\lambda vv][\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2].\psi e_2][\psi e_1 v_0]$

$= [\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2].\psi e_2][\psi e_1 v_0]$ since $\varphi e_1 v_0 \neq \underline{0}$

$= [\psi e_1.[\psi e_2.\psi e_1]^{n-1}.\psi e_2][\psi e_1 v_0]$ since $n - 1 < n_0$

$= [\psi e_1.[\psi e_2.\psi e_1]^n]\ v_0$

and hence the induction is complete. A similar induction proves (ii)′ ||
This result confirms the interpretation of Section 4.5.

4.13 Hoare (1969) and Dijkstra (1969) both describe a result about an iterating statement which they use to prove the correctness of short programs. Hoare states the result as an axion and Dijkstra deduces it from a definition of the iterating statement similar to the first equivalence of Section 4.9. They state that, if B is a Boolean expression and S is a statement and if the truth of P_A B before S is executed is sufficient for P to be true after S is executed, then if P is true before

<u>while</u> B <u>do</u> S

is executed then it is true afterwards (assuming the iteration terminates).

To state this in terms corresponding to the notation we have used in this dissertation we concern ourselves with $\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2]$ when $\psi e_1 = \lambda v v$. The result we wish to establish is then that, if it is possible to deduce $P[\psi e_2 v]$ given Pv and $\varphi e_1 v \neq \underline{0}$ for some v, then given Pv_0 (and that the iteration terminates) we can deduce $P[\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2]v_0]$.

For from Section 4.10 we have $\exists n \geq 0$ such that, if $0 \leq k < n$ then $[\varphi e_1 . [\psi e_2]^k]v_0 \neq \underline{0}$ and hence

$$Pv_0 \quad , \varphi e_1 v_0 \neq \underline{0} \quad \vdash \quad P[\psi e_2 v_0]$$

$$P[\psi e_2 v_0],[\varphi e_1 . \psi e_2]v_0 \neq \underline{0} \vdash P[[\psi e_2]^2 v_0]$$

....

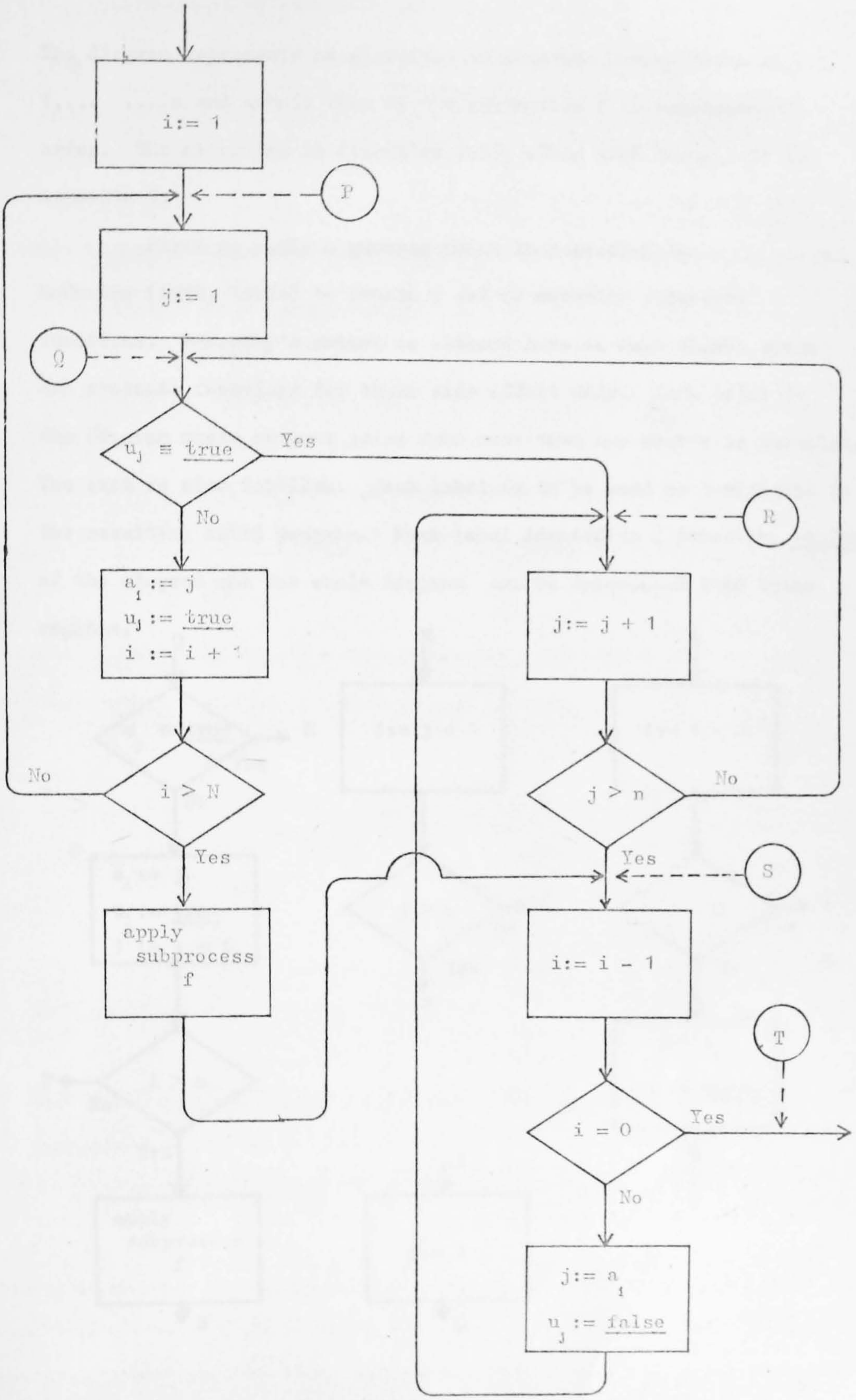
$$P[[\psi e_2]^{n-1} v_0],[\varphi e_1 . [\psi e_2]^{n-1}]v_0 \neq \underline{0} \vdash P[[\psi e_2]^n v_0]$$

that is to say $\qquad P[\psi[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2]v_0]$ ||

The way in which this result is used is not of interest here. Suffice it to say that such propositions about source language statements are usually sufficient to prove correctness of programs using them and further that this form of semantic specification seems to be very suitable for this purpose. Certainly it is a more suitable form of the semantics than our functional description. I suggest however that determining directly that the implementation gives the iterating primary this interpretation would involve a process not very different from the one used in this dissertation, although probably not explicitly determining the semantic functions.

4.14 Finally we turn to the problem of control sequencing in ALEPH. Due to the absence of an explicit control transfer instruction such as the goto in ALGOL, we may not be able to express some algorithms in their most natural form in ALEPH. It is our purpose in this Section to show how an arbitrary flow diagram can be encoded as an ALEPH program. We leave a discussion of how an explicit control transfer instruction affects both the design of ALEPH and the results of this dissertation until Chapter 5.

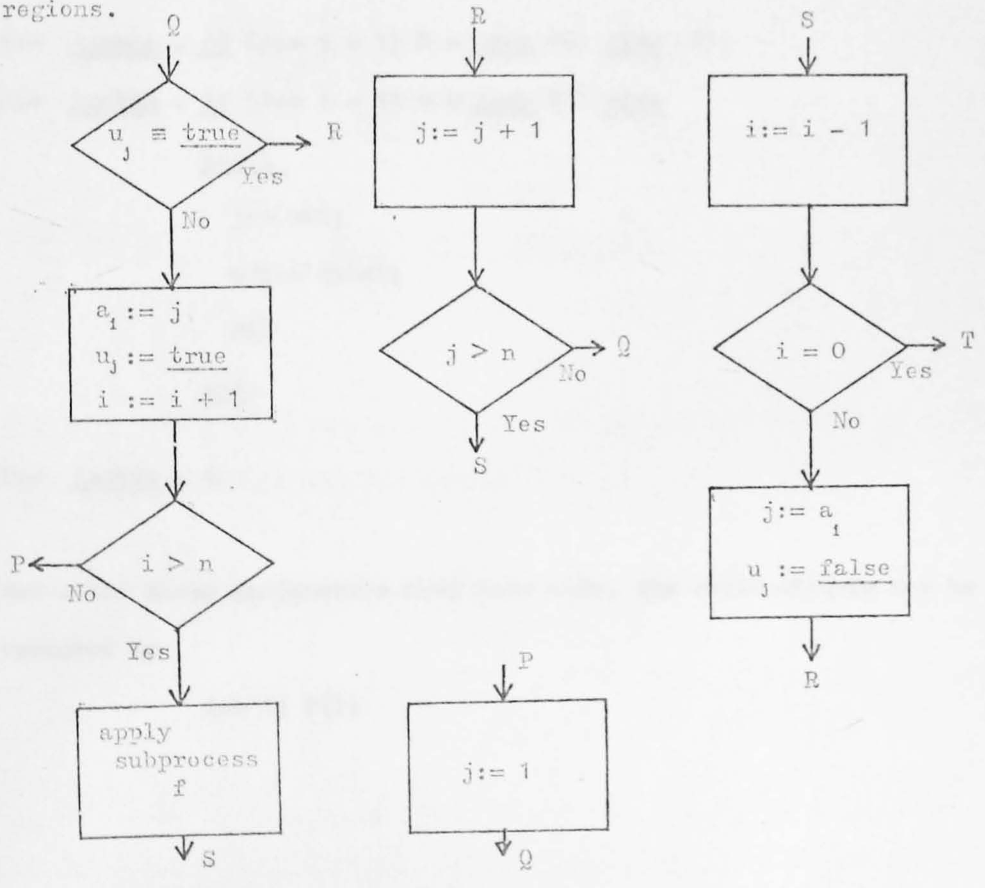
The process to be described will be applied to the following flow diagram



$i := 1$
P
$j := 1$
Q
$u_j \equiv$ true
Yes
No
$a_i := j$
$u_j :=$ true
$i := i + 1$
$i > N$
No
Yes
apply subprocess f
R
$j := j + 1$
$j > n$
No
Yes
S
$i := i - 1$
T
$i = 0$
Yes
No
$j := a_i$
$u_j :=$ false

The diagram represents an algorithm to generate permutations of 1,... ...,n and submit them to the subprocess f in lexicographic order. The algorithm is described fully along with Example 11 in Appendix 3.

First we apply a process which is described in McCarthy (1960, 1962a) to obtain a set of mutually recursive functions. McCarthy's method is altered here to suit ALEPH, which can evaluate functions for their side effect only. Each point in the diagram where control joins from more than one source is labelled. The exit is also labelled. Each label is to be used as a variable in the resulting ALEPH program. Each label identifies a tree-like <u>region</u> of the diagram and the whole diagram can be decomposed into these regions.



Because of this tree-like structure, and because the exit from each region identifies a successor region, we can encode each region as a parameterless function in an obvious way.

```
P:= lambda . begin j:= 1; Q() end;
Q:= lambda . if u@j then R() else
             begin
               a@(i):= j;
               u@(j):= true;
               if (i:= i + 1) > n then
                       begin f(); S() end else P()
             end;

R:= lambda . if (j:= j + 1) > n then S() else Q();
S:= lambda . if (i:= i - 1) = 0 then T() else
             begin
               j:= a@i;
               u@j:= false;
               R()
             end;

T:= lambda . 0
```

Now after these assignments have been made, the whole diagram can be executed by

```
             i:= 1; P();
```

We see that, just before each function has been completed, it calls another function in the scheme. This is extremely wasteful of stack space since we shall have a nest of "marks", one for each label passed in the diagram, and when finally T is called, the nest of marks is complete in the sense that none of the functions P, Q, R, S have yet been exited. The exit from T causes a string of exits to be made and the functions are exited in reverse order to which they were entered. To counteract this, we alter each of the above functions to return as its value the name of the function it selects as its successor. Thus, for example, Q becomes

```
Q:=  lambda . if u@j then R else
          begin
             a@(i):= j;
             u@(j):= true;
             if (i:= i + 1 ) >n then
                   begin f(); S end else P
          end;
```

Now if we alter P, R and S similarly and replace the assignment to T by T:= 0, the scheme can be executed by

```
          i:= 1; let x = P while x ≠ 0 do x:= x();
```

Here we see that each function is exited before the next is entered and hence we have an overhead of only one "mark" on the stack. The complete scheme as presented here is programmed and annotated as Example 11 in Appendix 3.

An alternative way in which the regions identified above can be programmed has been reported by Bohm and Jacopini (1966) and modified by Cooper (1967b, 1968). This consists of using the labels of each region as Boolean variables which permit access to the region. Each region selects a successor by setting its identifier true. Thus the above diagram is encoded as

```
let P = false  let Q = false  let R = false  let S = false  let T = false
while not T do
if P then
  begin P:=  false;  j:= 1;  Q:=  true  end  else
if Q then
  begin Q:= false;
        if u@j then R:=  true  else
        begin a@(i):=  j;  u@(j):=  true;
           if (i:= i + 1) > n then
                 begin f(); S:= true end else P:= true
  end else
if R then
  begin R:=  false;
        if (j:= j + 1) > n then S:= true else Q:= true
  end else
```

```
if S then
   begin S:=  false;
         if (i:= i - 1) = 0 then T:=  true else
         begin
            j:= a@i;  u@(j):=  false;
            R:=  true
         end
   end else
begin
   i:=  1;
   P:=  true
end
```

This is Example 12 of Appendix 3.

An important feature of both the above processes is the existence of an effective procedure for encoding <u>any</u> flow diagram using only the sequencing facilities provided in ALEPH.

4.15 It is to be hoped that the results of this Chapter have demonstrated the way in which the functional description can be used. The results themselves are not very deep. Neither is it true that the functional description can effectively answer every question one can ask about ALEPH. However, if results of the sort derived in this Chapter are required and if it is necessary to establish them directly from the implementation, the functional form of the semantics seems to be a useful intermediate stage.

CHAPTER 5

5.0 The derivation of a functional description of the semantics of ALEPH has been presented as an approach to programming language design, rather than as a contribution to the problem of proving the correctness of an implementation. The reason for this is that the original version of ALEPH and of its implementation were changed substantially in order to make the derivation possible. In Section 3.18 however, we postulated that the correctness of the implementation had been proved, if the derived functional description was informally interpretable as what was intended, and in Chapter 4 have made such an interpretation. Indeed, had the functional description been presented, independantly of its derivation, before the implementation was described, then the derivations of Chapter 3 would have been sufficient to prove correctness of the implementation. It is not necessary however to derive semantic functions in this way, it is necessary only to show that the implementation and the functional description have the same effect.

McCarthy (1962) defines correctness of a compiler in this way and McCarthy and Painter (1967) prove the correctness of a compiler for simple arithmetic expressions according to McCarthy's definition. By means of an abstract analytic syntax for the source language they can give a functional description of its semantics. Similarly the semantics of the object language are defined and then a translator, whose correctness is in question is described. It is then proved that executing the translated expression leaves the value predicted by the functional description in the accumulator. The proof is based on an inductive principle similar to that expounded in Theorem 1 (Section 3.0). The approach presented in this dissertation is obviously very similar to this, differing mainly in details and in the fact that the functional description is regarded as a result rather than data.

It was not possible however, as we have pointed out in Chapter 1, to establish the correctness of the original implementation of ALEPH in the way McCarthy had described. This was for many reasons, in particular involved language concepts and complex implementation features. It was necessary therefore to tailor the language and its processor to the requirements of the proof. Once this process had been begun it was used freely and proved to be a very critical design tool. Some examples of this process of "feedback" can now be given.

5.1 The principle example of feedback affecting the implementation was to force the use of the extremely simple method of associating a value with a variable at run time. This association, described in Chapter 2 and again in Chapter 4, allocates a fixed location to each declaration of a name. The immediate consequences of this are to make the implementation of a "call by name" feature impossible. This can be illustrated by a simple example.

Consider the recursively defined factorial function:

```
let f = lambda n. if n = 0 then 1 else n × f (n - 1)
```

which can be rewritten, to give a simple case of recursively nested blocks, as follows:

```
let f = lambda n. if n = 0 then 1 else
                  let j = 0
                  begin j:= f(n - 1);
                        j × n
                  end
```

Now, if instead of having f return a value as its result, we provide it with an extra parameter, a function to apply to its result, we can rewrite this again as:

```
let f = lambda n,a. a(if n = 0 then 1 else
                      let j = 0
                      begin f(n - 1, lambda x.j:= x);
                            j × n
                      end)
```

The extra actual parameter provided in the recursive call implements one of the features provided by a call-by-name parameter, that is the ability to assign to variables at a previous level of (recursive) nesting. If we make the simple transformation of having "a" apply directly to its arguments "1" and "j × n" then the resulting function is incorrect:

```
let f = lambda n,a. if n = 0 then a(1) else
                    let j = 0
                    begin f(n - 1; lambda x.j:= x);
                          a(j × n)
                    end
```

Within the scope of "j" we now have two methods of updating it:

```
j:= e    and    a(e)
```

but since j is associated with a fixed table entry they both have the same effect. The second form, which was intended to update the previous incarnation of "j", does not work. Any simple implementation of a call-by-name feature using the variable table in this way is therefore going to meet the same difficulty. An alternative way of using the variable table to get over this problem is to note that if the extra actual parameter is a vector reference, that is the address of a stack location, then the separate identification for each incarnation of a name is achieved. The three functions described here are demonstrated in Examples 13, 14, 15 of Appendix 3.

Another way in which feedback affected the implementation was that it often caused the generated machine code to be reorganised, either because the analysis suggested a simpler organisation or because the original organisation was in error. In the latter case, the error showed up in the analysis, rather than giving an unacceptable semantic function. On one occasion an implementation of the iterating primary was found to overwrite the last element stacked before it was entered. This had not shown up in practice because in all the test programs run this value had been unimportant. On another occasion the implementation of the application primary was found to reserve an extra cell after the last actual parameter. This was not incorrect but was a wasteof storage. It appeared in the derivation and was corrected by inserting the macro "chk". The first of these design faults would undoubtedly have appeared but the second need never have been discovered. The analysis convinces that there are no other "bugs" of either type.

The way in which feedback affected the language design was mainly in the exclusion of features for which an adequate description could not be derived. An example is a feature which, although never implemented, was considered and rejected on this basis. The current function aprimary:

<u>this</u>

has null side-effect and returns as its value the function reference corresponding to the function currently being evaluated.

The highest common factor function of Section 2.17 could thus be written:

hcf:= <u>lambda</u> x,y. <u>if</u> x <u>mod</u> y = 0 <u>then</u> y <u>else</u> <u>this</u> (y,x <u>mod</u> y)

This aprimary is trivially implemented by

μ <u>this</u> = ths

where

ths : (T↑F, T=cM)

but cannot be defined by the semantic object language. This is because it does not possess property A ,in that its value is not dependant only on the variable table. It seems unlikely that any simple functional description would be able to describe such a context dependant object.

5.2 The semantic functions derived for vectors are incomplete. The difficulties involve the way in which storage is allocated and shared. By sharing of storage, we mean effectively that there is a multiplicity of references to it by which it can be updated. The main problem is then with guaranteeing that all these references are updated to refer to the updated storage. A simpler aspect of sharing was involved when we created multiple references to the stack in the derivation of the application primary. We took care of these by using equivalence relations which effectively kept a record of multiple references. In addition the model retained only one copy of each element of storage so that it is not necessary to update multiple copies. Extending this approach to vectors is not trivially possible however. As we have seen in Section 3.15 the effect of expression evaluation upon the equivalences expressing sharing of vector storage is not easily determined. Essentially what is required is an extended version of property A which determines this effect. This would in turn require all the lemmas to be reproved to ascertain that each phrase possess this extended property. It seems unlikely that such an extended property could be defined for this language and this symbolic model of the implementation.

The analysis of the implementation of the vector concepts was only worth including here for this reason. It points out that there are major conceptual difficulties involved with the implementation of vectors and to where these difficulties lie. As well as the concept of sharing of storage which we encountered in Section 3.18, there is the concept of allocation and deallocation of storage also encountered in that Section. Specific identification of the operations of allocation and deallocation in the semantic object language would seem to be demanded by the situation observed in Section 3.18.

5.3 The functional description of the semantics is incomplete also in that it does not specify the effect of the input and output streams. Again this could be remedied by using an extended version of property A with the added complexity that this entails. This is indeed substantial, for in addition to φV and ψV we should require to associate with the value phrase V two more functions to update the input and output streams respectively. In addition each of these four functions would obtain values, not only from the variable table but the input stream also, for example the side-effect function for the assignment primary would be:

$$\varphi[y := e] = \lambda v \lambda i U[\psi evi][\alpha y][\varphi evi]$$

In case of primaries defined in terms of many value phrases (e.g. compound primary, application primary) the added complexity would be enormous. We need only look at the compound primary where we see that the simple "dot" product form would be lost due to this increased number of arguments.

5.4 In Section 4.14 we introduced the concept of a <u>region</u> as a means of programming complex control problems in a language which has no explicit control transfer. A region is a parameterless function which returns as its result either another region or zero. Control is then exercised by repeatedly evaluating such a sequence of regions until zero is returned. Landin (1964, 1965b) introduces a similar grouping of phrases called a "program-point". It would have been simple to have implemented regions directly in ALEPH and thus avoided the repeated "marking" and "unmarking" involved in a control loop of the form:

$\underline{\text{while}}$ x $\neq$ 0 $\underline{\text{do}}$ x:= x()

It was not felt however that a region was a sufficiently general feature to warrant this inclusion.

Regions have illuminated the need for a more sophisticated block primary which allows for mutually cross referenced definitions. For example:

$\underline{\text{let}}\ y_1, \dots\ \dots, y_k = e_1, \dots\ \dots, e_k\ \ e$

with the meaning that each y_i is initialised to the value of the corresponding e_i. Apart from the fact that each e_i could refer to $y_1, \dots\ \dots, y_k$ this would have the same meaning as:

$(\underline{\text{lambda}}\ y_1, \dots\ \dots, y_k . e)(e_1, \dots\ \dots, e_k)$

and a correspondingly complex derivation.

The way in which a freely useable control transfer should be implemented in ALEPH is not clear. Certainly one could not allow "labelled expressions" since this would allow some unpleasant abnormalities, such as labels occurring within arithmetic expressions. Anything less than this, such as labelling only the components of a compound primary would be too restricted. This is however the course taken in EULER. The implementation would be correspondingly more complex because of the complex interaction with a block structure. The semantic description of a source language with a control transfer instruction is complicated also by the semantic incompleteness of syntactic units. We have in fact encountered this problem in ALEPH with the application primary. It was necessary to specify the associated function definition primary in order to fully specify the semantic functions. A control structure such as the iterating primary on the other hand combines syntactic and semantic completeness.

5.5 We have described in Section 5.1 how the method of associating a name with a value at run time was forced by the requirement of a simple symbolic model for analysis. In the functional description this choice appears as the function α, applicable to a name, to yield its index into the variable table. Certain of the semantic functions however are independant of α. These are the functions associated with the expression, the compound primary, the conditional primary and the iterating primary. These functions make no explicit mention of the method of access used to obtain the value of a variable, or of the structure of the object to which they are applicable. Consider for instance the side effect function of the conditional primary:

$$\psi[\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3] = \lambda v E\underline{O}[\varphi e_1 v][\psi e_3][\psi e_2][\psi e_1 v]$$

This might be considered as applicable to a much more complex information structure than an n-tuple, where the functions φe_1, ψe_1, ψe_2 and ψe_3 are defined in terms of primitives appropriate to this structure. As such, it would appear still to convey the required meaning. We should expect to derive this form from other implementations with alternative methods of storage allocation. We could not expect however to derive equivalent functions in the case of the block primary, say, even with an appropriate redefinition of U and σ. ALGOL, for example, does not regard the value of a variable in a recursively entered block with the same sanctity as ALEPH (cf. Example 14, Appendix 3). These structure-independant semantic functions might therefore, justifiably be considered with more reverence than the humble background of their simple implementation might suggest.

5.6 The implementation itself is worthy of some reverence however. For, consider again the side effect of the conditional primary. If we were to specify this as:

$$\psi[\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3] \ = \ \lambda v E\underline{Q}[\varphi e_1 v][\psi e_3][\psi e_2]v$$

where the side-effect of the expression e_1 is ignored, then although the semantics are (in a sense) simpler, this is by no means simply implementable. Since the primitives of the semantic object language can be implemented (say, in LISP) there is no doubt that we could implement any such functions which we should consider it suitable to choose. Although the example above seems silly, it would probably be seriously considered by a designer, choosing functions on an ad hoc basis, who had temporarily overlooked the side effect of e_1. The probability of considering such silly functions becomes even higher when they become more complex and so will the probability of accepting them. It would then be impossible to show the equivalence of the functional description and the implementation. Indeed the functional description of the semantics is nothing more than an implementation for a "functional" machine and just as prone to error as the proper implementation. The redundancy introduced by having two semantic descriptions and proving their equivalence adds a measure of reliability to the system.

It ought not to be claimed however that the approach presented in this dissertation, where the functional description is derived from the implementation, is guaranteed to be foolproof. Any formal description of the semantics of a programming language is based on the informal interpretation of certain primitive concepts in the semantic object language. Applying these primitive concepts in the derivation is therefore necessarily a fairly informal affair. Thus we are prone to error at this level. Further, in any practical case, the enormity of the derivations involved will add another level where errors are possible.

5.7 Finally let us summarise the results of this dissertation and see what conclusions can be drawn. ALEPH has been presented as a programming language designed under the constraint of having a functional description of its semantics provided. This constraint led to the development of an iterated approach to language design which simultaneously rationalised the source language, its implementation and its formal description. Inability to determine an acceptable functional description on an ad hoc basis led to the derivation of the semantics from the implementation. By defining the semantics of the object language and describing a translator, the semantics of each source phrase was derived from its translation.

When we consider how we might extend this approach in the case of ALEPH an obvious candidate for consideration is the vector concept. ALEPH is powerless without some method of structuring data but we have not made any impression upon the semantics of vectors here. The main problms have been discussed in Section 5.2 where it is concluded that a substantially different symbolic model is required. An alternative extension however would be to implement lists instead of vectors, with the only updating function beirg "cons". It is then justifiable to consider multiple references to common data as references to distinct copies. This avoids the sharing problem, rather than solving it.

It is not suggested that it should be possible to apply the method of deriving semantic functions to existing high level languages such as ALGOL and their implementations. This is for many reasons already noted in this Chapter and is itself the principal reason why the approach has been considered as a design tool. There would seem to be no reason why this tool should not be used to design much more extensive languages, with a careful choice of implementation and description primitives. The approach is, in general, very disapproving of irrational source language concepts. Ideally, concepts should be simple to describe both procedurally and functionally. Hopefully, these two qualifications correspond to simplicity of description, on the one hand to the machine and on the other, to the programmer.

APPENDIX 1 : GLOSSARY

This glossary collects together terms used in a technical sense in this dissertation. Most of the terms were introduced in Chapter 2 and the references in this glossary to sections of that Chapter identify the section in which the term is first used. The principal object of the glossary is to refresh the reader's memory and hence the definitions may not be complete. An underlined phrase occurring in a definition denotes a suggested cross reference.

The notation of Chapter 3 is used for metasyntactic variables, because it is felt that this is the chapter in which the glossary will be of most use.

$p, p_1, p_2, \ldots$	are primaries,
$e, e_1, e_2, \ldots$	are expressions,
$y, y_1, y_2, \ldots$	are names, and
a	is an aprimary.

Since most of the technical terms defined here refer to ALEPH constructs, this glossary also acts as a language reference manual.

<u>alternative block primary</u>

This primary has two forms, both allocate storage for vectors. The first form

$$p \equiv \underline{let}\ y = \underline{row}\ e_1\ e_2$$

sets up a <u>vector</u> of e_1 elements, each undefined, for use during the evaluation of e_2. The second form

$$p \equiv \underline{let}\ y = \underline{row}\ e_1\ \underline{each}\ e_3\ e_2$$

initialises each element to the value of the expression e_3, which is evaluated once only.

<u>application</u> (2.17)

The only meaningful dereferencing operation which can be performed upon a <u>function reference</u>.

<u>application primary</u> (2.17)

The primary

$$p \equiv a(e_1, \ldots\ \ldots, e_k)$$

calls the function referred to by the aprimary a with actual parameters $e_1, \ldots\ \ldots, e_k$.

<u>aprimary</u> (2.17)

An abbreviation for applicable primary, intended to subclassify those primaries which may return a function or vector reference as their value: variable primary, subscripting primary, parenthesised expression primary, application primary.

<u>assignment primary</u> (2.17)

The primary

$$p \equiv y := e$$

which returns the same value as e but has the <u>side-effect</u> of updating the value corresponding to y (in the <u>variable table</u>) to this value.

<u>basic operator</u> (2.2)

A basic operator is used to construct an expression from primaries:

$$e \equiv p_1 \ \underline{op}_1 \ p_2 \ \underline{op}_2 \ldots \quad \ldots \underline{op}_{k-1} \ p_k$$

Each operator $\underline{op}_i$ denotes a rule whereby values from the domain may be combined to form new values. The way in which the primaries in the above expression are combined is governed by the relative priorities of operators. No basic operator may have a <u>side-effect</u>.

basic symbol (2.0)

A source language program in ALEPH is represented by a sequence of basic symbols. In particular, certain underlined words are basic symbols (eg. let, while, do, etc.)

block primary (2.8)

The primary

$$p \equiv \underline{\text{let}}\ y = e_1\ e_2$$

introduces a name y, with initial value e_1 for use in the expression e_2. The value of the block primary is the value of e_2 and the side-effect is that of evaluating e_2 except that this effect is transparent to y. The expression e_1 may also contribute to the side-effect. See also alternative block primary.

compound primary (2.9)

The primary

$$p \equiv \underline{\text{begin}}\ e_1 ;\ldots\ \ldots ; e_k\ \underline{\text{end}}$$

provides for the grouping of expressions, where expressions are to be evaluated for their side-effect only. The value and side-effect of the compound primary is that of evaluating e_k in the environment produced by the side-effect of evaluating $e_1, \ldots\ \ldots, e_{k-1}$ in that order.

<u>conditional primary</u> (2.10)

The primary

$$p \equiv \underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3$$

which has the value and side-effect of evaluating one of e_2 or e_3 according to the value produced when e_1 is evaluated. If e_1 yields the value <u>zero</u>, then e_3 is selected, otherwise e_2 is selected.

<u>constant primary</u> (2.3)

The primary which allows the denotation of constant values from the <u>domain</u>.

<u>declaration</u> (2.8)

The occurrence of a name directly in the block and function definition primaries.

<u>declared</u> (2.8, 2.16)

The property attributed to a name throughout its <u>scope</u>.

<u>domain</u> (2.0)

The set of <u>values</u> over which an ALEPH program expresses a computation. In the case of this dissertation, the union of the set of integers (suitably truncated) and of function and vector references. The union rather than the direct sum implies type of element is not determinable from the element itself.

effect is null (2.4)

When the side-effect is such that the environment is left unchang ed.

expression (2.1)

The expression

$$e \equiv p_1 \; \underline{op}_1 \; p_2 \; \underline{op}_2 \ldots \; \ldots \underline{op}_{k-1} \; p_k$$

is constructed from primaries and basic operators. Each of the primaries is evaluated in order from left to right. The value of the expression is then the result of combining the value of each primary according to the definition of each basic operator. The side-effect is defined by the order of evaluation only.

first alternative block primary

see alternative block primary.

function definition primary (2.16)

The primary

$$p \equiv \underline{\text{lambda}} \; y_1, \ldots \; \ldots, y_k . e$$

which has null side effect and yields a function reference as its value.

function reference

A value from the domain which summarises the information of a certain function definition primary, in the sense that this information may be accessed by application.

functional description

A description based principally on the primitive operations of functional application and abstraction. Contrasts with the concept of a procedural description.

iterating primary (2.11)

The primary

$$p \equiv \underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2$$

repeatedly evaluates e_1 and e_2 (alternately) while e_1 is not zero. Thus e_1 is evaluated exactly once more than e_2. The side-effect is defined by this order of evaluation. The value is that yielded by e_2 when last evaluated, or zero if it is never evaluated.

marks pointer (2.25)

Functions are called by establishing a mark on the stack, which is a three element entry containing the function reference, return address and previous marks pointer. The marks pointer refers to the latest one of these.

name (2.4)

Syntactically, a name is a sequence of letters. A name identifies a variable and is introduced by a declaration.

null, effect is

see effect is null.

number (2.3)

Particular type of constant primary.

parenthesised expression primary

The primary

p ≡ (e)

used to give an expression the status of an aprimary, or to override priorities of operators.

primary (2.1)

In ALEPH all sequencing constructions are classified syntactically as primaries. A primary yields a value and may have a side-effect. As such a primary includes the usual concept of statement.

<u>procedural</u> description

Essentially a description is procedural if one must apply a procedure (or a machine) to determine its meaning. Ultimately all descriptions are procedural. However, the nature of the primitive operations of the procedure makes a step-wise state-transforming procedure,executing simple state-transforming commands intuitively "more procedural" than a description based on the primitive operation of functional application. In this sense the term procedural is used in this dissertation to qualify the machine language translation of an ALEPH program when this is regarded as a semantic description of that program. See <u>functional</u> description.

<u>scope</u> (2.8)

In the <u>block primary</u> the (syntactic) scope of a name is principally the qualified expression. Similarly in the <u>function definition primary</u> it is the function body. In general however, the scope may be reduced by the nested occurrence of a declaration of the same name within its scope. Also in the case of a block primary, the scope includes the initialising expression, but the method of referencing is restricted here.

<u>second alternative block primary</u>

see <u>alternative block primary</u>

<u>semantics</u> (2.1)

In ALEPH the semantics of a <u>value phrase</u> is a definition of its <u>value</u> and its <u>side-effect</u>.

<u>semantic function</u>

For a value phrase V, one of the functions φV and ψV which respectively determine the value and side effect of V.

<u>sharing</u>

The property which may be possessed by two or more distinctly referenceable components of a data structure, which determines that updating the component by any one of the references effects the value subsequently returned via the other references. Conceptually a problem arises when an address of a block of information is used in the implementation, but separate copies of the block are assumed in the symbolic model.

<u>side-effect</u> (of evaluation) (2.1)

The side-effect of a value phrase is its effect upon the <u>variables</u> of a program. Via the names occurring in the value phrase, the correspondences between name and value are updated. This is principally performed by the <u>assignment primary</u>. Informally, a value phrase may also have a side-effect upon the input and output streams, but in defining the <u>semantics</u> of ALEPH, we choose to ignore this.

<u>stack</u> (2.25)

The principal tool of the implementation is a stacked allocation of storage allowing for recursive evaluation of functions.

<u>subscripting primary</u> (2.21)

The primary

$$p \equiv a \circledcirc p_1$$

used to access the p_1 th element of the vector referred to by a. The aprimary a must yield a <u>vector reference</u> for this to be meaningful.

<u>subscripted assignment primary</u> (2.22)

The primary

$$p \equiv a \circledcirc p_1 := e$$

combines the effect of the assignment and subscripting primaries. The vector referred to by a is updated in that the value of its p_1 th element is set to the value of e.

<u>syntax</u>

In ALEPH, the rules governing the written representation of <u>value phrases</u>.

value (2.0)

A member of the domain. Also the result of evaluating a value phrase.

value phrase (2.1)

Expressions and primaries, components from which ALEPH programs are built. In general a value phrase is defined syntactically by a recursive form involving component value phrases. Rules governing the value and side-effect of a value phrase are defined in terms of this recursive structure.

variable

The correspondence between a name and a value. Subject to to frequent change by the side-effect of evaluating value phrases.

variable primary (2.4)

A name may occur as a primary in an expression. As such it denotes the value currently corresponding to this name.

variable table (2.25)

Particular way in which the correspondence between a name and a value has been implemented. Each declaration determines a table entry at run time which contains the value corresponding to the name declared.

vector (2.20)

A vector of length n contains n + 1 entries, indexed by 0,... ...,n. Unless altered by explicit assignment, the 0th element contains the value n. The elements are accessed via a vector reference using the subscripting or subscripted assignment primaries.

vector reference (2.20)

A particular value from the domain used to access the elements of a vector.

zero

Any member of a distinguished subset of the domain. Membership of this subset is tested to determine conditional alternatives at run time.

APPENDIX 2

This Appendix presents details of the author's implementation of ALEPH for the IBM 360/67 running under the Michigan Terminal System (MTS). These details may illuminate the description of Chapter 2 and add substance to the results of Chapter 3.

The compiler consists of a syntax directed string translator (produced by the author's POSSUM system) and a table driven assembler which are linked together as co-routines. The translator produces a string of three letter mnemonics which the assembler maps into machine code. The resulting object module is acceptable to the MTS loader when provided with a module of input and output routines. The translator consists of an analysis routine, based on the work reported by Conway (1963) and Kanner, Kosinski and Robinson (1965), which operates on a linked-list form of the syntax of the source language. The syntax is put into this form by POSSUM, which is described below.

Pete's Own System for String Manipulation (POSSUM)

POSSUM provides a notation in which string translations may be described. The notation given here is in the form implemented by the author but no details of this implementation are given since POSSUM is intended for use here only as a description language. For the purposes to which POSSUM is put in this dissertation, it is to be hoped that it provides a clear and unambiguous description of a certain string translator, and that a correct implementation of a program to perform this translation is intuitively possible.

An alphabet is a finite set of characters. For our purpose, an alphabet shall always be assumed to contain the letters A,B,.....,Z and the digits 0,1,.....,9. A string over the alphabet A is a finite sequence of characters from A. The set of all strings over A is denoted by A*. In particular the empty string is a member of A*. A translator is a mapping from A* into A* which is one-one. A translation is the image of a string under a translator.

A subset of A* is referred to as a genus. In particular a genus may be empty, finite or (countably) infinite. In POSSUM notation a translator description defines a finite set of genera, classifying the source language (and the object language under translation) and, simultaneously how a source string, which is a member of a given genus, is translated.

The principal items for the construction of a translator description are names (of genera) and strings (over the alphabet A).

A translator description utilises a finite number of genera, each of which it identifies by a name, this identification being made in a declaration statement. A name is formed from letters only, for example

CONDITIONAL VARIABLE PRIMARY

are names of genera. A translator description is enclosed in the brackets BEGIN.....END and consists of a declaration statement followed by a finite set of rules. Each rule defines a unique genus and translation. Thus there is exactly one rule associated with each genus and hence exactly as many declarations in the declaration statement as there are rules.

A declaration statement is a finite list of distinct names, separated by commas, prefixed by the word GENERA and terminated by a semicolon. For example

```
        GENERA    CONDITIONAL, VARIABLE,PRIMARY;
```

is a declaration statement.

Each rule in the finite list of rules consists of a name followed by a (possibly empty) list of options and terminated by a semicolon. An option is prefixed by the sumbol ::= and consists of a prefix followed by a transducer. For example

```
        COMPOUND  ::= ';' . ':OFF,' . EXPR . COMPOUND
                  ::= 'END' ;
```

is a rule defining the genus COMPOUND which has two options. The prefixes are ';' and 'END' respectively and the transducers are .':OFF,' . EXPR . COMPOUND and empty, respectively.

A prefix may be a string enclosed in apostrophes or an action prefix (EMPTY, NUMBER, GEN, NAME, SAVE). A transducer consists of a finite sequence of transduction elements, each prefixed by a dot. A transduction element may be a string enclosed in apostrophes, or a genus or an activity (GET, GETN, GETNUM).

Given a (source) string and a genus we may specify that a translation of the string with respect to the genus succeeds or fails, and it if succeeds we may specify the translated string. Translation with respect to a genus is defined in terms of translation with respect to its list of options. Translation with respect to a list of options is defined as follows.

1. If the list is empty then the translation fails, otherwise the first option is chosen and if the prefix match succeeds then translation proceeds under 2. with respect to this option, otherwise translation is performed with respect to the remaining list of options. Thus we see that the first option whose prefix match succeeds is chosen.

A prefix match succeeds or fails as follows: If the prefix is a string then the match succeeds if and only if the characters of this string are found heading the source string and further, if all the characters matched are letters, then the first character not matched is not a letter. The action prefix NAME will match the longest sequence of letters heading the input and the action prefix NUMBER the longest sequence of digits. Neither will match an empty sequence. The action prefixes GEN, EMPTY and SAVE all match the empty sequence (i.e. they succeed identically).

2. Having selected the first option with a prefix match, the matched string is removed from the input. If further, any spaces appear at the head of the input, these are removed too. We have now got a (possibly) shortened input string and a selected transducer. Translation proceeds with respect to this transducer as follows. Each transduction element is selected in turn. If the element is a string then this string is appended to the output (the translation). If the element is GET, GETN or GETNUM then the appropriate action (defined below) is invoked. If the element is a genus then the input string is translated with respect to this genus (as defined in 1.) and then translation proceeds with respect to the remaining transduction elements with a (possibly shortened) input string.

The effect of the action prefixes and their associated activities is defined below. If at any stage in the process, translation with respect to some intermediate genus should fail, then the whole translation fails and the translated string is undefined. However, an appropriate failure message can be generated by attaching this message to the genus when the genus is declared. For example

```
VARIABLE = 'UNDECLARED - IDENTIFIER'
```

Consider the translator description:

```
BEGIN
      GENERA    SUM, ADDEND, TERM;
      SUM ::= ' ' . TERM . ADDEND;
      ADDEND ::= '+' . TERM . ':ADD,' . ADDEND
             ::= ' ';
      TERM   ::= NUMBER . GETNUM . ',';
END
```

Any sequence of numbers separated by + signs will be matched and translated to an equivalent sequence with the operators in a reverse position. For example 12 + 143 + 17 + 0 would produce the translation:

```
12,143,:ADD,17,:ADD,0,:ADD,
```

where we have anticipated an appropriate action for GETNUM.

Initially translation proceeds with respect to the genus whose definition is provided by the first rule after the declarations.

<u>actions and activities</u>

<u>NUMBER and GETNUM</u>

As mentioned above NUMBER will match any non-empty sequence of digits on the input. The action is to preserve these in a specially provided location. The associated activity GETNUM will move the string of digits currently stored there to the output. The storage for these sequences is static and is overwritten each time NUMBER is successfully matched. The option

```
::= NUMBER . ':NUM:' . GETNUM . ','
```

will match any string which begins with a sequence of digits (e.g. 127+...) and will produce as output the digits matched, preceded by :NUM: and followed by a comma (example :NUM:127,).

GEN and GET

In order to provide some correlation in the output between the elements produced by a single transducer in the translation a device is used called a symbol generator. This is simply a counter which begins at 1 and counts up in ones, increasing by one each time GEN is matched. When GET is invoked, the sequence of digits corresponding base ten to the current value of this counter is appended to the output. The storage provided for the counter however is dynamic in the following sense. Every time the translator encounters a tranduction element which is a genus then the current value of the counter is saved and this value is restored if and when the translator returns to this incarnation of this transducer. Thus for instance the option:

```
::= GEN . ':FUN:' . GET . ',' . BINDING . ':NUF:' . GET . ','
```

will match any input directly. Regardless of the effect of BINDING upon the symbol generator, the output from both occurrences of GET will be the same, the value generated by GEN. This allows the assembler to correlate the occurrences of FUN and NUF in the intermediate language program.

NAME and GETN

As we mentioned above NAME as a prefix will match any sequence of letters at the head of the input. However, the action associated with NAME is much more complex than that. To begin with NAME may only be used as a prefix if a genus called NAME has been declared and defined (given a possibly empty list of options). A successful match against the prefix NAME then effectively activates an independant symbol generator and then constructs an option whose prefix is the sequence of letters matched and whose tranducer consists of a single transduction element which is the string of digits generated. This option is then linked into the definition of the genus NAME as a new first option, all the other options taking a place one later in the sequence. The effect therefore is to alter the definition of NAME to include the possibility of correlating the occurrences of identifiers in the source language. For example, the option

```
        ::= NAME
```

would match the input PETER+.... and cause the definition of NAME to become, for example

```
        NAME ::= 'PETER' . '17'
             ::= .....previous first option (if any).
```

Associated with the symbol generator used in conjunction with NAME we have the activity GETN which simply moves the current value, as a sequence of digits base ten, to the output. Thus if the above option had been

```
        ::= NAME . ':DEC:' . GETN . ','
```

the output :DEC:17, would have been produced in correlation with the 17 stored in the definition of NAME.

<u>SAVE</u>

If SAVE occurs as a prefix to an option then the current level of definitions attached to the genus NAME is noted. At the end of translation with respect to this option, this level of definition is restored, any options added by translating with respect to genera which were tranduction elements of this option, are lost. This gives the usually accepted definition of block structure to the source language. For example the option

```
        ::=  'USE' . BLOCK . ALPHA
```

in the context of the definition

```
  BLOCK  ::=  SAVE . DECLARATIONS
```

has the property that any definitions introduced (by adding options to NAME) when translating with respect to BLOCK are removed before translation with respect to ALPHA

EMPTY

This action prefix is simply an alternative to the empty string.

The translator description for ALEPH

ALEPH, as described in this dissertation is implemented by POSSUM and the actual translator description follows. Each three letter mnemonic is preceded by a colon, which is used by the assembler as a locator. Parameterless mnemonics are delimited by a comma , while others are separated from their (integer) parameter by a colon and the parameter is delimited by a comma. The mnemonics correspond directly to the macro names used to specify translations in Chapter 2. It therefore is possible for the reader to follow the action of the translator by referring to the definition of the translation of each primary as given in Chapter 2.

```
£COMMENT   ***   TRANSLATOR DESCRIPTION FOR ALEPH* IN POSSUMO* NOTATION    ***
£LIS SYNTAX
        BEGIN

        GENERA PROG,EXPR,DISJUNCTION,DISJOIN,CONJUNCTION,CONJOIN,
               NEGATION,RELATION,RELATIVE,SUM,ADDEND,TERM,MULTIPLIER,
               FACTOR,PRIMARY,BLOCK,LOOP,CONDITIONAL,ENDCOND,
               VARIABLE,ASSIGNATION,FUNCTION,BINDING,INITIAL,
               APRIMARY,ARGUMENT,PARAMETERS,VALUE,SUBSCRIPT,SUBSCRASS,

               COMPOUND    ='INVALID-EXPRESSION-DELIMETER',
               DECLARATION='ADJACENT-DELIMETERS-IN-DECLARATION-LIST',
               DO          ='DO-MISSED-IN-ITERATING-PRIMARY',
               ANTECEDENT ='THEN-MISSED-IN-CONDITIONAL-PRIMARY',
               CONSEQUENT ='ELSE-MISSED-IN-CONDITIONAL-PRIMARY',
               CLOSE       ='CLOSING-PARENTHESIS-MISSED',
               NAME        ='UNDECLARED-NAME/INVALID-PRIMARY',
               FORMALS     ='ADJACENT-DELIMITERS IN-FORMAL-LIST',
               MOREFORMS  ='INVALID-DELIMITER-IN-FORMAL-LIST' ,
               MOREPARAMS ='INVALID-DELIMITER-IN-ACTUAL-LIST' ,
               EQUAL       ='EQUAL-SIGN-MISSED-IN-DECLARATION-LIST' ;

        PROG           ::= '' . ':ENT,' . PRIMARY .  ':EXT,#' ;

        EXPR           ::= '' . DISJUNCTION ;

        DISJUNCTION ::= '' .CONJUNCTION . DISJOIN ;
        DISJOIN        ::= 'OR' . CONJUNCTION . ':QOR,' . DISJOIN
                       ::= EMPTY ;
        CONJUNCTION ::= '' . NEGATION . CONJOIN ;
        CONJOIN        ::= 'AND' . NEGATION . ':AND,' . CONJOIN
                       ::= EMPTY ;
        NEGATION       ::= 'NOT' . NEGATION . ':NOT,'
                       ::= '' . RELATION ;
        RELATION       ::= ''  .  SUM . RELATIVE ;
        RELATIVE       ::= '=' . SUM . ':EQL,'
```

```
              ::= '<=' . SUM . ':LEQ,'
              ::= '>=' . SUM . ':GEQ,'
              ::= '¬=' . SUM . ':NEQ,'
              ::= '<'  . SUM . ':LSS,'
              ::= '>'  . SUM . ':GRT,'
              ::= EMPTY ;
SUM           ::= '+' . TERM . ADDEND
              ::= '-' . TERM . ':NEG,' . ADDEND
              ::= '' . TERM . ADDEND ;
ADDEND        ::= '+' . TERM . ':ADD,' . ADDEND
              ::= '-' . TERM . ':SUB,' . ADDEND
              ::= EMPTY ;
TERM          ::= '' . FACTOR . MULTIPLIER ;
MULTIPLIER    ::= '*' . FACTOR . ':MUL,' . MULTIPLIER
              ::= '/' . FACTOR . ':DIV,' . MULTIPLIER
              ::= 'MOD'.FACTOR . ':MOD,' . MULTIPLIER
              ::= EMPTY ;
FACTOR        ::= '' . PRIMARY  ;

PRIMARY       ::= 'BEGIN' . EXPR . COMPOUND
              ::= 'WHILE' . LOOP
              ::= 'LET'   . BLOCK
              ::= 'IF'    . CONDITIONAL
              ::= NUMBER  . ':NUM:+' . GETNUM . ','
              ::= 'INPUT' . ':INP,'
              ::= 'DIGITS' . PRIMARY . ':DIG,'
              ::= 'FIELDS' . PRIMARY . ':FLD,'
              ::= 'LAMBDA'. FUNCTION
              ::= 'OUTPUT'. PRIMARY . ':OUT,'
              ::= '' . APRIMARY . ARGUMENT ;

 APRIMARY     ::= '('. EXPR . CLOSE
              ::= ''. VARIABLE . ASSIGNATION  ;

 COMPOUND     ::= ';' . ':OFF,' . EXPR . COMPOUND
              ::= 'END' ;

BLOCK         ::= SAVE . DECLARATION ;
DECLARATION ::= NAME . EQUAL . VALUE ;
VALUE         ::= 'ROW' . EXPR . INITIAL .  GETN . ',' . EXPR . ':WOR:' . GETN . ','
              ::=  ''   . EXPR . ':LET:' . GETN . ',' . EXPR . ':TEL:' . GETN . ','  ;
INITIAL       ::= 'EACH' . EXPR . ':ROW:'
              ::= '' . ':ARR:' ;

LOOP          ::= GEN . ':WHL:' . GET . ',' . EXPR . DO . EXPR . ':GOW:' . GET . ',' ;
DO            ::= 'DO' . ':DOW:' . GET . ',' ;

CONDITIONAL ::= GEN . EXPR . ANTECEDENT . CONSEQUENT . ENDCOND ;
ANTECEDENT    ::= 'THEN' . ':THN:' . GET . ',' . EXPR ;
CONSEQUENT    ::= 'ELSE' . ':ELS:' . GET . ',' . EXPR ;
ENDCOND       ::= '' . ':NDC:' . GET . ',' ;

CLOSE         ::= ')' ;

VARIABLE      ::= '' . ':GET:' . NAME . ',' ;

ASSIGNATION ::= ':=' . EXPR . ':ASS,'
              ::= EMPTY . ':VAL,' ;

NAME  ;

FUNCTION      ::= GEN . ':FUN:' . GET . ',' . BINDING . ':NUF:' . GET . ',' ;
BINDING       ::= SAVE . FORMALS ;
FORMALS       ::= NAME . ':FOR:' . GET . ',' . MOREFORMS . ':ROF:' . GET . ','
              ::= '.' . ':CHK,' .EXPR . ':DCT,' ;
MOREFORMS     ::= ',' . FORMALS
              ::= '.' . ':CHK,' . EXPR . ':DCT,' ;

ARGUMENT      ::= '(' . ':MRK,' . PARAMETERS . ':UNM,' . ARGUMENT
              ::= '' . SUBSCRIPT ;
PARAMETERS    ::= ')' . ':CLL,'
              ::='' . EXPR . MOREPARAMS ;
```

```
         MOREPARAMS  ::= ')' . ':CLL,'
                     ::= ',' . EXPR . MOREPARAMS ;

         EQUAL       ::= '=' ;

         SUBSCRIPT   ::= '@' . PRIMARY . ':INC,' . SUBSCRASS . ARGUMENT
                     ::= EMPTY ;

         SUBSCRASS   ::= ':=' . EXPR . ':IAS,'
                     ::= EMPTY . ':IVA,' ;

         END
END OF FILE
```

The sequence of mnemonics which represents the translation of an ALEPH program is processed by the author's assembler. The assembler is table driven and the table is compiled into a form acceptable to the MTS loader by the 360 G ASSEMBLER using a macro DEF illustrated below. For example, the mnemonic ASS, whose definition in Chapter 2 is

```
ass : (W=cF, V@W=T, F=pF)
```

is encoded straightforwardly as

```
DEF  ASS
L    W, O(F)
ST   T, O(W,V)
SR   F,4
```

where W,F,T and V are registers with appropriate definitions, and where register 4 contains the number of bytes per integer (= 4). In the case of ELS however:

```
els(i) : (→B@i, A@i: T↓F)
```

the encoding is less straightforward

```
DEF  ELS, A=6, P1=2
L   W, 4(0,J)
BR  W
L   T, 0(F)
SR  F, 4
```

Here J is a register referring to the indirection table and the displacement of 4 selects the B field. The keyword parameter A=6 specifies that the A field of the indirection table is to be initialised to the address of the instruction which begins on the 6th byte of this macro. The keyword parameter P1=2 specifies that the displacement beginning at the second byte of this macro is to be incremented by an amount calculated from i, to select the correct entry in the indirection table.

The following table shows most of the instructions used in the hypothetical machine of Chapter 3, along with their effect upon the environment $\xi = [t,f,m,v]$. In addition, the author's interpretation of these instructions for the IBM 360/67 and a suggested scheme for the IBM 1130 are tabulated. In the case of the 360, the displacement i* denotes a value inserted by the assembler, calculated from the parameter i. Similarly the 1130 implementation requires the displacement inserted into the instruction. The 1130 code could not be generated in a single pass because of insuffficent index registers. The use of i^* is not intended to imply that the displacements are the same function of i, only to signify that a calculation is necessary.

		IBM 360/67	IBM 1130
F = nF	[t,(∩,f),m,v]	AR F,4	MDX F 1
F = pF	[t,f^-,m,v]	SR F,4	MDX F -1
T = cF	[f^+,f,m,v]	L T,0(F)	LD F 0
cF = T	[t,(t,f^-),m,v]	ST T,0(F)	STO F 0
T↑F	[t,(t,f),m,v]	(F = nF, cF = T)	
T↓F	[f^+,f^-,m,v]	(T = cF, F =pF)	
T = V@i	[vi,f,m,v]	L T,i*(V)	LD V + i*
V@i = T	[t,f,m,Uvit]	ST T,i*(V)	STO V + i*
T = V@W		L T,0(W,V)	LD W V
T = V@T	[vt,f,m,v]	L T,0(T)	
→B@i		L W,i*(J) BR W	BSC I J + i*
(T = 0) ⇒ → A@i		LTR T,T L W,i*(J) BCR 8,W	BSC I J + i*, +-
F = n^TF	[t,(∩,... (∩,f)),m,v]	SLL T,2 AR F,T	
T = M		LR T,M	

When a register is used to index V then the register contains a calculated displacement rather than the index, where these differ.

The intention here has been simply to illustrate the proximity of the hypothetical device of Chapter 2 to actual current day machine operated in a certain way. Thus this Appendix may seem annoyingly incomplete. To attempt to describe the author's implementation fully would require much more space than could be justified in a dissertation of this nature.

APPENDIX 3

This Appendix presents fifteen examples of ALEPH programs in the form of the output obtained when these programs are run under the author's 360 (MTS) implementation. The form of the interface with MTS should be obvious when the Appendix is inspected. The programs are held in a file called TESTSOURCE and are referred to as TESTSOURCE (i) for i = 1,... ...,15. A file called DESCRIBE contains descriptions of each program, DESCRIBE (i) describing TESTSOURCE (i) and the file DATA contains sample data, DATA (i) being used for TESTSOURCE (i). Each example is presented by listing, first the source program then its description. The compiler is stored in the file ALEPH* and it obtains its source from SCARDS and dumps the load module produced in the file attached to device 2.

```
$RUN    ALEPH*    SCARDS = TESTSOURCE (i)    2 = -LOAD
```

The file -LOAD collects the machine language translation of TESTSOURCE (i). The command

```
$RUN    ALGO*+-LOAD    SCARDS = DATA (i)
```

runs the program -LOAD using the input and output routines stored in ALGO* and the data stored as DATA (i). The data is printed when it is read by the primary <u>input</u>, in the form

```
DATUM    x,
```

and then the results are printed in a format defined as follows. The primaries p_1 and p_2, defined by

$$p_1 \equiv \underline{\text{digits}}\ p$$

$$p_2 \equiv \underline{\text{fields}}\ p$$

for some primary p control the shape of an output record. The first primary, when evaluated, determines that subsequent output values shall be assumed to contain a maximum of p digits and hence p columns will be used to print each value. Similarly, the number of values printed on each record is defined by the primary p_2 to be the value of p. Hence

$\underline{\text{digits}}$ 3; $\underline{\text{fields}}$ n;

will cause output records to contain n values, each occupying 3 columns of the record. If <u>digits</u> and <u>fields</u> are not used then output defaults to 11 digit integers printed 1 per record. The remainder of the Appendix should be self-explanatory.

```
$COMMENT      *** EXAMPLE 1 ***
$LIST TESTSOURCE(1)
   100      OUTPUT LET N=INPUT
   101             LET S=1
   102             LET I=0
   103             WHILE (I:=I+1)<=N DO S:=S*I
END OF FILE
$LIST DESCRIBE(1)
   100      THIS SIMPLE PROGRAM COMPUTES THE FACTORIAL FUNCTION BY USING THE
   101      ITERATING PRIMARY.THE VARIABLE I IS SUCCESSIVELY INCREMENTED BY 1 AND
   102      MULTIPLIED INTO THE PRODUCT S. THE VALUE OF THE ITERATING PRIMARY
   103      I.E. THE PRODUCT S, IS OUTPUT AS THE RESULT.
END OF FILE
$EMPTY -LOAD
DONE.
$RUN ALEPH* SCARDS=TESTSOURCE(1) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
$RUN ALGO*+-LOAD SCARDS=DATA(1)
EXECUTION BEGINS
DATUM  6,
     720
EXECUTION TERMINATED
```

```
$COMMENT        *** EXAMPLE 2 ***
$LIST TESTSOURCE(2)
   200      OUTPUT LET N=INPUT
   201             LET I=0
   202             LET C=0
   203               WHILE (I:=I+1)<N DO
   204               C:=C+ IF (LET X=N
   205                         LET Y=I
   206                         WHILE X MOD Y ¬=0 DO
   207                         Y:= X MOD X:=Y        )=1 THEN 1 ELSE 0
END OF FILE
$LIST DESCRIBE(2)
   200      AS DESCRIBED IN CHAPTER 2 THIS PROGRAM COMPUTES EULERS TOTIENT. THE
   201      INNERMOST ITERATING PRIMARY DETERMINES THE HIGHEST COMMON FACTOR OF X
   202      AND Y. THE USE OF THE BLOCK PRIMARY LOCALISES THE SIDE EFFECT UPON
   203      X AND Y OF THIS DETERMINATION. THE CONDITIONAL PRIMARY THUS HAS 1
   204      AS ITS VALUE IF N AND I HAVE A COMMON FACTOR AND 0 IF NOT. THE VARIABLE
   205      C THUS ACCUMULATES THE NUMBER OF INTEGERS LESS THAN N WHICH HAVE A
   206      FACTOR IN COMMON WITH N. THIS IS EULER'S TOTIENT.
END OF FILE
$EMPTY -LOAD
DONE.
$RUN ALEPH* SCARDS=TESTSOURCE(2) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
$RUN ALGO*+-LOAD SCARDS=DATA(2)
EXECUTION BEGINS
DATUM    84,
         23
EXECUTION TERMINATED
```

```
£COMMENT       *** EXAMPLE 3 ***
£LIST TESTSOURCE(3)
   300       LET N=INPUT+1
   301       LET R=INPUT
   302       LET S=1  LET T=1
   303       BEGIN
   304          LET I=0
   305          WHILE I<R DO
   306          BEGIN S:=S*(N-I:=I+1);
   307                T:=T*I
   308          END;
   309
   310          OUTPUT (S/T)
   311       END
END OF FILE
£LIST DESCRIBE(3)
   300       AGAIN THIS EXAMPLE IS DESCRIBED IN CHAPTER2. THE COMPOUND PRIMARY
   301       SIMPLY DETERMINES THE NUMBER OF WAYS R THINGS CAN BE PERMED FROM
   302       N-1 BY ACCUMULATING THE PRODUCTS OF 1,...,R IN T AND N-1,...,N-R
   303       IN S. THE QUOTIENT S/T IS THE DESIRED COEFFICIENT.
END OF FILE
£EMPTY -LOAD
DONE.
£RUN ALEPH* SCARDS=TESTSOURCE(3) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
£RUN ALGO*+-LOAD SCARDS=DATA(3)
EXECUTION BEGINS
DATUM    7,
DATUM    5,
         21
EXECUTION TERMINATED
```

```
$COMMENT      *** EXAMPLE 4 ***
$LIST TESTSOURCE(4)
   400      LET HCF=LAMBDA X,Y . WHILE X MOD Y ¬=0 DO Y:=X MOD X:=Y
   401      LET PHI=LAMBDA N .
   402              LET I=0
   403              LET C=0
   404              WHILE (I:=I+1)<N DO
   405              IF HCF(N,I)=1 THEN C:=C+1 ELSE C
   406
   407      OUTPUT PHI(INPUT)
END OF FILE
$LIST DESCRIBE(4)
   400      EXAMPLE 2 IS REPEATED HERE USING FUNCTIONS FOR HIGHEST COMMON FACTOR
   401      (HCF) AND EULER'S TOTIENT (PHI). IT DIFFERS VERY LITTLE FROM EXAMPLE 2.
END OF FILE
$EMPTY -LOAD
DONE.
$RUN ALEPH* SCARDS=TESTSOURCE(4) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
$RUN ALGO*+-LOAD SCARDS=DATA(4)
EXECUTION BEGINS
DATUM  84,
        23
EXECUTION TERMINATED
```

```
£COMMENT       *** EXAMPLE 5 ***
£LIST TESTSOURCE(5)
   500     LET SQRT =LAMBDA X . LET I=0
   501          WHILE (I:=I+1)*I<=X DO I
   502
   503     LET CIRCLE=LAMBDA N,RSQ . IF N=0 THEN 1 ELSE
   504                    CIRCLE(N-1,RSQ)+2*
   505                    LET C=0
   506                    LET I=SQRT(RSQ)+1
   507                    WHILE (I:=I-1)>0 DO
   508                          C:=C+CIRCLE(N-1,RSQ-I*I)
   509
   510     OUTPUT CIRCLE(INPUT,LET K=INPUT K*K)
END OF FILE
£LIST DESCRIBE(5)
   500     THIS IS A MORE INTERESTING PROBLEM AND OUR FIRST EXAMPLE OF ALEPH
   501     USED RECURSIVELY. A SIMPLE SQUARE ROOT FUNCTION IS USED WHICH
   502     RETURNS THE INTEGER PART OF THE SQUARE ROOT OF ITS PARAMETER, AS
   503     LONG AS THE PARAMETER IS GREATER THAN ZERO, AND RETURNS ZERO OTHERWISE.
   504     IT DOES THIS SIMPLY BY INCREMENTING I UP BY ONES FROM ONE AND TRYING
   505     EACH INCREMENTED VALUE BY SQUARING IT. AS SOON AS I SQUARED EXCEEDS X, THE
   506     PARAMETER, THE PREVIOUS VALUE OF I IS ACCEPTED. FOR THE USE TO WHICH IT
   507     IS TO BE PUT HERE, PRACTICAL CONSIDERATIONS DICTATE THAT IT WILL NOT BE
   508     NECESSARY TO COUNT BEYOND 10 SAY, THUS THE METHOD IS NOT NECESSARILY
   509     INEFFICIENT.
   510     THE FUNCTION CIRCLE(N,RSQ) RETURNS THE NUMBER OF LATTICE POINTS ON
   511     AN N-DIMENSIONAL UNIT GRID CONTAINED IN OR ON THE SURFACE OF AN
   512     N-DIMENSIONAL HYPERSPHERE, CENTRED AT THE ORIGIN, WHOSE RADIUS SQUARED
   513     IS GIVEN BY THE PARAMETER RSQ. A ZERO DIMENSIONAL SPHERE OF ANY RADIUS
   514     CONVENTIONALLY CONTAINS A SINGLE POINT . A SPHERE OF NON-ZERO DIMENSION
   515     N CONTAINS THOSE POINTS WHICH ARE CONTAINED IN THE (N-1) DIMENSIONAL
   516     CROSS SECTIONS ORTHOGONAL TO AN AXIS OF THE COORDINATE SYSTEM, CUT BY
   517     THE PLANES OF LATTICE POINTS. THERE IS A SYMMETRY ABOUT THE ORIGIN
   518     WHICH MEANS WE NEED ONLY DOUBLE THE POINTS IN A HYPERHEMISPHERE
   519     THE CENTRAL CROSS-SECTION HOWEVER OCCURS ONLY ONCE. THUS
   520     THE ITERATING PRIMARY SUMS THE CROSS-SECTIONS CIRCLE(N-1,RSQ-I*I)
   521      WHERE I RUNS FROM 1 TO SQRT(RSQ) . IT DOES THIS BACKWARDS TO SAVE
   522      STORING THE VALUE OF SQRT(RSQ). THE VALUE OF THIS SUM, DOUBLED AND
   523      INCREMENTED BY CIRCLE(N-1,RSQ), THE CENTRAL CROSS-SECTION IS THE
   524      VALUE OF THE FUNCTION
END OF FILE
$EMPTY -LOAD
DONE.
$RUN ALEPH* SCARDS=TESTSOURCE(5) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
$RUN ALGO*+-LOAD SCARDS=DATA(5)
EXECUTION BEGINS
DATUM    4,
DATUM    10,
   49689
EXECUTION TERMINATED
```

```
£COMMENT    *** EXAMPLE 6 ***
£LIST TESTSOURCE(6)
            LET N=INPUT
            LET Q=(LET S=1
                   LET J=0 WHILE (J:=J+1)<N DO S:=S*J )
            LET A=ROW N
            LET M=INPUT
            BEGIN
                   LET J=N+1 WHILE (J:=J-1)>1 DO
                   BEGIN A@(J):=M/Q; M:=M MOD Q;
                         Q:=Q/(J-1)
                   END ;

                   A@1:=0 ;

                   LET I=1 WHILE (I:=I+1)<=N DO
                   LET J=0 WHILE (J:=J+1)<=I-1 DO
                   IF A@J<A@I THEN 0 ELSE A@(J):=A@(J)+1 ;

                   DIGITS 3; FIELDS N;
                   LET I=N+1 WHILE (I:=I-1)>=1 DO OUTPUT (A@I+1)
            END
END OF FILE
£LIST DESCRIBE(6)
            THIS IS LEHMERS LEXICOGRAPHIC METHOD OF DETERMINING THE M-TH
            PERMUTATION IN LEXICOGRAPHIC ORDER OF THE INTEGERS 1,... ...,N.
            FIRST N IS READ IN AND THEN Q IS SET TO THE VALUE OF THE FACTORIAL
            OF N-1. A VECTOR OF N ELEMENTS WITH UNDEFINED VALUES IS DECLARED FOR
            WORK SPACE. THE VALUE M IS THEN OBTAINED FROM THE INPUT STREAM.
            THE FACTORIAL EXPANSION OF M IS THEN CALCULATED AS DESCRIBED IN SECTION
            2.23. THESE DIGITS ARE COMBINED BY SCANNING THE FIRST I-1 ELEMENTS
            FOR I RUNNING BETWEEN 2 AND N, AND INCREASING EACH ELEMENT BY 1 IF
            IT IS LESS THAN THE I'TH.THUS IF N=4 AND M=17
                            17=2.3 +2.2 +1.1
                        AND A=(0,1,2,2) WHICH WHEN SCANNED
                     YIELDS A=(0,1,3,2) WHICH IS THE 17'TH PERMUTATION OF 0,...,3
```

```
    612     IN REVERSED ORDER.
    613     THE USE OF DIGITS AND FIELDS DEMONSTRATES LAYOUT CONTROL AND THE FINAL
    614     ITERATING PRIMARY DUMPS THE REQUIRED PERMUTATION AS A ROW OF N, 3-DIGIT
    615     NUMBERS.
END OF FILE
£EMPTY -LOAD
DONE.
£RUN ALEPH* SCARDS=TESTSOURCE(6) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
£RUN ALGO*+-LOAD SCARDS=DATA(6)
EXECUTION BEGINS
DATUM   10,
DATUM   1000000,
  3  8  9  4 10  2  6  7  1  5
EXECUTION TERMINATED
```

```
£COMMENT       *** EXAMPLE 7 ***
£LIST TESTSOURCE(7)
   700      LET TRUE =-1
   701      LET FALSE= 0
   702      LET N=INPUT
   703      LET COL=ROW N EACH FALSE
   704      LET POS=ROW N
   705      LET PR =ROW 2*N-1 EACH FALSE
   706      LET SC =ROW 2*N-1 EACH FALSE
   707      LET SOLUTION=LAMBDA . LET J=0
   708                  WHILE (J:=J+1)<=N DO OUTPUT POS@J
   709
   710      LET QUEEN = LAMBDA I . IF I>N THEN SOLUTION() ELSE
   711                 LET J=0 WHILE(J:=J+1)<=N DO
   712                 IF COL@(J) OR PR@(I+J-1) OR SC@(N+I-J) THEN 0 ELSE
   714                 BEGIN
   715                       POS@(I):=J;
   716                       COL@(J):=PR@(I+J-1):=SC@(N+I-J):=TRUE ;
   717                       QUEEN(I+1);
   718                       COL@(J):=PR@(I+J-1):=SC@(N+I-J):=FALSE
   719                 END
   720
   721      BEGIN DIGITS 3; FIELDS N; QUEEN(1) END
END OF FILE
£LIST DESCRIBE(7)
   700      THIS IS A RECURSIVE SOLUTION TO THE NXN QUEENS PROBLEM. WHEN SUPPLIED
   701      WITH THE INTEGER N IT PRINTS A SEQUENCE OF VECTORS,EACH A PERMUTATION
   702      OF 1 TO N EACH OF WHICH IS INTERPRETABLE AS THE COLUMN POSITIONS FOR
   703      THE QUEEN IN THE CORRESPONDING ROW OF A CHESSBOARD. QUEENS THUS
   704      ARRAYED CANNOT TAKE EACH OTHER . THREE VECTORS ARE USED TO MARK
   705      OCCUPANCY OF THE COLUMNS (COL), THE PRINCIPAL DIAGONALS (PR) AND THE
   706      SECONDARY DIAGONALS (SC). THE FUNCTION "QUEEN" TAKES A PARAMETER I
   707      REPRESENTING THE NEXT ROW ON WHICH TO PLACE A QUEEN. THUS IF I>N IT
   708      CALLS "SOLUTION" TO PRINT THE SOLUTION VECTOR (POS). OTHERWISE, FOR
   709      EACH POSSIBLE VALUE OF THE COLUMN COUNTER (J) THE SQUARE (I,J) IS
   710      CHECKED TO SEE IF IT IS COVERED (CORRESPONDING COLUMN OR DIAGONALS
   711      OCCUPIED). IN CASE IT IS NOT COVERED THEN THIS SQUARE IS RECORDED
   712      AS A CANDIDATE BY UPDATING PCS,CCL,PR AND SC. THE ROUTINE IS ENTERED
   713      RECURSIVELY WITH THE NEXT HIGHER VALUE OF THE ROW COUNTER TO
   714      DETERMINE ALL SOLUTIONS WITH THIS PARTIAL COMPONENT (REPRESENTED
   715      BY PCS@1,...   ...,PCS@I).
END OF FILE
£EMPTY -LOAD
DONE.
£RUN ALEPH* SCARDS=TESTSOURCE(7) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
£RUN ALGO*+-LOAD SCARDS=DATA(7)
EXECUTION BEGINS
DATUM  6,
 2  4  6  1  3  5
 3  6  2  5  1  4
 4  1  5  2  6  3
 5  3  1  6  4  2
EXECUTION TERMINATED
```

```
£COMMENT      *** EXAMPLE 8 ***
£LIST TESTSOURCE(8)
   800      LET Y=INPUT
   801      LET X=3
   802      LET L=0
   803      LET C=0
   804      LET P=LAMBDA.
   805            IF X>Y THEN C ELSE
   806            LET N=ROW 1
   807            BEGIN
   808                  N@0:=X;  N@1:=L;
   809                  X:=X+2;
   810                  LET T=L WHILE T¬=C DO
   811                  IF X MOD T@0 =0 THEN
   812                      BEGIN X:=X+2; T:=L END ELSE T:=T@1 ;
   814                  C:=C+1 ;
   816                  L:=N ;
   817                  P()
   818            END
   819      OUTPUT P()
END OF FILE
£LIST DESCRIBE(8)
   800      THIS IS AN INTERESTING USE OF RECURSIVELY NESTED BLOCKS TO GENERATE
   801      A LINKED LIST OF PRIME NUMBERS. THE INTENTION IS TO DETERMINE THE
   802      NUMBER OF ODD PRIMES LESS THAN THE VALUE INPUT, WHICH IS STORED IN Y.
   803      WHEN P IS ENTERED X IS THE LARGEST PRIME SO FAR GENERATED AND C IS
   804      THE NUMBER OF PRIMES GENERATED. AT THIS STAGE ALSO L IS A LINKED
   805      LIST (IN A SENSE TO BE DESCRIBED) OF THE PRIMES LESS THAN X. IF
   806      X>Y THEN C IS THE RESULT OF THE COMPUTATION, OTHERWISE A TWO ELEMENT
   807      VECTOR N (MAX. INDEX 1) IS CREATED.THE TWO ELEMENTS OF N ARE SET
   808      TO CONTAIN X AND L RESPECTIVELY. THUS A NEW ELEMENT IS ATTACHED TO THE
   809      FRONT OF THE LIST. X+2 IS POSTULATED AS THE NEXT PRIME AND T IS USED
   810      TO RUN DOWN THE LIST AND CHECK THIS HYPOTHESIS. IF A DIVISOR IS FOUND
   811      THE GUESS IS INCREASED BY 2 AND THE SEARCH REPEATED, OTHERWISE X IS
   812      A NEW LARGEST PRIME. C IS INCREASED BY 1 AND L IS SET TO ITS NEW VALUE
   813      N. P IS THEN REENTERED TO SEARCH FOR THE NEXT PRIME.
```

```
END OF FILE
£EMPTY -LOAD
DONE.
£RUN ALEPH* SCARDS=TESTSOURCE(8) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
£RUN ALGO*+-LOAD SCARDS=DATA(8)
EXECUTION BEGINS
DATUM   500,
        94
EXECUTION TERMINATED
```

```
$COMMENT     *** EXAMPLE 9 ***
$LIST TESTSOURCE(9)
   900      LET M=INPUT  LET N=INPUT
   901      LET TRUE=-1  LET FALSE=0
   902      LET S=ROW N  LET T=ROW N  LET B=ROW N
   903      LET X=0
   904      LET P=LAMBDA  I . IF  I>N THEN
   905            IF T@N<=X THEN 0 ELSE
   906            BEGIN  OUTPUT X:=T@N;
   907                 LET I=0 WHILE (I:=I+1)<=N DO  OUTPUT B@(I):=S@I
   908            END  ELSE
   909         LET J=S@(I-1)
   910         LET U=T@(I-1)+2
   911         WHILE (U:=U-1)>J DO
   912         BEGIN S@(I):=U ;
   913         T@(I):=(LET V=ROW M*S@(I)+1 EACH FALSE
   914              LET NEST =LAMBDA K,SUM,SLS .
   915                        IF K>I THEN V@(SUM):=TRUE ELSE
   916                        LET L=-1 WHILE ( L:=L+1)<=M-SLS DO
   917                        NEST(K+1,SUM+L*S@K,SLS+L)
   918              BEGIN NEST(1,0,0);
   919              (LET K=T@(I-1) WHILE V@K DO K:=K+1)-1
   920              END) ;
   921         P(I+1)
   922         END
   923      BEGIN S@(1):=1;T@(1):=M;
   924      DIGITS 3 ; FIELDS(N+1) ;
   925      P(2)
   926      END
END OF FILE
$LIST DESCRIBE(9)
   900      THE PROBLEM HERE IS TO DETERMINE THE DENOMINATIONS OF N STAMPS TO BE
   901      ISSUED BY THE POST-OFFICE IF THE MAXIMUM OF M IS ALLOWED PER ENVELOPE
   902      SO THAT THE LONGEST UNBROKEN SEQUENCE OF POSTAGES CAN BE CONSTRUCTED.
   903      OF PRINCIPAL INTEREST IS THE RECURSIVE DEFINTION OF "P" AND "NEST".
   904      B@1,...   ...,B@N IS THE BEST SET SO FAR FOUND AND X THE MAXIMUM
   905     OBTAINABLE POSTAGE, SO THAT EVERY POSTAGE LESS THAN X MAY BE OBTAINED.
   906     WHEN P IS ENTERED S@1,...   ...,S@(I-1) IS A PARTIALLY CONSTRUCTED
   907     ISSUE AND T@1,...   ...,T@(I-1) ARE PARTIAL POSTAGES IN THE SENSE THAT
   908     T@J IS THE MAXIMUM OBTAINABLE POSTAGE USING ONLY S@1,...   ...,S@J FOR
   909     EACH J,1<=J<=I-1. ON ENTRY TO P WHEN I>N,S AND T ARE COMPLETED AND
   910     S IS SAVED IN B IF IT IS AN IMPROVEMENT. THIS OPPORTUNITY IS ALSO TAKEN
   911     TO PRINT S. IF I<=N ON ENTRY TO P HOWEVER THEN ANY OF THE VALUES U,
   912     WHERE S@(I-1)+1<=U<=T@(I-1)+1 ARE CANDIDATES FOR S@I. THUS EACH
   913     IS TRIED BY CALCULATING T@I AND RE-ENTERING P WITH AN INCREMENTED
   914     VALUE OF I FOR EACH POSSIBLE VALUE U. THE CALCULATION OF T@I IS
   915     HOWEVER NON-TRIVIAL . A VECTOR V OF APPROPRIATE SIZE IS ESTABLISHED
   916     SO THAT V@SUM CAN BE SET TRUE IF SUM IS AN OBTAINABLE POSTAGE USING THE
   917     PARTIAL ISSUE S@1,...   ...,S@I. THE VALUES OF V ARE SET UP BY THE
   918     FUNCTION NEST WHICH WHEN ENTERED IS ABOUT TO STICK S@K ON A LETTER
   919     WHICH ALREADY HAS A PARTIAL POSTAGE OF SUM CONTRIBUTED BY SLS (<=M)
   920     STAMPS  . THUS IF K>I THEN SUM IS RECORDED AS AN OBTAINABLE POSTAGE.
   921     ALTERNATIVELY, IF K<=I THEN S@K CAN BE USED L TIMES WHERE 0<=L<=M-SLS
   922     THUS FOR EACH OF THESE POSSIBILITIES NEST IS ENTERED RECURSIVELY
   923     WITH K INCREASED BY 1, SUM INCREASED BY L*S@K AND SLS INCREASED BY
   924     L (SLS MEANS SUM OF L'S). NEST (1,0,0) THEREFORE HAS THE DESCRIBED SIDE
   925     EFFECT UPON V WHICH IS THEN SCANNED FOR THE FIRST FALSE ENTRY. THIS
   926     SCAN IS REDUCED BY KNOWING THAT THE FIRST T@(I-1) ELEMENTS ARE TRUE
   927     AND IS STOPPED IN THE EXTREME CASE BY PROVIDING AN EXTRA ELEMENT FOR
   928     V WHICH REMAINS FALSE. S@1 AND T@1 ARE INITIALLY 1 AND M RESPECTIVELY
   929     AND THE COMPUTATION IS BEGUN BY CALLING P(2). WHENEVER AN IMPROVED
   930     VALUE IS OBTAINED IT IS PRINTED AND SO THE LAST LINE OF OUTPUT IS THE
   931     DESIRED SOLUTION.THIS ALGORITHM WAS DEVISED WITH THE ASSISTANCE OF
   932     MR T. ANDERSON OF THE UNIVERSITY OF NEWCASTLE UPON TYNE.
END OF FILE
$EMPTY -LOAD
DONE.
$RUN ALEPH* SCARDS=TESTSOURCE(9) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
$RUN ALGO*+-LOAD SCARDS=DATA(9)
EXECUTION BEGINS
```

```
DATUM    4,
DATUM    4,
16   1  5   9 13
39   1  5   9 12
40   1  5   9 10
41   1  4  11 13
44   1  3  11 18
EXECUTION TERMINATED
```

```
$COMMENT     *** EXAMPLE 10 ***
$LIST TESTSOURCE(10)
  1000      LET N=INPUT
  1001      BEGIN
  1002
  1003
  1004      (LAMBDA F.OUTPUT F(N))
  1005      (LAMBDA X.
  1006        LET G=LAMBDA X,H.
  1007               LET F=LAMBDA Z.H(Z,H)
  1008               IF X=0 THEN 1 ELSE X*F(X-1)
  1009        G(X,G) );
  1010
  1016
  1017      (LAMBDA F.OUTPUT F(N))
  1018      (LAMBDA X.
  1019        LET G=LAMBDA X,H.(LAMBDA F. IF X=0 THEN 1 ELSE X*F(X-1))(LAMBDA Z.H(Z,H))
  1020        G(X,G) );
  1021
  1027
  1028      (LAMBDA F.OUTPUT F(N))
  1029      (LAMBDA X.
  1030        (LAMBDA G.G(X,G))
  1031        (LAMBDA X,H.(LAMBDA F. IF X=0 THEN 1 ELSE X*F(X-1))(LAMBDA Z.H(Z,H))))
  1032
  1033      END
  1034
END OF FILE
$LIST DESCRIBE(10)
  1000      THIS PROGRAM PROVIDES THREE VERSIONS OF THE FACTORIAL FUNCTION,
  1001      REFERRED TO IN CHAPTER 4,EACH THE OBJECT OF THE OPERATOR
  1002      (LAMBDA  F. OUTPUT  F(N)). THE SECOND AND THIRD VERSIONS ARE OBTAINED
  1003      RESPECTIVELY BY APPLICATION OF THE EQUIVALENCE
  1004                     (LAMBDA Y.E2)(E1) = (LET Y =E2 E1)
  1005      DERIVED IN CHAPTER4. THE FIRST VERSION IS DERIVED FROM MCCARTHY
  1006      (1963) AND IS EQUIVALENT TO AN EXPLICIT PROGRAMMING OF THE PARADOXICAL
  1007      COMBINATOR. IT HAS THE PROPERTY THAT THE NAME OF A FUNCTION IS NOT
  1008      REFERRED TO EXPLICITLY WITHIN ITS BODY. BY DEFINING F=LAMBDA Z.H(Z,H)
  1009      THE EXPRESSION
  1010                  IF X=0 THEN 1 ELSE X*F(X-1)
  1011      CAN BE TURNED INTO A FUNCTION G OF TWO VARIABLES, THE PARAMETER
  1012      WHOSE FACTORIAL IS REQUIRED AND G ITSELF. WHEN THE BLOCK PRIMARY
  1013      INCLUDES A RECURSIVE DEFINITION AT ITS HEAD IT CAN BE REPLACED BY
  1014      A FORM SIMILAR TO THE THIRD VERSION WHICH EXCLUDES USE OF THE BLOCK
  1015      PRIMARY.
END OF FILE
£EMPTY -LOAD
DONE.
£RUN ALEPH* SCARDS=TESTSOURCE(10) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
£RUN ALGO*+-LOAD SCARDS=DATA(10)
EXECUTION BEGINS
DATUM   6,
     720
     720
     720
EXECUTION TERMINATED
```

```
$COMMENT       *** EXAMPLE 11 ***
$LIST TESTSOURCE(11)
  1100      LET TRUE=-1
  1101      LET FALSE=0
  1102      LET N=INPUT
  1103      LET A=ROW N
  1104      LET U=ROW N EACH FALSE
  1105      LET P=0 LET Q=0 LET R=0 LET S=0 LET T=0
  1106      LET C=0 LET M=INPUT
  1107      LET F=LAMBDA.IF(C:=C+1)MOD M¬=0 THEN 0 ELSE
  1108                  LET I=0 WHILE (I:=I+1)<=N DO OUTPUT A@I
  1109      LET I=0 LET J=0
  1110      BEGIN
  1111
  1112        P:=LAMBDA.BEGIN J:=1;Q END;
  1113
  1114        Q:=LAMBDA.
  1115          IF U@J THEN R ELSE
  1116          BEGIN
  1117            A@(I):=J;
  1118            U@(J):=TRUE;
  1119            IF (I:=I+1)>N THEN
  1120              BEGIN F(); S END ELSE P
  1121          END;
  1122
  1123        R:=LAMBDA.IF(J:=J+1)>N THEN S ELSE Q;
  1124
  1125        S:=LAMBDA.IF(I:=I-1)=0 THEN T ELSE
  1126          BEGIN
  1127            J:=A@I;
  1128            U@(J):=FALSE;
  1129            R
  1130          END;
  1131
  1132        DIGITS 2; FIELDS N;
  1133
  1134       I:=1;  LET X=P  WHILE X¬=0 DO X:=X()
  1135
  1136       END
END OF FILE
£LIST DESCRIBE(11)
  1100       THIS PROGRAM USES REGIONS AS DESCRIBED IN CHAPTER 4 TO CONTROL THE
  1101       SEQUENCE OF EXECUTION. EACH REGION (P,Q,R,S, AND T) RETURNS ANOTHER
  1102       REGION AS ITS RESULT. THE SEQUENCE OF EVALUATION IS DEFINED BY
  1103       EVALUATING EACH RETURNED REGION IN TURN UNTIL THE TERMINATING REGION
  1104       (T) IS OBTAINED. THE PROGRAM GENERATES ALL PERMUTATIONS OF
  1105       1,...   ...,N IN LEXICOGRAPHIC ORDER AND SUBMITS THEN TO THE
  1106       SUBPROCESS F. IN THIS CASE F PRINTS EVERY M'TH PERMUTATION. ON ENTRY
  1107       TO Q, A@1,...   ...,A@(I-1) CONTAINS A PARTIAL PERMUTATION DEFINED BY U,
  1108       U@(A@1),...   ...,U@(A@(I-1)) ARE ALL TRUE AND THE REMAINING
  1109       ELEMENTS OF U ARE FALSE. Q'S OBJECTIVE IS TO TRY J AS A VALUE FOR
  1110       A@I. IF U@J IS TRUE THEN THIS IS NOT POSSIBLE AND R IS CALLED TO
  1111       INCREMENT J. IN CASE J CAN OCCUR THEN THIS IS RECORDED AND THE ELEMENT
  1112       COUNTER I IS INCREASED. IF I>N THEN F IS CALLED AND THEN S TO BEGIN
  1113       BACKTRACKING, OTHERWISE P IS CALLED TO INITIALISE J TO TRY FOR VALUES
  1114       OF A@I. R IS USED TO INCRESE J AND CALLS Q TO TRY THIS INCREASED VALUE
  1115       OR S IF J IS OUT OF RANGE. S ESTABLISHES A BACKTRACKING PROCESS BY
  1116       REDUCING THE ELEMENT COUNTER I AND RESETTING J FROM A@I, FINALLY R
  1117       IS CALLED TO TRY THE NEXT VALUE FOR J.
END OF FILE
£EMPTY -LOAD
DONE.
£RUN ALEPH* SCARDS=TESTSOURCE(11) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
£RUN ALGO*+-LOAD SCARDS=DATA(11)
EXECUTION BEGINS
DATUM    7,
DATUM    504,
1 6 2 7 5 4 3
2 4 3 7 6 5 1
3 1 5 7 6 4 2
```

```
3 6 5 7 4 2 1
4 3 7 6 5 2 1
5 2 1 7 6 4 3
5 7 2 6 4 3 1
6 4 3 7 5 2 1
7 2 5 6 4 3 1
7 6 5 4 3 2 1
EXECUTION TERMINATED
```

```
$COMMENT      *** EXAMPLE 12 ***
$LIST TESTSOURCE(12)
           LET TRUE=-1
           LET FALSE=0
           LET N=INPUT
           LET A=ROW N
           LET U=ROW N EACH FALSE
           LET P=FALSE LET Q=FALSE LET R=FALSE LET S=FALSE LET T=FALSE
           LET C=0 LET M=INPUT
           LET F=LAMBDA.IF(C:=C+1)MOD M¬=0 THEN 0 ELSE
                        LET I=0 WHILE (I:=I+1)<=N DO OUTPUT A@I
           LET I=0 LET J=0
           BEGIN

             DIGITS 2; FIELDS N;

             WHILE NOT T DO
             IF P THEN
               BEGIN P:=FALSE; J:=1; Q:=TRUE END ELSE
             IF Q THEN
               BEGIN Q:=FALSE;
                 IF U@J THEN R:=TRUE ELSE
                   BEGIN A@(I):=J;
                     U@(J):=TRUE;
                     IF(I:=I+1)>N THEN BEGIN F(); S:=TRUE END
                                  ELSE P:=TRUE
                   END
               END ELSE
             IF R THEN
               BEGIN R:=FALSE;
                 IF(J:=J+1)>N THEN S:=TRUE ELSE Q:=TRUE
               END ELSE
             IF S THEN
               BEGIN S:=FALSE;
                 IF(I:=I-1)=0 THEN T:=TRUE ELSE
                   BEGIN J:=A@I;
```

```
  1234          U@(J):=FALSE;
  1235          R:=TRUE
  1236        END
  1237          END ELSE
  1238        BEGIN I:=1;
  1239          P:=TRUE
  1240        END
  1241      END
END OF FILE
£LIST DESCRIBE(12)
  1200      THIS IS JUST AN ALTERNATIVE METHOD OF PROGRAMMING THE REGIONS
  1201      IDENTIFIED IN EXAMPLE 11 AND AS SUCH IS DESCRIBED IN CHAPTER 4. HERE
  1202      BOOLEAN VARIABLES P,Q,R,S,T CORRESPONDING TO THE SAME VARIABLES IN
  1203      EXAMPLE 11, PERMIT ACCESS TO APPROPRIATE SECTIONS OF CODE
END OF FILE
£EMPTY -LOAD
DONE.
£RUN ALEPH* SCARDS=TESTSOURCE(12) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
£RUN ALGO*+-LOAD SCARDS=DATA(12)
EXECUTION BEGINS
DATUM   7,
DATUM  5504,
1 6 2 7 5 4 3
2 4 3 7 6 5 1
3 1 5 7 6 4 2
3 6 5 7 4 2 1
4 3 7 6 5 2 1
5 2 1 7 6 4 3
5 7 2 6 4 3 1
6 4 3 7 5 2 1
7 2 5 6 4 3 1
7 6 5 4 3 2 1
 EXECUTION TERMINATED
```

```
$COMMENT      *** EXAMPLE 13 ***
$LIST TESTSOURCE(13)
  1300      LET F=LAMBDA N,A.
  1301           A(IF N=0 THEN 1 ELSE
  1302           LET J=0
  1303           BEGIN F(N-1,LAMBDA X.J:=X);
  1304                 J*N
  1305           END)
  1306
  1307      F(INPUT,LAMBDA X.OUTPUT X)
  1308
END OF FILE
$LIST DESCRIBE(13)
  1300      THIS SIMPLE EXAMPLE, IN CONJUNCTION WITH THE NEXT IS INTENDED TO
  1301      DEMONSTRATE ONE OF THE LESS FORTUNATE ASPECTS OF THE USE OF A
  1302      VARIABLE TABLE. CONSIDER THE FUNCTION
  1303         F=LAMBDA N. IF N=0 THEN 1 ELSE N*F(N-1)
  1304      WHICH RETURNS THE FACTORIAL OF ITS ARGUMENT. PROVIDING  THIS
  1305      FUNCTION F WITH AN EXTRA PARAMETER A, WHICH IS A FUNCTION TO BE
  1306      APPLIED TO THE RESULT OF F, AND ADJUSTING THE INTERNAL CALL OF F TO
  1307      SUPPLY LAMBDA X.J:=X AS THE CORRESPONDING ACTUAL PARAMETER, FOR SOME
  1308      LOCAL J, COMPLETES THE FORMAL CONSTRUCTION OF THIS PROGRAM. THIS
  1309      TRANSFORMATION IS O.K. AND THE PROGRAM COMPUTES THE FACTORIAL CORRECTLY.
END OF FILE
$EMPTY -LOAD
DONE.
$RUN ALEPH* SCARDS=TESTSOURCE(13) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
$RUN ALGO*+-LOAD SCARDS=DATA(13)
EXECUTION BEGINS
DATUM    6,
        720
EXECUTION TERMINATED
```

```
$COMMENT       *** EXAMPLE 14 ***
$LIST TESTSOURCE(14)
  1400      LET F=LAMBDA N,A.
  1401            IF N=0 THEN A(1) ELSE
  1402            LET J=0
  1403            BEGIN F(N-1,LAMBDA X.J:=X);
  1404                   A(J*N)
  1405            END
  1406
  1407      F(INPUT,LAMBDA X.OUTPUT X)
END OF FILE
$LIST DESCRIBE(14)
  1400      THIS PROGRAM IS OBTAINED FROM THE PROGRAM OF EXAMPLE 13 BY THE SIMPLE
  1401      DEVICE OF MOVING THE CALL OF A TO APPLY DIRECTLY TO ITS ARGUMENTS.
  1402      THE PROGRAM IS NO LONGER VALID HOWEVER BECAUSE OF THE NATURE OF THE ACTUAL
  1403      PARAMETER SUPPLIED FOR A,NAMELY LAMBDA X.J:=X. WHEN THE CALL OF THIS PARAMETER
  1404      IS MOVED INSIDE THE LOCAL DECLARATION OF J ITS APPLICATION ALTERS THE
  1405      CURRENT INCARNATION RATHER THAN THE PRECEDING.
END OF FILE
$EMPTY -LOAD
DONE.
$RUN ALEPH* SCARDS=TESTSOURCE(14) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
$RUN ALGO*+-LOAD SCARDS=DATA(14)
EXECUTION BEGINS
DATUM   6,
         0
EXECUTION TERMINATED
```

```
$COMMENT       *** EXAMPLE 15 ***
$LIST TESTSOURCE(15)
   1500      LET F=LAMBDA N,A.
   1501            IF N=0 THEN A@0:=1 ELSE
   1502            LET J=ROW 0
   1503            BEGIN F(N-1,J);
   1504                  A@0:=N*J@0
   1505            END
   1506
   1507      LET J=ROW 0
   1508      BEGIN F(INPUT,J);
   1509            OUTPUT J@0
   1510      END
END OF FILE
$LIST DESCRIBE(15)
   1500      THE ERROR ENCOUNTERED IN EXAMPLE 14 IS CORRECTED BY SUPPLYING A VECTOR
   1501      AS THE ADDITIONAL ACTUAL PARAMETER, RATHER THAN A FUNCTION.
END OF FILE
$EMPTY -LOAD
DONE.
$RUN ALEPH* SCARDS=TESTSOURCE(15) 2=-LOAD
EXECUTION BEGINS
EXECUTION TERMINATED
$RUN ALGO*+-LOAD SCARDS=DATA(15)
EXECUTION BEGINS
DATUM    6,
       720
EXECUTION TERMINATED
```

APPENDIX 4 - REFERENCES

1. Bandat, K. (1968) "On the formal definition of PL1" AFIPS (Spring) 1968, Vol. 32, p.363.

2. Bohm, C. (1964a) "The CUCH as a formal and description language" in Steel, T.B. (1964c), p.179.

3. Bohm, C. and Gross, W. (1964b) "Introduction to the CUCH" in Automata Theory. Ed. Caianiello, Academic Press.

4. Bohm, C. and Jacopini, G. (1966) "Flow diagrams, Turing machines and languages with only two formation rules", C.A.C.M. Vol. 9, 1966, p.366.

5. Church, A. (1941) "The Calculi of Lambda - Conversion", Annals of Math. Studies No. 6, Princeton University Press.

6. Conway, M.E. (1963) "Design of a separable transition diagram compiler", C.A.C.M. Vol. 6, 1963, p.396.

7. Cooper, D.C. (1967) "Bohm and Jacopini's reduction of flow charts" (letter), C.A.C.M. Vol. 10. No. 8, August 1967, p.463.

8. Cooper, D.C. (1968) "Some transformations and standard forms of graphs with applications to computer programs" Machine Intelligence 2, Edinburgh University Press.

9. Curry, H.B. and Feys, R. (1958) "Combinatory Logic"
Volume 1. North-Holland, Amsterdam.

10. Dijkstra, E.W. (1960) "Recursive Programming"
Num. Math. Vol. 2, p.312.

11. Dijkstra, E.W. (1961a) "Making a translator for Algol 60"
Annual Review in Automatic Programming, Vol. 3, p.347.

12. Dijkstra, E.W. (1961b) "An Algol 60 translator the the XI"
Annual Review in Automatic Programming, Vol. 3, p.329.

13. Dijkstra, E.W. (1962) "An attempt to unify the constituent concepts of serial program execution",
Symbolic Languages in Data Processing,
Proc. ICC Symp. (Rome) 1962. p.237.

14. Dijkstra, E.W. (1963) "On the design of machine independent programming languages", Annual Review in Automatic Programming, Vol. 3, 1963, p.27-42.

15. Dijkstra, E.W. (1968) "Goto statement considered harmful"
(letter), C.A.C.M. Vol. 11, p.147,

16. Dijkstra, E.W. (1969) "Notes on structured programming"
EWD249, TH-Report 70, Technological University
Eindhoven.

17. Feldman, J. and Gries, D. (1968) "Translator writing systems"
C.A.C.M. Vol. 11, p.77.

18. Floyd, R.W. (1967) "Assigning meanings to programs" AMS Symposium in Appl. Maths. 19.

19. Garwick, J.V. (1964) "The definition of programming languages by their compilers" in Steel, T.B. (1964c), p.139.

20. Hoare, C.R. (1969) "An axiomatic basis for computer programming" C.A.C.M. Vol. 12, Nov.1969, p.576.

21. Kanner, H., Kosinski, P. and Robinson, C.L. (1965) "The structure of yet another Algol compiler" C.A.C.M. Vol. 8, p.427.

22. Landin, P.J. (1963a) "The mechanical evaluation of expressions", Computer Journal, Vol. 6, p.308.

23. Landin, P.J. (1963b) "The λ-calculus approach" Advances in Programming & Non-Numerical Computing, Ed. Fox.

24. Landin, P.J. (1964) "A formal description of Algol 60" in Steel, T.B. (1964c), p.266

25. Landin, P.J. (1965a) "An abstract machine for designers of computing languages" IFIP Proc.(1965), Vol.2.

26. Landin, P.J. (1965b) "A correspondence between Algol 60 and Churches λ-calculus" C.A.C.M. Vol. 8 1965, p.89 (part 1), p.158 (part 2).

27. Landin, P.J. (1966) "The next 700 programming languages" C.A.C.M. Vol. 9 1966, p.157.

28. McCarthy, J. (1960) "Recursive functions of symbolic expressions and their evaluation by machine" C.A.C.M. Vol. 3, p.184.

29. McCarthy, J. (1962a) "Towards a mathematical science of computation", IFIP Proc. (1962), P.21.

30. McCarthy, J. et al (1962b) LISP 1.5 Programmers Manual

Computation Lab. Report MIT. 1962.

31. McCarthy, J. (1963) "A basis for a mathematical theory of computation" in "Computer programming and formal systems" eds. P. Braffort and D. Herschberg, North Holland, Amsterdam.

32. McCarthy, J. (1964) "A formal description of a subset of Algol" in Steel, T.B. (1964c), p.1.

33. McCarthy, J. (1965) "Problems in theory of computation" IFIP 1965, Vol.1.

34. McCarthy, J. and Painter, J. (1967) "Correctness of a compiler for arithmetic expressions" AMS Symposium in Appl. Math. 19, 1967.

35. Naur, P. (1963) "Revised report on the algorithmic language, Algol 60", C.A.C.M. Vol.6, January 1963, p.1-17.

36. Nivat, M. and Nolin, N. (1964) "Contribution to the development of Algol semantics" in Steel, T.B. (1964c), p.148.

37. Rice, J.R. (1968) "The goto statement reconsidered" (with reply by E.W. Dijkstra), C.A.C.M. Vol. 11, No.8, August 1968.

38. Steel, T.B. (1964a) "Beginnings of a theory of information handling", C.A.C.M. 7 (1964), p.96.

39. Steel, T.B. (1964b) "A formalization of semantics for programming language description", in Steel, T.B. (1964c), p.25.

40. Steel, T.B. (Ed.) (1964c) "Formal language description languages for computer programming", Pro. IFIP Conf. Baden, 1964. North-Holland Pub. Co.

41. Strachey, C. (1964) "Towards a formal semantics" in Steel, T.B. (1964 c) p.198.

42. van Wijngaarden, A. (1962) "Generalised Algol",
Symbolic Languages in Data Processing,
Proc. ICC Symp. (Rome) 1962, p.409.

43. van Wijngaarden, A. (1964) "Recursive definition
of syntax and semantics" in
Steel, T.B. (1964c), p.13.

44. Wirth, N. (1963) "A generalisation of Algol",
C.A.C.M. Vol.6, 1963, p.547.

45. Wirth, N. (1966) "Euler, a generalisation of Algol
and its formal definition",
C.A.C.M. Vol.9, 1966, p.13, p.89.