The Pragmatic Formalization of Computing Systems Relative to a given High-Level Language

BY

Derek John Cooke.

Thesis submitted for the degree of Doctor of Philosophy of the University of London.

Abstract

Using a generalization of the 3-place operator S [116], derived from Markov's substitution operator $\sum$ [74], we demonstrate a procedure for specifying the semantics of high-level programming languages in terms of a small set of fundamental operations. These intrinsic functions are defined relative to a realistic, though not mathematically aesthetic, model into which implementation restrictions must also be incorporated.

Application of our system yields the specification of a computing system as seen by the user of a high-level language; and which has been determined partially by the language designer and partially by the implementor of the language on the specific configuration in question.

This thesis contains in its first 4 chapters mainly supporting material. The reader is therefore advised that he will probably find it easier to read if he starts his study at Ch.V. He will then find it necessary to read Ch.III before proceeding to Ch.VII.

Acknowledgements

I should like to thank the Science Research Council for financing the research reported herein by way of a research studentship. In like manner I am most grateful to the University of London, first at the Institute of Computer Science and then at the Department of Computing and Control of Imperial College, for affording me facilities for the pursuance of said research.

At a more personal level I should like to record my indebtedness to my fellow research students, for their willingness to listen to my ideas and eagerness to 'correct' them. Special thanks are due to Dr. Tom Wesselkamper, also a former ICS research student, without whose work mine may never have begun.

The biggest debt of gratitude I owe is to my supervisor, Eric Nixon, former Dean of Research Students of the Institute. No mere words seem adequate to describe the experience of being one of his students - it's certainly not an easy life but without doubt you learn a lot. Thank you.

Last but not least, thanks to my typists, Miss Helen Knight and Mrs. Betty Wright.

Contents

1. Introduction.

2. Non-Extensible Syntax and Left-to-Right Parsing.

2.1 On Syntax Modification.

2.1.1 Introduction.

2.1.2 The Algorithm.

2.1.2.1 Derivation of Left-factored Form.

2.1.2.2 Equivalence of LFF and GNF.

2.1.2.3 Investigation of NBU Conditions.

2.1.2.4 Miscellaneous checks and reductions.

2.1.2.5 The Composite Algorithm.

2.2 On Syntax Construction.

2.2.1 The Construction Rules.

2.2.1.1 Local Rules.

2.2.1.2 Global Rules.

2.2.2 Identities.

2.2.2.1 Local Identities.

2.2.2.2 Non-local Identities.

2.2.3 General Remarks and Hints.

2.2.3.1 Connectedness.

2.2.3.2 Finite Generation.

2.2.3.3 Minimality.

2.2.3.4 On the occurence of Λ

2.2.3.4.1 Factorization.

2.2.3.4.2 Lists and Sequences.

2.2.3.5 Trivial Classes.

2.2.4 Abreviated BNF.

2.2.5 Consequences.

2.2.6 Summary.

3. The Language X : An Introduction.

3.1 Informal Description.

3.2 Verification that the given Syntax is suitable for Left-to-right Parsing.

3.3 Executive Semantics.

4. Base Functions of the Source Language.

4.1 Markov Algorithms.

4.2 Examples of two elementary algorithms.

4.3 Extended Markov Algorithms (EMAs).

4.4 Composition of Algorithms and Embedding.

4.4.1 Sequential Composition.

4.4.2 Embedding.

4.5 Iteration and Ramification.

4.5.1 Formal derivation of RAM and PTL.

4.5.2 Examples.

5. The Program Space and the Operator k.

5.1 Informal Discussion.

5.2 Some Mathematical Definitions.

5.2.1 Lists.

5.2.2 Digraphs.

5.2.3 Finite Disjoint Unions.

5.3 Description of the Space.

5.3.1 Non-Mathematical Description.

5.3.2 Mathematical Formulation.

5.3.3 The k - completion of CPS.

6. The Carabiner Language.

6.1 The Operator S.

6.2 The Operator k.

6.3 The Operator e.

6.4 Procedures and Functions.

6.5 On Orders of Evaluation and Control Functions.

6.5.1 Non-Control Functions.

6.5.2 Control Routines.

6.5.3 Activation of Control Routines.

6.5.4 The 'Next Instruction'.

6.6 Dormant Procedures.

6.6.1 Examples.

6.6.2 Discussion.

6.7 On Position Specification and Value Selection in CPS.

6.8 Macros and Set Theory.

6.8.1 Operations on CPS.

6.8.2 The S definition of set operations and predicates.

6.8.3 On S abbreviations and pointer/value associations.

7. Translation and a Formal Definition of X.

7.1 Parsing Strategy.

7.2 The Semantic Injections of Language X.

7.2.1 Control Translations in X.

7.3 A Translated X Program.

8. Properties of the Program Space.

8.1 On Derivatives, Neighbourhoods, Relatives and STRUCTs.

8.2 A Dynamic Topology for CPS.

8.3 On Stability of Programs.

8.4 On Modal Substructures and Coercions.

9. On Describing Other Programming Language Features.

9.1 Assignment and I/O.

9.2 Transfer of Control and Block Structure.

9.3 Functions and Parameters.

9.4 Type Checking.

9.5 Structured Data.

9.6 String Manipulation.

10. Closing Remarks.

References.

Appendix: The Removal of 'Goto' s.

CHAPTER 1

INTRODUCTION

The main object of the research reported herein is the setting up of a formal (pragmatic) system for describing the semantics of computing systems.

Such a descriptive device must first deal with the syntax of programs driving the computing system. At the present the terms 'syntax' and 'semantics' seem to be redefined by researchers to suit their individual needs; therefore, before proceeding we attempt to clarify our position. Following Carnap [16]:-

> " Every situation in which a language is employed involves three principal factors: (1) the speaker, an organism in a determinate condition within a determinate environment; (2) the linguistic expressions used, these being sounds or shapes (e.g. written characters) produced by the speaker (for instance, a sentence consisting of certain words of the French language); and (3) the objects, properties, states of affairs, or the like, which the speaker intends to designate by the expressions he produces - and which we term the designata of the expressions (thus e.g. the colour red is the designatum of the French word 'rouge'). The entire theory of an object language is called the semiotic of that language; this semiotic is formulated in the meta-language. Within the semiotic of a language, three regions may be distinguished according to which of the three aforementioned factors receive attention. Thus, an investigation which refers explicity to the speaker of the language - no matter whether other factors are drawn in or not - falls in the region of pragmatics. If the investigation ignores the speaker, but concentrates on the expressions of the language and their designata, then the investigation belongs to the province of semantics. Finally, an investigation which makes no reference either to the speaker or to the designata of the expression, but attends strictly to the expressions and their forms (the ways expression are constructed out of signs in determinate order), is said to be a formal or syntactical investigation and is counted as belonging to the province of (logical) syntax. "

In computing terms we interpret this as follows: taking as an example the arithmetic operation 'plus' acting on two integers, then:

(a) the syntax specifies the string

$\alpha + \beta$ where α and β are arbitrary strings of digits,

(b) the semantics of the above is the 'abstract' notion of a particular way of combining the quantities denoted by the strings α and β, and

(c) the pragmatics, is a well-defined process for realizing the semantic notion; e.g. in a machine language we may have:

```
LOAD  α
ADD   β
```

or, in a machine-independent form:-

$(\alpha + \beta) \equiv$ <u>if</u> $\beta = 0$
<u>then</u> α
<u>else</u> (<u>if</u> $\beta > 0$
<u>then</u> $(\alpha^{+} + \beta^{-})$
<u>else</u> $-((-\alpha) + (-\beta)))$

- where $-\alpha$ denotes negation

$\alpha^{+} \to \alpha + 1$ is the successor function and

$\beta^{-} \to \beta - 1$ is the predecessor function.

Our system, Carabiner, affords formal descriptions of Computing systems based on syntactically inextensible high-level programming languages (see [89] for classification of various types of extensibility)

and consists of three parts; a model of the program space, a language and a translator. It is machine and language independent but for any given system the model is characterised by the representation used in the language upon which the system is based - i.e. given a language and its implementation, then the specification of this (system-dependent) dialect of the language via the Carabiner model enables the exact output from any program written in the language and executed by this implementation to be determined by means of the model. Conversely, Carabiner may be used to define an abstract machine for the language and hence to give a prescription for its implementation against which the correctness of the resulting system can be tested. In this respect, Carabiner is a definitional UNCOL [79, 98, 99].

In particular we note that there is no idealization <u>within</u> the model. Arithmetic quantities are neither assumed to have boundless rangê nor to be continuous.

Carabiner places no constraints upon the designer or the implementer, save that of well-definedness, nor upon the programmer provided his program is legal - i.e. he is permitted to write nonsense if the language so allows.

It has been shown by Wesselkamper [116] that the substitution operator S, defined by:-

$$S\ a\ b\ c = \begin{cases} c & \text{if } a = b \\ a & \text{otherwise} \end{cases}$$

(where a, b, c are 'values')

is sufficient for defining any n-adic operator over an m-valued logic for any given m.

By additionally using a suitable naming operator, we can thus completely describe any stored program computer in terms of its memory states. Hence, by defining a hierarchy of intermediate languages, it should be possible to express the operations of any high-level programming language in terms of these two 'basic' operations only; this, however, necessitates (at some stage) a translation from the representations used in the source language into some binary representation. Carabiner does not descend to such levels but, instead, extends the domain and codomain (range) of S so as to be able to manipulate assembler level operations. These in turn may be defined by means of S or, using a high-level representation, by means of Extended Markov Algorithms.

Wesselkamper demonstrates his Crampon system by modelling Algol - ε - a subset of Algol-60. However, the implied translation from Algol-ε to Crampon is nowhere formalised; nor for that matter is the translation from Algol-60 into Algol- ε [84, 85], Carabiner's syntax directed translator provides the mechanism for such formalization and yields a (Crampon-like) intermediate language; thus providing a well-defined procedure for realising Algol-60, Algol- ε and most other programming languages in terms of the basic operations of Crampon and Carabiner.

For any language which has an inextensible syntax [89], it may be possible to devise a grammar by which any sentence of the language is recognisable by a very simple 'left-to-right' parsing machine. The conditions required for this to be so, the construction of suitable grammars, and modifications which may be applied in an attempt to derive suitable grammars from ones which violate the conditions are discussed at great length in chapter 2.

Throughout the thesis, we use an example language, language X, to illustrate salient points. This language is first introduced in chapter 3, where we give its syntax and show that it is suitable for direct recognition. We also discuss the desired semantics of X in an informal way.

In the chapters that follow we shall assume that any grammar used satisfies the above mentioned conditions and hence explicit mention of this is usually avoided.

The basic functions used (ultimately) by the high-level languages are described in a uniform way by means of Extended Markov Algorithms as defined in chapter 4.

In chapter 5 the mathematical space, used to model the states of the computing system, is introduced together with the first primitive Carabiner operation k which is used as a digraph traverser. The subsequent chapter describes the Carabiner language in full and outlines its development into its present form.

Having made precise the language and the space in which it acts we give, in chapter 7, a full definition of language X in an extension of BNF. The form of this definition is not unlike the attribute grammars of Bochmann [10, 11], but we feel that our system is more uniform and concrete.

A topological description of the space is given in chapter 8 and is linked to the concept of the stability of a program relative to its data.

The relevance and applicability of Carabiner to programming features not present in language X is considered in chapter 9 by discussing the Carabiner definition of a set of languages specially designed to test such definitional mechanisms.

In the last chapter we make concluding remarks and compare our efforts with those of other workers in the field.

Finally we note that the Carabiner language was originally intended to be a 'structured' language and, although this constraint is not imposed on the user, the language can be made free of explicit goto commands. If this is done, the modelling of high-level languages in which such a feature exists will necessitate restructuring of the program. A simple algorithm for doing this at flow chart level is given in the appendix.

CHAPTER 2

NON-EXTENSIBLE SYNTAX AND LEFT-TO-RIGHT PARSING

In order to be able to perform a syntax-directed translation from a high-level source language into Carabiner, in such a way that no translation actions need to be undone because of wrongly recognised constructs; it is most convenient if we can parse sentences of the language directly from left-to-right without 'backing-up'.

Any (top-down) syntax-directed translator has to deal with this problem and, as practical evidence has shown [35] [46], the simplest way to do this is to modify the language's syntactic definition; or alternatively to construct the grammar in a way so that such analysis is always possible.

Trivially, this means that all syntactic structure needs to be known before the execution of _any_ program in the relevant language, and hence this precludes any syntactic extensions being made during the execution of such a program; hence we must dictate that, as we are using a naïve top-down parser in order to simplify the syntax analysis as much as possible, _no syntactic extensions are permissible within the source language_.

In §2.1 we give an algorithm which attempts to modify a given syntax into an equivalent one which is suitable for left-to-right no-lookahead parsing whilst in §2.2 we draw up a set of rules for the construction of syntax which _is_ directly parsable from left-to-right.

These sections are, in a sense, inverses of each other and thus either may be omitted; however we feel that greater insight is obtained by considering both.

Though most of the work reported in §2.1 is conceptually simple, its formalizations seems complicated; in an attempt to present the material in a more readable form we therefore develop the arguments by means of a series of examples of increasing complexity. Consequently this chapter may seem disproportionately large compared with subsequent chapters. For this, and for any duplication which may occur from our presentation of the material of §2.1 and §2.2, we ask the reader's forbearance.

2.1 On Syntax Modifications

Throughout this section we make extensive use of various relations and their representations as digraphs and diagrams, which enable us (i) to <u>see</u> the relevant structure, and (ii) to formalise the underlying mathematical framework more easily. Of particular note are the 'skeleton' and the 'Backup Diagram'; the skeleton dictates the order in which several well-known modifications can be more usefully applied and also leads to a formalization of a locked grammar.

Theorems relating to secondary locks, and the equivalence of Left Factored Form and Greibach Normal Form [51] conclude the first half of the section.

For completeness, an algorithm to remove a 'lock' is given at the end of section, although this is not the work of the author.

Our formalization of the Backup Diagram (and the associated 'following-sequence'), the usage graph and the linkage graph are new and though most of the results obtained from them and their associated relations are well-known the given derivation is more precise than can be obtained by a non-mathematical approach.

2.1.1 Introduction

An algorithm, SYMAL, is developed which attempts to transform sets of BNF production rules into a form such that any sentence generated by the grammar, defined by such a set, may be parsed directly from left-to-right with no look-ahead.

It is necessary, as a first step, to obtain a grammar in which each production is of the form:-

$$\langle\text{Class}\rangle ::=^{\dagger} \alpha_1|\alpha_2|\ldots|\alpha_n$$

where: (i) $n \in \mathbb{N}$

(ii) α_n may be Λ (the void option)

$\alpha_i \neq \Lambda : 1 \leq i < n$

(iii) each α_i : $(1 \leq i \leq n)$ and $\alpha_i \neq \Lambda$ begins with a different terminal symbol.

such a form will be known as a Left factored form (LFF).

Notice that LFF assumes the ordering of alternatives in a BNF production is important and that the order given is that used by the parser.

Further transformation will be required in order to obtain a grammar such that sentences in the language may be recognised without the necessity for 'backtracking'.

e.g. The sentence 'xx' which may be generated by the grammar:

$$C \rightarrow xCx \mid \Lambda$$

cannot be recognised without backtracking.

However if the grammar is transformed to the equivalent form:

$$C \rightarrow xxC \mid \Lambda$$

Such backtracking is not necessary.

† alternatively we use '→' instead of '::='

Naturally the manipulations discussed herein bear a close relationship to the formal theory of LR (left-to-right)† and LF (left-factored) [119] grammars, but our approach is more pragmatic though several theoretical aspects have been considered.

Of the modifications presented, those related to the 'Backup Diagram' are new. Left recursion is dealt with in Greibach [49], in fact we note that our left factored form, if it exists, is equivalent to Greibach Normal Form [51]; therefore, any of the identities derived in §2.1.2.3 may be applied to any context-free grammar (via its normal form) provided that grammar is unambiguous and that each non-terminal of the grammar is capable of generating a terminal string or Λ (i.e. any cfg which satisfies the sufficient conditions for conversion into LFF).

The aim of SYMAL is the same as Foster's Syntax Improving Device [46], in which back-substitution of class definitions is used in an attempt to recover from violation of no-backup (NBU) conditions [59]. We use this and other techniques applied to classes which are selected by examination of suitable relations.

Throughout this chapter we shall usually denote class names by upper case letters, subscripted where necessary, or subscripted 'C's

$$A, B, C, \ldots \text{ etc.}$$

or

$$C_1, C_2, C_3, \ldots \text{ etc.}$$

---

†For discussion of the theory of general LR parsing methods, the reader is referred to a recent survey paper by Aho & Johnson [1].

Terminal strings are denoted by lower case letters or subscripted 'T's; hence we have such productions as:-

$$A \quad ::= \quad Aa|BC|Dd|\Lambda$$

$$C_7 \quad ::= \quad C_6x|yC_8|C_1C_2$$

$$C_2 \quad \rightarrow \quad T_1C_1|C_3T_7T_{37}|\Lambda$$

Elsewhere Greek letters are used in locally defined roles.

2.1.2 The Algorithm

There are two main parts to the algorithm, the derivation of LFF, §2.1.2.1, and removal of NBU violations §2.1.2.3. Various other checks and modifications which may be made at suitable points either within or outside of the two major steps are given in §2.1.2.4, the composite algorithm being assembled in §2.1.2.5.

2.1.2.1 Derivation of a left-factored form

We simply describe the transformations and illustrate them by examples. Justification that such transformations exist and preserve ambiguity is given elsewhere [49], [50].

There are two fundamental procedures, LFAC and SYNSUB. LFAC checks for left-factors (i.e. meaningful leading common string factors in the options of a class definition). These factors are located and removed pairwise, a new class being created to hold the remaining parts of the factored options.

e.g.: $A ::= ab|ac$

becomes $\begin{cases} A ::= aX \\ X ::= b|c \end{cases}$

and $A ::= ab|ac|ad$

becomes
$$\begin{cases} A ::= aX|ad \\ X ::= b|c \end{cases}$$

hence
$$\begin{cases} A ::= aY \\ X ::= b|c \\ Y ::= X|d \end{cases}$$

We note that ambiguous grammars may give rise, ultimately†, to productions of the form:

$$A ::= \Lambda|\Lambda ,$$

such a production is easily detected and causes termination of the algorithm.

The definition of LFAC, given above, is normally sufficient for our purposes, however trouble can occur when equivalent classes have differing forms and LFAC will be suitably modified later.

Before describing SYNSUB (partial SYNtax SUBstitution), we note the well-known identity used to remove left recursion:

$$A \rightarrow Ab|c$$

equivalent to:

$$\begin{cases} A \rightarrow cX \\ X \rightarrow bX|\Lambda \end{cases}$$

or, in a more complex case:

$$A \rightarrow Ab|c|d$$

$$\equiv \begin{cases} A \rightarrow cX|dX \\ X \rightarrow bX|\Lambda \end{cases}$$

---

† either in LFAC or other routines described later.

Here, of course, a,b,c,d, may be classes or terminals. If $b = \Lambda$ then the original productions for A yield ambiguous parses (of c and d) and we require that the algorithm should halt and fail. The associated subroutine is called DLR.

Now, SYNSUB takes a class definition and attempts to find a class name leading an option, if it fails then SYNSUB causes no modification; otherwise, if the class is the same as the class being processed then apply the identity to remove left recursion (i.e. call DLR), if it is different then substitute the definition of the class (expanded where necessary) for the occurrence of the class name:

Hence:

$$\begin{cases} A ::= Bb|c \\ B ::= c|a \end{cases}$$

becomes

$$\begin{cases} A ::= cb|ab|c \\ B ::= c|a \end{cases}$$

then factorise, so:

$$A ::= cb|ab|c$$

becomes

$$\begin{cases} A ::= cX|ab \\ X ::= b|\Lambda \end{cases}$$

The resultant (current) class is then processed by SYNSUB until no leading classes occur in the resultant class definition.

If we combine the modifications above in the following way we are part way to the required (sub) algorithm. Call this version AI.

<u>AI</u> (i) (left) factorise all definitions

(ii) Remove all direct-left-recursion

(iii) Process each definition, including added definitions, by SYNSUB. (This generally involves further factorization and removal of direct left recursion of individual definitions).

We now give some examples of simple grammars and their modifications under the above algorithm. (Step-by-step derivations of all modifications of the following examples are given elsewhere [25].)

<u>Grammar A</u>:

$$X ::= ab|abc|abcd|ac$$

Under AI this yields:

$$\begin{cases} X ::= aX_3 \\ X_1 ::= c|\Lambda \\ X_2 ::= cX_4|\Lambda \\ X_3 ::= bX_2|c \\ X_4 ::= d|\Lambda \end{cases}$$

<u>Grammar B</u>:

$$\begin{cases} A ::= Ab|B \\ B ::= Ac|d \end{cases}$$

Under AI this yields:

$$\begin{cases} A ::= dA_1A_2 \\ B ::= dB_1 \\ A_1 ::= bA_1|\Lambda \\ A_2 ::= cA_1A_2|\Lambda \\ B_1 ::= bA_1A_2|cB_2|\Lambda \\ B_2 ::= bA_1A_2c|c|\Lambda \end{cases}$$

Grammar C:

$$\begin{cases} A ::= Ba \mid c \\ B ::= Cb \mid Dd \\ C ::= Ae \mid g \\ D ::= Af \end{cases}$$

Under AI this yields:

$$\begin{cases} A ::= gbaA_1A_2 \mid cA_1A_2 \\ B ::= gbB_2 \mid cA_1A_2B_3 \\ C ::= gC_1 \mid cA_1A_2e \\ D ::= gbaA_1A_2f \mid cA_1A_2f \\ A_1 ::= ebaA_1 \mid \Lambda \\ A_2 ::= fdaA_1A_2 \mid \Lambda \\ B_1 ::= aA_1A_2eb \mid \Lambda \\ B_2 ::= aA_1A_2B_4 \mid \Lambda \\ B_3 ::= eb \mid fd \\ B_4 ::= eb \mid fd \\ C_1 ::= baA_1A_2e \mid \Lambda \end{cases}$$

We now consider some less straightforward examples:

Grammar D:

$$\begin{cases} A ::= Ba \mid C \\ B ::= b \\ C ::= d \mid Da \\ D ::= b \mid e \end{cases}$$

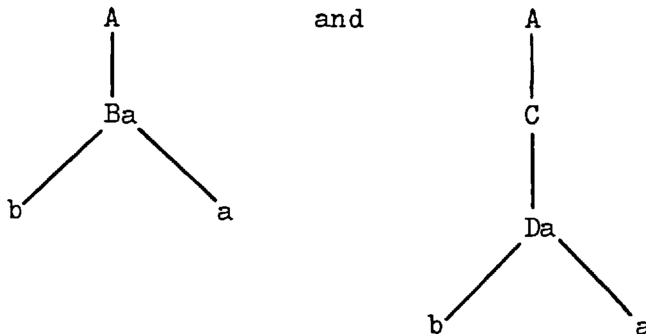
Proceeding as in the previous cases gives:

	A ::= Ba|C	synsub on A
a)	A ::= ba|C	synsub on A
b)	A ::= ba|d|Da	synsub on A
c)	A ::= ba|d|ba|ea	factorise A
d)	$\begin{cases} A ::= baA_1 \mid d \mid ea \\ A_1 ::= \Lambda \mid \Lambda \end{cases}$	A now OK ✳

Whence $A_1 \Rightarrow$ ambiguous grammar, moreover if we trace the steps back from d), we have

d)	$\begin{cases} A_1 ::= \Lambda \mid \Lambda \\ A ::= baA_1 \end{cases}$
c)	A ::= ba|ba
b)	A ::= ba|Da
a)	A ::= Ba|C

hence, the two parses of 'ba' are:



Thus, when our method detects ambiguity in a grammar, it also exhibits an ambiguous parse.

Next we consider a non-ambiguous grammar for which the method fails:

<u>Grammar E</u>:

$$\begin{cases} A ::= B \mid a \\ B ::= C \mid b \\ C ::= Bd \end{cases}$$

Applying AI to grammar E results in the creation of an infinite sequence of additional classes, i.e.:

	A	::= B|a	
			synsub A
a)	A	::= C|b|a	
			synsub A
b)	A	::= Bd|b|a	
			synsub A
c)	A	::= Cd|bd|b|a	
			factorise
d)	$\begin{cases} A & ::= Cd\|bA_1\|a \\ A_1 & ::= d\|\Lambda \end{cases}$		synsub A
e)	A	::= Bdd|bA_1|a	
			synsub A
f)	A	::= Cdd|bdd|bA_1|a	
			factorise
g)	$\begin{cases} A & ::= Cdd\|bA_2\|a \\ A_2 & ::= dd\|A_1 \end{cases}$		synsub A
h)	A	::= Bddd|bA_2|a	
			synsub A
i)	A	::= Cddd|bddd|bA_2|a	
			factorise
j)	$\begin{cases} A & ::= Cddd\|bA_3\|a \\ A_3 & ::= ddd\|A_2 \end{cases}$		

etc. The extension goes on for ever because the classes B and C are mutually left-recursive. However consider what happens if we process the productions in a different order. To achieve this affect we do not yet modify the algorithm but change the order of the productions of the grammar

Grammar E': (root is A)

$$\begin{cases} B ::= C|b \\ A ::= B|a \\ C ::= Bd \end{cases}$$

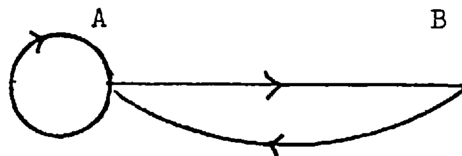
this yields:

$$\begin{cases} B ::= bB_1 \\ A ::= bB_1 | a \\ C ::= bB_1 d \\ B_1 ::= dB_1 | \Lambda \end{cases}$$

Although AI fails on grammar E its success on grammar E' now leads us to attempt modifications to the algorithm to enable it to succeed on a larger class of grammars.

The fundamental problem that faults the simplification technique of AI is that of removing (indirect) left recursion between classes. If we abstract this left-recursive property between the productions to give a relation ρ on C^2 where C is the set of classes in a grammar such that $C_1 \rho C_2$ iff C_2 occurs as the first element of an option in the definition of C_1. We may then represent this in the 'diagram':

Using this symbolism, grammar B, generates the following diagram.



Now if we consider our algorithm after stage (ii) i.e. when all preliminary factorization and removal of direct left-recursion has been done then the corresponding diagram has no loops i.e. $C_i \not\rho C_i$

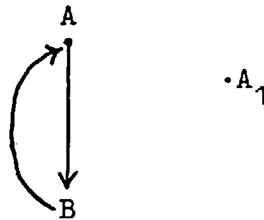
for any i, and no duplicated arcs, i.e. if $C_i \rho C_j$ then there is only one arrow joining C_i to C_j (though there may be one in the reverse direction). Hence we have a directed graph or digraph [52].

Using digraph notation (described in chapter 5 and elsewhere [52]), let us consider the digraphs generated by the given grammars after stage (ii) of AI. We shall call these graphs <u>skeletons</u>.

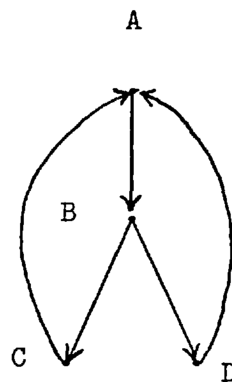
<u>Skeleton A</u>

X

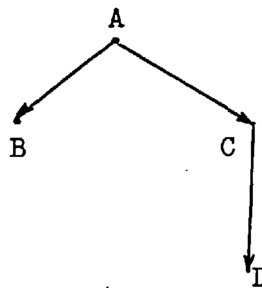
<u>Skeleton B</u>



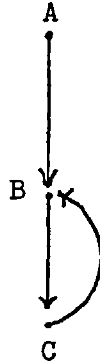
<u>Skeleton C</u>



<u>Skeleton D</u>



Skeleton E



Now with each node α we associate a set

$$s_\alpha = \{x : \alpha\rho x\} .$$

This set is sometimes called the outbundle of α.

We note at this stage that (i) for no α do we have $\alpha \in s_\alpha$. and (ii) our ultimate aim is to reduce the skeleton to a set of discrete points. This is equivalent to the condition $s_\alpha = \emptyset$ for all α.

From the outbundle we define an infinite sequence of sets called successor sets.

The nth successor set, of α, s_α^n, is defined by the iterative identity

$$s_\alpha^n = \{x : y\rho x \text{ and } y \in s_\alpha^{n-1}\} \quad n > 1$$

and

$$s_\alpha^1 = s_\alpha$$

example: from the grammar D and its skeleton given above we see:

$$s_A^1 = s_A = \{B,C\}$$

$$s_B^1 = s_B = \emptyset \qquad \text{(the null set)}$$

$$s_C^1 = s_C = \{D\}$$

$$s_D^1 = s_D = \emptyset$$

hence $s_A^2 = \{D\}$

$s_A^3 = \emptyset$

$s_A^n = \emptyset \qquad \forall\, n \geq 3$

$s_B^n = \emptyset \qquad \forall\, n \geq 1$

$s_C^n = \emptyset \qquad \forall\, n \geq 2$

$s_D^n = \emptyset \qquad \forall\, n \geq 1$

Since the skeletons are not in general acyclic (definition in §5.2), we have instances when $\alpha \in s_\alpha^n$ for some $n > 1$. These are precisely the situations where we are thrown into recursive loops. (Trivially, the characterisation of loops by the sets s_α^n gives rise to a simple method of detecting such loops.) In order for us to be able to use the routine SYNSUB successfully we have to select nodes, α, of the skeleton with the property that every element of $s_\alpha (\neq \emptyset)$ ultimately generates either α or $\emptyset$. However we may simplify this condition:

Define the nth reduced successor set, r_α^n, of α, by:

$$r_\alpha^1 = s_\alpha$$

and

$$r_\alpha^n = \{x : y\rho x,\ x \neq \alpha,\ y \in r_\alpha^{n-1},\ n > 1\}$$

then (a) if $r_\alpha^1 = \emptyset$ there is no reduction to be done,

(b) if $n > 1$ st $r_\alpha^n = \emptyset$ then SYNSUB can be applied to α and its outbundle reduced to $\emptyset$.

example: from grammar C we have:

$$r_A^1 = s_A = \{B\}$$

$$r_B^1 = s_B = \{C, D\}$$

$$r_C^1 = s_C = \{A\}$$

$$r_D^1 = s_D = \{A\}$$

$$r_A^2 = \{C,D\}$$

$$r_A^3 = \emptyset$$

$$r_B^2 = \{A\}$$

$$r_B^3 = \emptyset$$

$$r_C^2 = \{B\}$$

$$r_C^3 = \{D\}$$

$$r_C^4 = \{A\}$$

$$r_C^5 = \{B\} \qquad \text{etc.}$$

$$r_D^2 = \{B\}$$

$$r_D^3 = \{C\}$$

$$r_D^4 = \{A\}$$

$$r_D^5 = \{B\} \qquad \text{etc.}$$

Notice that if $\exists\, n > m : r_\alpha^n \neq \emptyset$ and m is the number of nodes in the graph, then $\nexists\, p \in \mathbb{N}$ st.

$$r_\alpha^p = \emptyset$$

Defn: A node, α, of the skeleton such that $n \in \mathbb{N} : n > 1$ and $r_\alpha^n = \emptyset$, is called a central point of the skeleton.

Defn: A node, α, of the skeleton such that $r_\alpha^1 = \emptyset$ is called a loose point of the skeleton.

<u>example</u>: from Grammar E we have:

$r_A^1 = \{B\}$

$r_B^1 = \{C\}$

$r_C^1 = \{B\}$

$r_A^2 = \{C\}$

$r_A^3 = \{B\}$ etc.

$r_B^2 = \emptyset$

$r_C^2 = \emptyset$

Hence the central points of grammars C and E are A, B and B,C respectively, and there are no loose points. After a central point has been processed by 'SYNSUB' it becomes a loose point and need be considered no further by SYNSUB.

<u>Defn</u>: If at some stage in the modification of a grammar (i) there are no central points and (ii) the number of loose points is less than the total number of class nodes in the current state of the skeleton for that grammar; then the grammar is said to be <u>locked</u>.

We shall return to the problem of locked grammars later.

Following the previous theory on skeletons we adjust the modification process to give algorithm AII.

<u>AII</u>:
(i) (left) factorise all definitions.
(ii) Remove all direct left recursion.
(iii) Count loose points in skeleton.

(iv) Locate new central points in skeleton (if non goto stage (vi)).

(v) Remove a central point by 'SYNSUB', update count of loose points and modify skeleton. If there are further located central points goto (v) otherwise goto stage (iv).

(vi) If number of loose points equals current total of nodes in skeleton goto (vii). Otherwise fail (Grammar is locked).

(vii) Exit.

We now consider the modification of some grammars (those given before plus new ones) under AII.

<u>Grammar A</u>:

$$X ::= ab|abc|abcd|ac$$

This grammar has a <u>loose skeleton</u> (i.e. all nodes are loose points) and hence gives the same result under AII as under AI, i.e.

$$\begin{cases} X ::= aX_3 \\ X_1 ::= c|\Lambda \\ X_2 ::= cX_4|\Lambda \\ X_3 ::= bX_2|c \\ X_4 ::= d|\Lambda \end{cases}$$

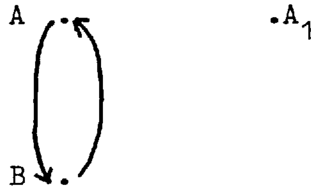
<u>Grammar B</u>:

$$\begin{cases} A ::= Ab|B \\ B ::= Ac|d \end{cases}$$

After stage (ii) of AII this gives

$$\begin{cases} A ::= BA_1 \\ B ::= Ac \mid d \\ A_1 ::= bA_1 \mid \Lambda \end{cases}$$

with skeleton.

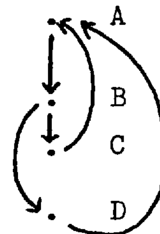


Trivially A_1 is loose and A,B central, hence the result is as before, i.e.

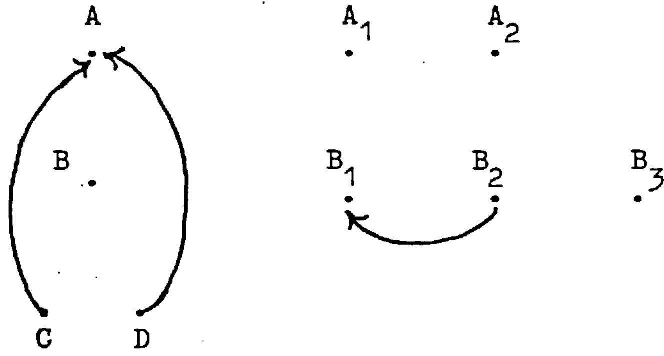
$$\begin{cases} A ::= dA_1A_2 \\ B ::= dB_1 \\ A_1 ::= bA_1 \mid \Lambda \\ A_2 ::= cA_1A_2 \mid \Lambda \\ B_1 ::= bA_1A_2 \mid cB_2 \mid \Lambda \\ B_2 ::= bA_1A_2c \mid c \mid \Lambda \end{cases}$$

<u>Grammar C</u>:

$$\begin{cases} A ::= Ba \mid c \\ B ::= Cb \mid Db \\ C ::= Ae \mid g \\ D ::= Af \end{cases}$$



This grammar has central points A and B. After they have been <u>loosened</u> (i.e. processed by 'synsub' into loose points) the skeleton becomes



Then, A, B, A_1, A_2, B_1, B_3 become loose and C, D, B_2 are central. Hence the order of simplifcation coincides with that of AI (i.e. $A, B, C, D, A_1, A_2, B_1, B_2, B_3, B_4, C_1$) and thus gives the same result.

i.e.

$$\begin{cases} A & ::= gbaA_1A_2 \mid cA_1A_2 \\ B & ::= gbB_2 \mid cA_1A_2B_3 \\ C & ::= gC_1 \mid cA_1A_2e \\ D & ::= gbaA_1A_2f \mid cA_1A_2f \\ A_1 & ::= ebaA_1 \mid \Lambda \\ A_2 & ::= fdaA_1A_2 \mid \Lambda \\ B_1 & ::= aA_1A_2eb \mid \Lambda \\ B_2 & ::= aA_1A_2B_4 \mid \Lambda \\ B_3 & ::= eb \mid fd \\ B_4 & ::= eb \mid fd \\ C_1 & ::= baA_1A_2e \mid \Lambda \end{cases}$$

Grammar D:

$$\begin{cases} A ::= Ba \mid C \\ B ::= b \\ C ::= d \mid Da \\ D ::= b \mid e \end{cases}$$

Trivially B and D are loose and, A and C are central, hence the first central point which we try to loosen is A. This, of course, results in an ambiguity state and halts just as under AI.

Grammar E:

$$\begin{cases} A ::= B \mid a \\ B ::= C \mid b \\ C ::= Bd \end{cases}$$

This grammar could not be modified by AI, however under AII it generates the following

$$\begin{cases} A ::= bB_1 \mid a \\ B ::= bB_1 \\ C ::= dB_1 d \\ B_1 ::= dB_1 \mid \Lambda \end{cases}$$

Grammar F:

$$\begin{cases} C_1 ::= C_2 a \\ C_2 ::= C_3 b \\ C_3 ::= C_4 c \mid C_5 d \\ C_4 ::= C_4 e \mid f \\ C_5 ::= C_2 g \mid C_6 h \\ C_6 ::= C_7 l \\ C_7 ::= C_8 \\ C_8 ::= C_6 m \mid C_9 n \\ C_9 ::= C_{10} p \mid q \\ C_{10} :: \ C_9 t \mid x \end{cases}$$

Under the action of AII this generates:

$$C_1 ::= fC_{11} cbC_{19} a \mid xpC_{12} nlC_{14} hdbC_{19} a \mid qC_{12} nlC_{14} hdbC_{19} a$$

$$C_2 ::= fC_{11} cbC_{19} \mid xpC_{12} nlC_{14} hdbC_{19} \mid qC_{12} nlC_{14} hdbC_{19}$$

$$C_3 ::= fC_{11} cC_{20} \mid xpC_{12} nlC_{14} hdC_{21} \mid qC_{12} nlC_{14} hdC_{22}$$

C_4 ::= fC_{11}

C_5 ::= $fC_{11}cbC_{19}g | xpC_{12}nlC_{14}hC_{23} |$
$qC_{12}nlC_{14}hC_{24}$

C_6 ::= $xpC_{12}nlC_{14} | qC_{12}nlC_{14}$

C_7 ::= $xpC_{12}nC_{15} | qC_{12}nC_{16}$

C_8 ::= $xpC_{12}nC_{17} | qC_{12}nC_{18}$

C_9 ::= $xpC_{12} | qC_{12}$

C_{10} ::= $xC_{13} | qC_{12}t$

C_{11} ::= $eC_{11} | \Lambda$

C_{12} ::= $tpC_{12} | \Lambda$

C_{13} ::= $pC_{12}t | \Lambda$

C_{14} ::= $mlC_{14} | \Lambda$

C_{15} ::= $lC_{14}m | \Lambda$

C_{16} ::= $lC_{14}m | \Lambda$

C_{17} ::= $lC_{14}m | \Lambda$

C_{18} ::= $lC_{14}m | \Lambda$

C_{19} ::= $gbdC_{19} | \Lambda$

C_{20} ::= $bC_{19}gd | \Lambda$

C_{21} ::= $bC_{19}gd | \Lambda$

C_{22} ::= $bC_{19}gd | \Lambda$

C_{23} ::= $dbC_{19}g | \Lambda$

C_{24} ::= $dbC_{19}g | \Lambda$

The above set of classes can obviously be reduced since some of the productions are not used and there are many duplicates. The removal of these will be considered later. Next we consider Grammar G.

Grammar G:

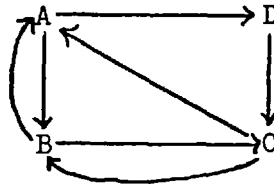
$$A ::= Ba|Db$$

$$B ::= Cc|Ad$$

$$C ::= Ae|Bf$$

$$D ::= Cg|h$$

The skeleton is



This is locked since,

$$r_A^4 = \{C\}$$

$$r_B^4 = \{A,C\}$$

$$r_C^4 = \{A,B,D\}$$

$$r_D^4 = \{A,B,C\}$$

Defn: Given a locked grammar in which the nodes $\alpha_1,\ldots,\alpha_n$ are locked.

i.e. $r_{\alpha_i}^n \neq \emptyset \; \forall n$,

then an α_i st. $\alpha_i \in s_{\alpha_i}^n$ for some i is a nest point.

Defn: Given a nest point α_i, the nest generated by α_i is $\alpha_i \bigcup_{n=1}^{\infty} r_{\alpha_i}^n$ and the order of this set is called the size of the nest.

Trivially, grammar G generates a nest of size 4, i.e. the whole skeleton is a nest.

To unlock a nest we use a transformation given (in a slightly different form) in [46]. Before giving the general form of the transformation we demonstrate its use in the case of grammar G.

Firstly, we write the productions in a tabular form:

	(A)	(B)	(C)	(D)	
A ::=		Ba|		Db	
B ::=	Ad|		Cc		
C ::=	Ae|	Bf			
D ::=			Cg|		h

Secondly, from this table we create two arrays, Y and Z.

$Y_{i,j}$	j →	A	B	C	D
i ↓					
A			d	e	
B		a		f	
C			c		g
D		b			
Z_j					h

Note: Here no entry is not equivalent to an entry 'Λ' and, $Y_{i,j}$ and Z_j are not classes but arbitrary strings. In what follows $X_{i,j}$ represent newly created classes which could later be renamed to coincide with earlier notation.

The original class definitions are now transformed into:

A ::= hX_{DA}

B ::= hX_{DB}

C ::= hX_{DC}

D ::= hX_{DD}

where:-

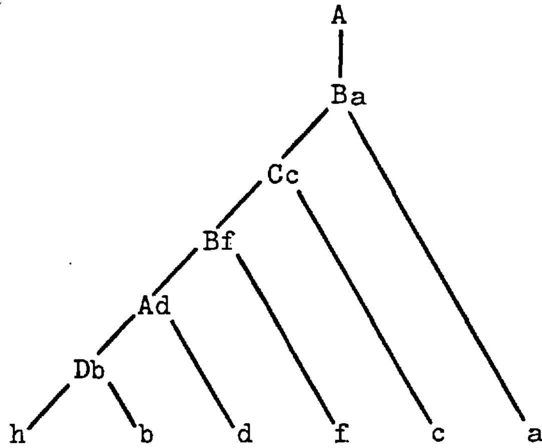
		(A)	(B)	(C)	(D)	
X_{AA}	::=		$dX_{BA} \mid$	$eX_{CA} \mid$		Λ
X_{AB}	::=		$dX_{BB} \mid$	eX_{CB}		
X_{AC}	::=		$dX_{BC} \mid$	eX_{CC}		
X_{AD}	::=		$dX_{BD} \mid$	eX_{CD}		
X_{BA}	::=	$aX_{AA} \mid$		fX_{CA}		
X_{BB}	::=	$aX_{AB} \mid$		$fX_{CB} \mid$		Λ
X_{BC}	::=	$aX_{AC} \mid$		fX_{CC}		
X_{BD}	::=	$aX_{AD} \mid$		fX_{CD}		
X_{CA}	::=		$cX_{BA} \mid$		gX_{DA}	
X_{CB}	::=		$cX_{BB} \mid$		gX_{DB}	
X_{CC}	::=		$cX_{BC} \mid$		$gX_{DC} \mid$	Λ
X_{CD}	::=		$cX_{BD} \mid$		gX_{DD}	
X_{DA}	::=	bX_{AA}				
X_{DB}	::=	bX_{AB}				
X_{DC}	::=	bX_{AC}				
X_{DD}	::=	$bX_{AD} \mid$				Λ

We note that if the Grammar G is fully defined by the given set of classes (i.e. A,B,C,D) then 12 of the X classes and B,C,D are redundant, hence we are left with,

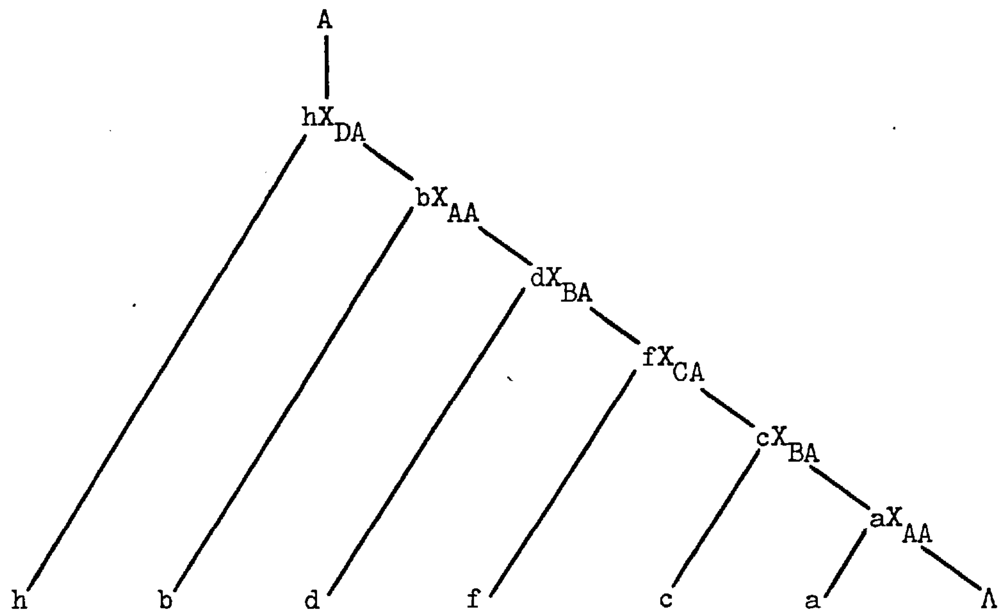
A,

X_{AA}, X_{BA}, X_{CA} and X_{DA}.

The grammar G is extreme in that it only gives rise to right-branching parse trees. An example of the parse by the original grammar and its modified form, illustrates the power of the transformation.



Parse of 'hbdfca' under original grammar.



Parse of 'hbdfca' under modified grammar.

Notice, if we had had (i) $A ::= B|Db$, then $Y_{BA} = \Lambda$ and $X_{BA} ::= X_{AA}|fX_{CA}$ etc., or (ii) $D ::= Cg|h|j$, then $Z_D = h|j$ and the resultant $A ::= hX_{DA}|jX_{DA}$ etc.,

or (iii) $A ::= BC|Db$, then the parses could involve X-productions which have their second co-ordinate not equal to A, e.g. X_{BC}, thus the necessity, in general, for constructing all the X-productions.

Formally, we may describe the transformation 'unlock' as follows:

Given a nest of locked classes,

$$C = \{C_i : i \in P \subset \mathbb{N}\}$$

each of which is defined by:

$$C_i ::= C_j\ Y_{j,i} | Z_i$$

or

$$C_i ::= C_j\ Y_{j,i}$$

where (i) C_i is left factored

(ii) j ranges over all classes, $C_j : j \in P$, and C_j leads an option of C_i

(iii) $Y_{j,i}$ is a string which may be 'Λ' and does not include '|'

(iv) Z_i, if it exists, is a string which may be 'Λ', or contain '|'.

The result of the transformation is the set C'.

$$C' = \{C_i' : C_i \in C\} \cup \{X_{i,j} : C_i, C_j \in C\}$$

where:

$C_i' ::= Z_j X_{ji}$ with j ranging over all Z_j which exist (i.e. are strings, including Λ; see previous example).

$X_{rr} ::= Y_{rj} X_{jr} | \Lambda$ with j ranging over all Y_{rj} which exist.

$X_{rs} ::= Y_{rj} X_{js}$ with j ranging over all Y_{rj} which exist, and $r \neq s$.

Apart from the complex manipulation of indices which is involved (this is trivially similar to the summation conventions of tensor algebra), we must take care to distinguish between the

non-occurrence of strings and the occurrence of 'Λ'

e.g.: $A ::= B|Df$

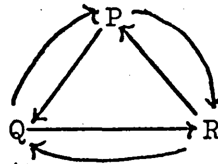
$$\Rightarrow \quad Y_{AA} = \emptyset \mp \Lambda$$
$$Y_{BA} = \Lambda$$
$$Y_{CA} = \emptyset \mp \Lambda$$
$$Y_{DA} = f$$

Before making the required modification to our algorithm we consider the following (wildly ambiguous) grammar:

<u>Grammar H</u>:

$$P \rightarrow Q|R$$
$$Q \rightarrow P|R$$
$$R \rightarrow P|Q|x$$

The skeleton for this grammar is:



Obviously it is locked, with a nest of size 3.

If we apply the previous modifications to unlock P,Q,R we have:

$$P ::= xX_{RP}$$
$$Q ::= xX_{RQ}$$
$$R ::= xX_{RR}$$

but, $X_{PP} ::= X_{QP} | X_{RP} | \Lambda$

etc.

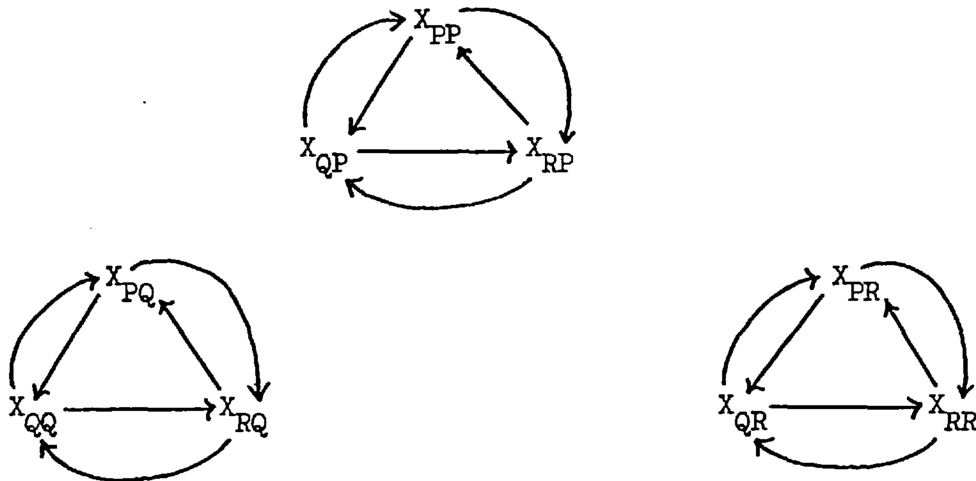
$X_{QP} ::= X_{PP} | X_{RP}$

etc.

$X_{RP} ::= X_{PP} | X_{QP}$

Hence we have 3 secondary locks (i.e. productions created to unlock a set of productions are also locked).

The skeletons are:



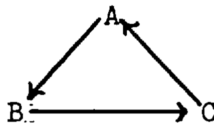
As mentioned before, the grammar is ambiguous. This is seen from the parses:

P	P
↓	↓
R	Q
↓	↓
x	R
	↓
	x

Moreover, we have:-

Theorem: Any secondary lock implies ambiguity.

(This is not a bi-implication since we may have ambiguity in a set of productions which does not constitute a lock. e.g.



but this would be detected elsewhere since it would generate '$\Lambda|\Lambda$' after application of the back-substitution and factorization routines, or give $A \overset{*}{\rightarrow} A$, (i.e. A derives A)).

The proof of the ambiguity is as follows:

Since the original grammar is locked we apply the transformation to remove the lock. In so doing it creates a set of new classes, $N = \{N_i\}$ with the property that each N_i is of the form.

$$N_i ::= \alpha_{j_1} N_{j_1} \mid \dots \mid \alpha_{j_{n_i}} N_{j_{n_i}}$$

or

$$N_i ::= \alpha_{j_1} N_{j_1} \mid \dots \mid \alpha_{j_{n_i}} N_{j_{n_i}} \mid \Lambda$$

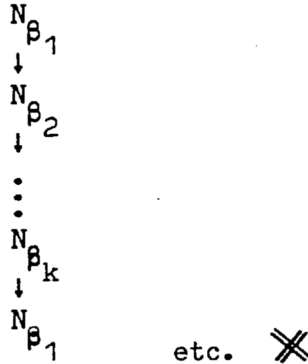
Now if the set N locks there must be a cycle in the skeleton (more precisely in the restriction of the skeleton to the set N) and hence $\alpha_{\gamma_i} = \Lambda$ for a suitable set of γ_i's.

i.e. $N_{\beta_1} \rightarrow N_{\beta_2} \rightarrow N_{\beta_3} \dots N_{\beta_k}$ (with N_{β_k} returning to N_{β_1})

with

$$N_{\beta_i} \in N, \quad 1 \le i \le k$$

hence $\exists$ option in the definitions of N_{β_i} such that we may have a parse:



Removal of locks

Given a set of locked nodes we must pick out and remove the minimal nest. To do this we calculate the size of the nest of each nest point and select one, α_j, with the least size (this is not unique). If size $(\alpha_j) = p$, then

$$\text{nest}(\alpha_j) = \alpha_j \bigcup_{n=1}^{p} r^n_{\alpha_j}$$

This nest is removed as described. If nest $(\alpha_j) \nsubseteq L$, where L is the set of locked nodes, then we repeat the operation on the set $L \setminus \text{nest}(\alpha_j)$.

We now give, as promised earlier, the definition of an extended factorisation routine. This is followed by the final form (AIII) of the LFF subalgorithm.

Extended factorisation: in an attempt to prevent needless extension to the set of classes by creating new elements whose definition is in some way equivalent to an already existing class we modify the factorisation routine as follows:

After a factorisation which results in the creation of a new class, C_j, check if this is a repetition of a previous class, subject to any combination of the following:

(i) if $\exists\, C_i ::= \alpha_1|\alpha_2| \dots |\alpha_n$

and

$C_j ::= \alpha_{f(1)}|\alpha_{f(2)}| \dots |\alpha_{f(n)}$ $\quad i \neq j$

where f is a bijection on $\{1,2,\dots,n\}$ then $C_j \equiv C_i$; replace the occurrence of the name C_j by C_i and delete the definition of C_j.

(ii) if $\exists C_i = f(C_i)$

and $C_j = f(C_j)$ $\quad i \neq j$, strings μ_ℓ

with $f(x) = \mu_1 \, x \, \mu_2 \, x \, \mu_3 \dots \mu_{n-1} \, x \, \mu_n$

and $\quad x \notin \mu_\ell \; (1 \leq \ell \leq n)$

then $C_j \equiv C_i$; replace the occurrence of the name C_j by C_i and delete the definition of C_j.

With this more powerful version of the factorisation routine we give the final version of the LFF algorithm:

AIII (i) (left) factorise all definitions.

(ii) Remove all direct left recursion.

(iii) Count loose points in skeleton.

(iv) Locate central points in skeleton. If none go to (vi).

(v) Remove central points, update count of loose points, modify skeleton and goto (iv).

(vi) If the number of nodes in skeleton equals number of loose points then exit, else go to (vii).

(vii) If 'unlock' has been entered already then check if current nest created by a previous 'unlock'; if so then halt (ambiguous grammar) otherwise unlock, update count of loose points, modify skeleton and goto (iv).

Since, via the 'unlock' transformation, this is a trivial extension of AII we give no examples of its usage.

2.1.2.2 Equivalence of LFF and GNF

Given a context-free language L with $\Lambda \notin L$, defined by a cfg, G, then G is strongly equivalent to G^1 (i.e. they both define L) where all productions of G^1 are in Greibach Normal Form, i.e.

$$Z \rightarrow a\ Y_1 \ldots Y_m \quad \text{where } Z,\ Y_i\ (1 \le i \le m)\ (0 \le m)$$

are classes and a is a terminal symbol.

This is proved in [51] and trivially asserts the equivalence of LFF and GNF for any language $L(\Lambda \notin L)$ for which a grammar exists such that modification to LFF is possible.

A sufficient (but not necessary) condition for the formation of an LFF is that the grammar should be unambiguous. Hence:

Theorem:

Given a cfl $L : \Lambda \notin L$ and an associated grammar G which is unambiguous then $\exists$ equivalent grammars G^1, G^{1*} and G^2,G^{2*} in LFF and GNF respectively which may be derived by translations.

$$G \Rightarrow G^1 \Rightarrow G^2$$

and

$$G \Rightarrow G^{2*} \Rightarrow G^{1*}$$

Proof: (a) $G \Rightarrow G^1$

trivial by application of AIII

(b) $G^1 \Rightarrow G^2$

G^1 is in LFF. If we remove any redundant classes (see §2.1.2.4) to give $\bar{G}$ then the grammars G^1 and $\bar{G}$ are equivalent in that they generate the same language, L.

Given any class X (not the root) such that $X \rightarrow \Lambda$, then there is a (non-looping) chain, C_i (see linkage graph in §2.1.2.4), such that

$$C_1 \xrightarrow{1} C_2 \xrightarrow{1} C_3 \ldots \xrightarrow{1} C_n = X$$

with $C_i \xrightarrow{1} C_{i+1}$ if $\exists$ a production $C_i \rightarrow \alpha\, C_{i+1}\, \beta$, α,β are strings over classes and terminal symbols ($\cup\, \Lambda$), and $C_1 \not\rightarrow \Lambda$ but $C_i \xrightarrow{*} \Lambda$ $(i \neq 1)$.

For any such class, X (i) back substitute as dictated by the chain, and (ii) back substitute for all occurrences of X in any production $(\neq X)$.

We then have the form,

$$C \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$$

with $\alpha_i \neq \Lambda$ for all classes C. Moreover each α_i is of the form,

$$t\beta_1, \ldots, \beta_p \quad : \quad t \in T, \quad \beta_i \in T \cup C$$

where T,C are the disjoint sets of terminals and classes.

Now for all $\beta_i \in T$ substitute a class name C_{β_i} and define

$$C_{\beta_i} \rightarrow \beta_i$$

Then the grammar is in GNF i.e. we have the required G^2.

(c) $G \Rightarrow G^{2*}$

∴ proved in [51].

(d) $G^{2*} \Rightarrow G^{1*}$

by application of AIII.

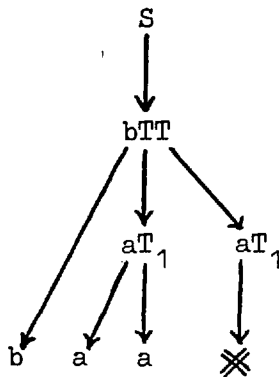
2.1.2.3 Investigation of NBU Conditions

To illustrate the sort of back-up problems which arise we consider some offending grammars.

(1) $\begin{cases} S \rightarrow bTT \\ T \rightarrow a|aa \end{cases}$

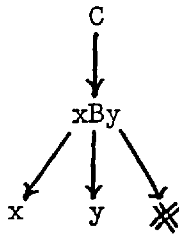
$$AIII \Rightarrow \begin{cases} S \rightarrow bTT \\ T \rightarrow aT_1 \\ T_1 \rightarrow a|\Lambda \end{cases}$$

an attempt to parse 'baa' from left to right yields



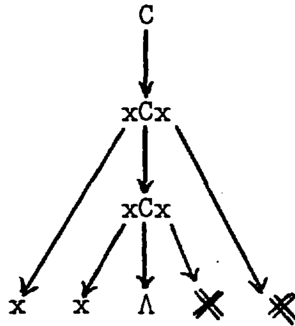
(2) $\begin{cases} C \rightarrow xBy \\ B \rightarrow y|\Lambda \end{cases}$ (in LFF)

an attempt to parse 'xy' yields



(3) $C \rightarrow xCx|\Lambda$ (in LFF)

an attempt to parse 'xx' yields



No-backup (NBU)

We first give some definitions. In what follows we shall use the symbols T and C to denote the set of terminal symbols and the set of classes respectively, with

$$T \cap C = \emptyset\ , \quad \Lambda \notin T \cup C$$

If by use of a finite sequence of re-writing rules, we can replace a class X by a string α over $T \cup C \cup \{\Lambda\}$ then we say that X derives α and we write

$$X \overset{*}{\rightarrow} \alpha$$

Trivially, X is Λ-producing if $X \overset{*}{\rightarrow} \Lambda$ and the negation of the derivation property is denoted by $\overset{*}{\not\rightarrow}$.

If we have

$$X \rightarrow Y_1 | Y_2 | \ldots | Y_n \qquad n \geq 1$$

with Y_i $(1 \leq i < n)$ of the form $t_i\ \alpha_i$ and $Y_n = \Lambda$ or $t_n\ \alpha_n$ where $t_i \in T$ and α_i is an arbitrary string over $T \cup C$, then we construct two relations (for each such X) on $C \cup T \cup \{\Lambda\}$ as follows:

first (X) = $\{t_i\ :\ 1 \leq i < n\} \cup g_n\ (X)$

where

$$g_n(X) = \begin{cases} t_n & \text{if } Y_n = t_n\ \alpha_n \\ \emptyset & \text{if } Y_n = \Lambda \end{cases}$$

follow (X): If, given a grammar G in which we have the non-terminal

(i.e. class) X, and there exist occurrences of X such that

$$X_i \rightarrow \alpha_i \, X \, \beta_i \; : \; 1 \le i \le n$$

with n finite and $X_i \in C$

where $\alpha_i = t_i \, \gamma_i$, and β_i, γ_i are arbitrary strings

(in particular we may have $X \in \beta_i$, γ_i or $\beta_i, \gamma_i = \Lambda$)

There are 3 distinct possibilities for β_i

(i) $\beta_i = \Lambda$

(ii) $\beta_i = t_i' \, \delta_i \qquad t_i' \in T$

(iii) $\beta_i = X_i' \, \delta_i \qquad X_i' \in C$

Now define h_i on X such that:

$$h_i(X) = \begin{cases} \text{follow } (X_i) & \text{if } \beta_i = \Lambda \text{ and } X_i \neq X \\ \emptyset & \text{if } \beta_i = \Lambda \text{ and } X_i = X \end{cases}$$

$$h_i(X) = t_i' \qquad \text{if } \beta_i = t_i' \, \delta_i$$

and $$h_i(X) = \text{first } (X_i') \qquad \text{if } \beta_i = X_i' \, \delta_i$$

then $$\text{follow } (X) = \bigcup_{i=1}^{m} h_i(X)$$

We now give a well-known set of sufficient conditions for an unambiguous grammar to satisfy the NBU condition [59].

Conditions 1: (a) no left-recursion in C,

(b) for each class, the sets of left-most terminal symbols derived from each option must be disjoint,

(c) if $X \rightarrow Y_1 | \ldots | Y_n$

and $X \overset{*}{\rightarrow} \Lambda$

then $Y_i \overset{*}{\not\rightarrow} \Lambda \qquad (1 \le i < n)$

i.e. $X \overset{*}{\rightarrow} \Lambda \Rightarrow Y_n \overset{*}{\rightarrow} \Lambda$

(d) if $X \rightarrow Y_1 | \ldots | Y_n$

and $Y_n \overset{*}{\rightarrow} \Lambda$

then

$$\text{first}(X) \cap \text{follow}(X) = \emptyset$$

After phase 1 of our modifications (i.e. when the grammar is in left-factored form) this set of conditions reduces to:

<u>Condition 2</u>: if $X : X \rightarrow \Lambda$

then $\text{first}(X) \cap \text{follow}(X) = \emptyset$

We examine how this condition can be violated and give some identities which attempt to remove the violation.

<u>Direct violation</u>

(a) $X \rightarrow x\alpha Xx\beta \mid \gamma \mid \Lambda$ maybe $| \in \gamma$

(b) $X \rightarrow \alpha Xx\beta \mid x\gamma \mid \delta \mid \Lambda$ maybe $| \in \delta$

(a1) $X \rightarrow x^n X x^m \mid \Lambda$ $m,n \geq 1$

$\equiv X \rightarrow x^{n+m} X \mid \Lambda$

(a2) $X \rightarrow xXx\alpha \mid \Lambda$ $\alpha \neq x^m,\ m \geq 0$

$$\equiv \begin{cases} X \rightarrow xx\ Y\alpha \mid \Lambda \\ Y \rightarrow xY\alpha x \mid \Lambda \end{cases}$$

(a3) $X \rightarrow x\gamma Xx \mid \Lambda$ $\gamma \neq x^n,\ n \geq 0$

$$\equiv \begin{cases} X \rightarrow x\gamma xY \mid \Lambda \\ Y \rightarrow \gamma xYx \mid \Lambda \end{cases}$$

(a4) $X \rightarrow x\gamma Xx\alpha \mid \Lambda$

$\equiv X \rightarrow x\gamma x\alpha X \mid \Lambda$ if $\alpha = \gamma$

$$\equiv \begin{cases} X \rightarrow x\gamma xY\alpha \mid \Lambda \\ Y \rightarrow \gamma xY\alpha x \mid \Lambda \end{cases}$$

(a5) $X \to x\alpha Xx\beta \mid \gamma \mid \Lambda$

$$\equiv \begin{cases} X \to x\alpha Y\beta \mid \gamma \mid \Lambda \\ Y \to xZ \mid \gamma x \\ Z \to \alpha Y\beta x \mid \Lambda \end{cases}$$

(b1) $X \to \alpha Xx\beta \mid x\gamma \mid \delta \mid \Lambda$

$$\equiv \begin{cases} X \to \alpha Y\beta \mid x\gamma \mid \delta \mid \Lambda \\ Y \to \alpha Y\beta x \mid \delta x \mid xZ \\ Z \to \gamma x \mid \Lambda \end{cases}$$

Indirect violation

$X \to x\alpha \mid \beta \mid \Lambda$ (maybe $| \in \beta$)

such that $x \in$ first(X) $\cap$ follow(X)

case (i): $\exists Y : Y \to \gamma Xx\delta$ $(Y \neq X)$

solution; we back-substitute X into Y and factorise;

e.g.
$$\begin{cases} X \to xa \mid \Lambda \\ Y \to bXxc \end{cases}$$

becomes $Y \to bxaxc \mid bxc$

hence
$$\begin{cases} Y \to bxY_1 \\ Y_1 \to axc \mid c \end{cases}$$

case (ii): $\exists$ a sequence $C_1, \ldots, C_n$ in C such that

$$\begin{aligned} &\exists C_1 \to \alpha_1 X , \\ &C_2 \to \alpha_2 C_1 , \\ &\vdots \\ &C_{n-1} \to \alpha_{n-1} C_{n-2} , \\ &C_n \to \alpha_n C_{n-1} x \beta_n \end{aligned}$$

solution; if all $C_1, \ldots, C_n$ are different then back-substitute for

X in C_1, after α_1

C_1 in C_2, after α_2

⋮

C_{n-1} in C_n, after α_n, and factorise:

If all the $C_1, \ldots, C_n$ are not disjoint, then back-substitute (as above) those which are, until we reach one class for the second time. This can happen in two ways, either

(a) when the embedded class is reached, or

(b) if $\exists\, i,j,k$ st

$$C_i \to \alpha_j\, C_j \mid \alpha_k\, C_k\, .$$

(a) Before attempting formalization, we give 3 examples:

(i)
$$\begin{cases} X \to \delta X x \gamma \mid cY \\ Y \to x\alpha \mid \beta \mid \Lambda \end{cases}$$

$$\equiv \begin{cases} X \to \delta X'\gamma \mid cY \\ X' \to \delta X'\gamma x \mid cY' \\ Y \to x\alpha \mid \beta \mid \Lambda \\ Y' \to xY'' \mid \beta x \\ Y'' \to \alpha x \mid \Lambda \end{cases}$$

(ii)
$$\begin{cases} X \to \alpha X \beta \mid \delta Y \\ Y \to \beta \mid \Lambda \end{cases}$$

$$\equiv \begin{cases} X \to \alpha X' Y \mid \delta Y \\ X' \to \alpha X'\beta \mid \delta\beta \\ Y \to \beta \mid \Lambda \end{cases}$$

(iii)
$$\begin{cases} X \to \alpha X \beta\gamma \mid \delta Y \\ Y \to \beta \mid \Lambda \end{cases}$$

$$\equiv \begin{cases} X \to \alpha X'\gamma \mid \delta Y \\ X' \to \alpha X'\gamma\beta \mid \delta Y\beta \\ Y \to \beta \mid \Lambda \end{cases}$$

We note that (iii) needs further reduction (by the case (i) rule) to:

$$\begin{cases} X \to \alpha X'\gamma \mid \delta Y \\ X' \to \alpha X'\gamma\beta \mid \delta\beta Y \\ Y \to \beta \mid \Lambda \end{cases}$$

- that (i) needs further reduction if

$$\left.\begin{aligned} \alpha &= t\mu \\ \gamma &= t\mu \end{aligned}\right\} \qquad \text{for some } t \in T ,$$

and, that (i) cannot succeed if

$$\left.\begin{aligned} \alpha &= x^n \\ \gamma &= x^m \end{aligned}\right\} \qquad \text{for some } n,m \geq 1$$

The following reduction may be extracted:

(c1)

$$\begin{cases} X \to \delta X x\gamma \mid \mu Y \mid \eta \\ Y \to x\alpha \mid \beta \mid \Lambda \end{cases}$$

$$\equiv \begin{cases} X \to \delta X'\gamma \mid \mu Y \mid \eta \\ X' \to \delta X'\gamma x \mid \mu Y x \mid \eta x \\ Y \to x\alpha \mid \beta \mid \Lambda \end{cases}$$

Notice that we have $X' \to \mu Yx$ and $Y \to x\alpha \mid \Lambda$ so more reduction needs to be done.

(b) In this case no problems arise and we can proceed as before

example:

$$G \equiv \begin{cases} D \to fBd \mid y \\ C \to x \mid eB \\ B \to fC \mid cA \\ A \to d \mid \Lambda \end{cases}$$

(A in B) $B \rightarrow fC|cd|c$

(B in C) $C \rightarrow x|efC|ecd|ec$

(C in B) $B \rightarrow fx|fefC|fecd|fec|cd|c$

(B in D) $D \rightarrow ffxd|$

$ffefCd|$

$ffecdd|$

$ffecd|$

$fcdd|$

$fcd|$

y

on factorising this gives:

$$G_1 \equiv \begin{cases} D \rightarrow fD_1|y \\ D_1 \rightarrow fD_2|cD_3 \\ D_2 \rightarrow xd|eD_4 \\ D_3 \rightarrow dD_5 \\ D_4 \rightarrow fCd|cdD_5 \\ D_5 \rightarrow d|\Lambda \\ C \rightarrow x|eC_1 \\ C_1 \rightarrow fC|cD_5 \end{cases}$$

As stated, no problems arise in the above procedure, however no advantage is achieved by cutting loops. If, in the above, we replaced A in B then B in D we would have:

$$\begin{aligned} D &\rightarrow fD'|y \\ D' &\rightarrow fCd|cdA \\ C &\rightarrow x|eB \\ B &\rightarrow fC|cA \\ A &\rightarrow d|\Lambda \end{aligned}$$

But d $\in$ first(A) $\cap$ follow(A) since

$$\{d\} \subset \text{follow}(C) \subset \text{follow}(B) \subset \text{follow}(A)$$

and hence we must perform another chain of back-substitutions.

i.e. (A in B), (B in C), (C in D)

After factorization this gives:

$$G_2 \equiv \begin{cases} D & \rightarrow fD'|y \\ D' & \rightarrow fD''|cdA \\ D'' & \rightarrow xd|eD''' \\ D''' & \rightarrow fCd|cdA \\ C & \rightarrow x|eC' \\ C' & \rightarrow fC|ecA \\ A & \rightarrow d|\Lambda \end{cases}$$

If we remove the trivial class D_3 from G_1 then $G_1 \simeq G_2$ by the isomorphism.

$$\begin{array}{lll} D & \leftrightarrow & D \\ D' & \leftrightarrow & D_1 \\ D'' & \leftrightarrow & D_2 \\ D''' & \leftrightarrow & D_4 \\ C & \leftrightarrow & C \\ C' & \leftrightarrow & C_1 \\ A & \leftrightarrow & D_5 \end{array}$$

This suggests that the order of application of reduction sequences is unimportant but the amount of work involved may vary considerably. The problem of ordering the reductions and question of convergence of these reductions sequences is considered next.

<u>Representation of follow and first</u>: We consider some examples. These are given together with related diagrams, which will be formally defined later.

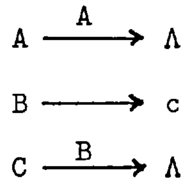
<u>e.g. 1</u>:

$A \to aA|bBc$

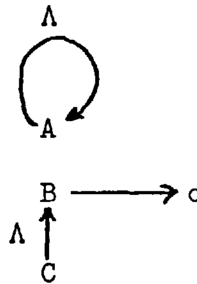
$B \to dC|x$

$C \to e \mid \Lambda$

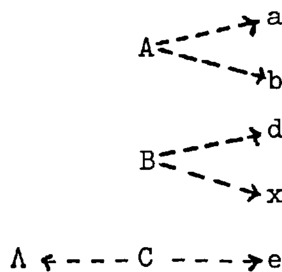
'follow' diagram F_0



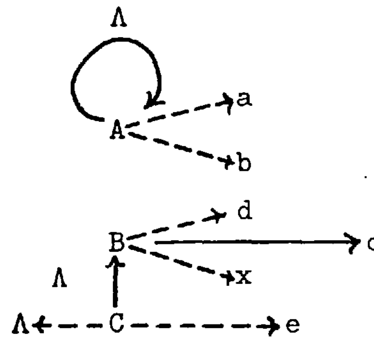
F_0^*, the closure of F_0



'first' digraph, F_1



'back-up diagram', $F_0^* \sqcup F_1$

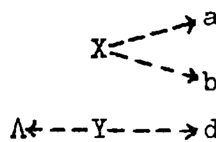


In this example there are no BU problems.

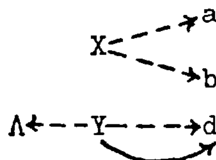
e.g. 2: $\begin{cases} X \to aYd \mid b \\ Y \to d \mid \Lambda \end{cases}$

$F_0 = F_0^*$:

F_1:



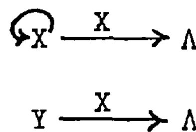
$F_0^* \sqcup F_1$:



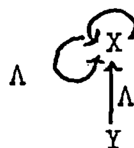
Here we do have problems but they are soluble (i.e. we can modify the grammar so that we satisfy the NBU condition).

e.g. 3: $\begin{cases} X \to aXX \mid bY \\ Y \to a \mid \Lambda \end{cases}$

F_0:



F_0^*:



Here we have violation of the NBU conditions which are not removable by our methods.

e.g. 4: $\begin{cases} X \rightarrow xxXxx \mid yY \\ Y \rightarrow x \mid \Lambda \end{cases}$

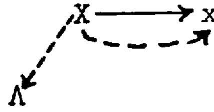
Here we have a violation which can be circumvented by our modifications.

e.g. 5: $X \rightarrow xXxa \mid \Lambda$

- a removable violation.

e.g. 6: $X \to x^n X x^m \mid \Lambda$

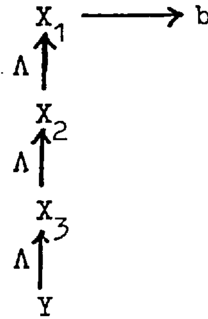
$F_0^* \sqcup F_1$:



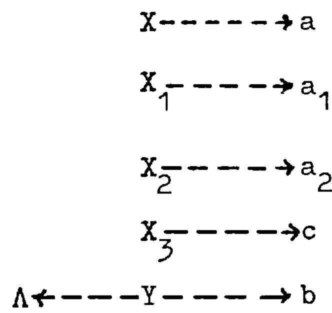
- another removable violation

e.g. 7:
$$\begin{cases} X \to aX_1b \\ X_1 \to a_1X_2 \\ X_2 \to a_2X_3 \\ X_3 \to cY \\ Y \to b \mid \Lambda \end{cases}$$

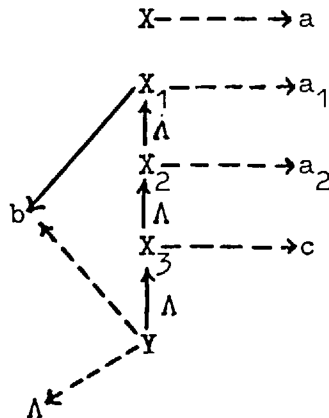
F_0^* :



F_1:



$F_0^* \sqcup F_1$:



- this grammar is also in violation, but the violation is removable.

Formalisation of first and follow digraphs

Referring to the defn. of follow, we define a function f_0 over C such that

$$\Lambda \in f_0(X) \quad \text{if} \quad \beta_i = \Lambda$$

$$t_i' \in f_0(X) \quad \text{if} \quad \beta_i = t_i' \; \delta_i, \; t_i' \in T$$

$$X_i' \in f_0(X) \quad \text{if} \quad \beta_i = X_i' \; \delta_i, \; X_i' \in C$$

moreover if $X_i \to \alpha_i \, X \, \beta_i$ and $\beta_i = \Lambda$, then label the relation, thus

$$X \xrightarrow{X_i} \Lambda$$

call the resultant diagram† F_0.

Now form the closure, F_0^*, of F_0 by the process:

If $\exists X : X \xrightarrow{Y} \Lambda$ (and $Y \neq \Lambda$), then delete the link

$X \xrightarrow{Y} \Lambda$ and construct a new (Λ-labelled) link

$X \xrightarrow{\Lambda} Y$

Similarly, referring to the definition of first, we define a function f_1 over C such that:

$$t_i \in f_1(X) \quad \text{if} \quad Y_i = y_i \alpha_i$$

$$\Lambda \in f_1(X) \quad \text{if} \quad Y_n = \Lambda$$

The resultant digraph† we call F_1 and represent it by broken lines.

Now form the disjoint union of F_0^* and F_1, i.e. $F_0^* \sqcup F_1$ with the natural equivalence relation between corresponding elements in the common base set $T \cup C \cup \{\Lambda\}$ (i.e. the sets of points); further, let $F(\bar{f}_0, h)$ be the free monoid on $F_0^* \sqcup F_1$ generated by $\bar{f}_0$ and h, where:

---

† Note: F_1 is a digraph but F_0 may not be.

$$h : X \to \begin{cases} f_0(X) & \text{if } \Lambda \in f_1(X) \\ \text{otherwise undefined} \end{cases}$$

and $\bar{f}_0$ is a Λ-labelled f_o-link.

<u>Defn</u>: Any element of $F(\bar{f}_0, h)$ is called a <u>following sequence</u>.

Then, problems arise iff

$$\exists\ T_i,\ C_j : \{\Lambda,\ T_i\} \subseteq f_1(C_j)$$

and either (i) $T_i \in f_0 h_1(C_j)$

or (ii) $T_i \in f_1 f_0 h_2(C_j)$

where h_1, h_2 are following sequences.

The above conditions may be interpreted as follows:

Case (i): We have a following sequence, α,

(i.e. a chain of; $X_i \xrightarrow{\Lambda} Y_i$

- and $X_k \longrightarrow Y_k$, with $X_k \dashrightarrow \Lambda$)

- ended by $X \xrightarrow{f_0} T_i \in T$

- and starting at C_j

Case (ii): This is a following sequence, α, from C_j to X, where

$Y \in f_0(X)$

and $T_i \in f_1(Y)$

(i.e. C_j is followed by Y which begins with T_i)

<u>Note</u>: If the grammar is not in LFF then we may have,

$T_i \in f_1{}^n(Y)$ $(n > 1)$

Defn: The diagram, $F_0^* \sqcup F_1$ together with its associated operations $F(\bar{f}_0, f_1)$ is called the Backup Diagram.

The identities given earlier, if applicable, will each remove one of these link sequences (i.e. an element of $F(\bar{f}_0, f_1)$). If the grammar has an (implied) infinity of such links, as in example 3, then the transformations will attempt to create a countable infinity of subsidiary classes to circumvent the violation; in this case, however, we can create a bounded approximation to the grammar which in some cases may be sufficient. An outline of how this may be done (on a simple example) is given in Appendix 9 of [25].

If the grammar is in violation and has a finite number of link sequences which cause the violation then these should be removed in (some) decreasing order of length - this may not be a unique ordering, and the removal of a linkage sequence may cause other links to change.

Removal of BU violation from a grammar in LFF may be algorithmatized thus:

AIV: (i) Form the backup diagram and check for unbounded sets of differing link-sequences between C_j and T_i, as described above - if any found then halt or make a bounded approximation.

(ii) Remove any superfluous links by applying the first identity (of the set a1 - a5, b1, c1) which is applicable after chains have been removed. If any new links have been created go to (i), else halt.

2.1.2.4 Miscellaneous checks and reductions

(a) Removal of trivial classes; any class where definition consists of a single option is called trivial. Any such class (other than the parse root) can easily be removed by back-substitution of the definition for every occurrence of the class name in other productions provided that this class is not directly self-embedding.
e.g.

$$C \not\to \alpha\, C\, \beta \quad \text{for any} \quad \alpha,\beta \not\ni |$$

(b) Removal of redundant classes: any (non-root) class which is never accessed (directly or indirectly) from the root class is redundant and cannot occur in any parse.

If a class is not redundant it is used and we abstract the property of being used in the following way:-

For each node, α, in C we derive a set, ℓ_α, of links from α.

i.e. $\ell_\alpha = \{\beta : \beta \in C \text{ and } \beta \text{ occurs explicitly in the definition of } \alpha\}$

e.g. if $C_1 ::= C_2a|bC_3|cC_4dC_1|C_3$

then $\ell_{C_1} = \{C_1, C_2, C_3, C_4\}$

If we order the classes in such a way that the root class is number 1, and declare an array 'used' of length $|C|$ with this same order, then:-

$\underline{A\bar{V}}$ (i) set used (1) = 1, used (i) = 0, $(i \neq 1)$

form ℓ_α for all $\alpha \in C$

set $\ell = \ell_{root} = \ell_{C_1}$

(ii) if $\ell = \emptyset$ then exit

(iii) if $C_i \in \ell$, then -

if used (i) = 1 goto (iv)

otherwise, set used (i) = 1

and $\ell = (\ell \setminus C_i) \cup \ell_{C_i}$

goto (ii)

(iv) set $\ell = \ell \setminus C_i$, goto (ii)

Since $|C|$ is finite for any usable BNF grammar and each class definition is finite, then all ℓ_α are constructable and $A\overline{V}$ must terminate. After termination, the set of classes, C_i : used (i) =1 are clearly the only ones used by the grammar.

We may call the diagram resulting from the relation ℓ , the <u>usage graph</u>:

e.g. $$\begin{cases} A \rightarrow aBc|d \\ B \rightarrow cA \\ C \rightarrow x \end{cases}$$

yields the usage graph:

C

i.e. C is redundant.

An alternative characterization of usage may be given thus:

$$\text{define} \quad \ell^n_\alpha = \ell_\alpha \quad \text{if } n = 1$$

$$= \bigcup_{\beta : \beta \in \ell^{n-1}_\alpha} \ell_\beta \quad \text{if } n > 1$$

then C_i is used iff $\exists\, m \in \mathbb{N} : m \leq |C|$

and $$C_i \in \ell^m_{C_1}$$

(c) Recognition of non-terminating grammars: We make the reasonable demand that any class name (linked to the root class) should be capable of generating at least one, possibly empty, string over $T \cup \{\Lambda\}$. In particular this excludes productions of the form

$$X \rightarrow \alpha X \beta$$

and

$$X \rightarrow \alpha Y$$

$$Y \rightarrow \beta X$$

etc. where no other productions exist.

Formally we require that

$$\{\gamma : C_i \overset{*}{\rightarrow} \gamma\} \cap (T^* \cup \{\Lambda\}) \neq \emptyset$$

for all $C_i \in C$, where T^* is the set of non-empty strings over T.

This property can easily be checked by amending the linkage relation ℓ_α, so that:

if $\exists\, \alpha \rightarrow \gamma : \alpha \in C$ and γ is either a string over T or $\gamma = \Lambda$

then $\tau \in \ell_\alpha$, where τ is a special symbol.

Call the extended relation ℓ^* over $C \cup \{\tau\}$

now let $\ell^{*1}_\alpha = \ell^*_\alpha$

and $\ell^{*n}_\alpha = \bigcup_{\beta : \beta \in \ell^{*\,n-1}_\alpha \setminus \{\tau\}} \ell^*_\beta$ $\quad (n > 1)$

then C_i derives a terminal string (or Λ) iff $\exists\, m \in \mathbb{N} : m \leq |C|$ and

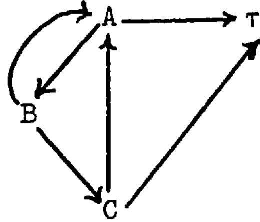
$$\tau \in \ell^{*\,m}_{C_i}$$

The diagram of the extended linkage relation ℓ^* is called the linkage graph.

e.g. (i)

$$\begin{cases} A \rightarrow aBc|d \\ B \rightarrow cA|eC \\ C \rightarrow x|yA \end{cases}$$

has the linkage graph:



This grammar has no non-terminating classes, however the following has:

e.g. (ii)

$$\begin{cases} A \rightarrow bB|c \\ B \rightarrow dC \\ C \rightarrow eB \end{cases}$$

A ⟶ T, A ↓ B, B ↓ C, C ↷ B

(d) <u>Removal of duplicate classes</u>: using the same characterisation of equivalence of productions as given in the definition of the extended factorization procedure we require to replace all occurrences of the name C_j by the name C_i whenever,

(i) $C_i ::= \alpha_1|\alpha_2|\ldots|\alpha_n$

and $C_j ::= \alpha_{f(1)}|\alpha_{f(2)}|\ldots|\alpha_{f(n)}$

where f is a bijection on $\{1,2,\ldots,\}$

or (ii) $C_i ::= h(C_i)$

and $C_j ::= g(C_j)$

where g, and h are string functions which are equivalent under (i)

e.g. $\equiv \begin{cases} X \rightarrow a|b \\ Y \rightarrow b|a \end{cases}$

and $\equiv \begin{cases} X \rightarrow aX|b \\ Y \rightarrow b|aY \end{cases}$

2.1.2.5 The Composite Algorithm

SYMAL
(i) Remove redundant classes and check for non-termination of grammar.
(ii) Derive LFF
(iii) Remove redundant and duplicate classes
(iv) Check and remove BU violations
(v) Remove redundant and trivial classes.

This algorithm may fail at stage (i) non-terminating grammar, stage (ii) ambiguous grammar, and stage (iv) infinitely strong BU violation. Details of the failures are given in the relevant subalgorithms.

We claim that the grammars produced by the algorithm are much easier to handle from the point of view of naive left-to-right parsing although they may be unwieldy and difficult to read. In answer to this (implied) criticism, we remark that it would be both possible and desirable to keep two versions (one original, one modified) of the grammar for use by the human reader and the syntax checker respectively. However a modified notation which makes these modified grammars 'readable' is given in §2.2.

2.2 On Syntax Construction

Using the material of §2.1 we formulate a set of rules which may be used in constructing the syntax[†] of a language so that, without modification, it is fully recognisable by a simple left-to-right no-lookahead parser.

Notes on the specification of several common constructs are also included.

As mentioned in the Introduction to the thesis, 'syntax' refers to the pure syntax which can be generated by a BNF grammar and ignores all context-sensitive semantic restraints which must elsewhere be checked.

2.2.1 The Construction Rules

The rules are of two types, local and global, these correspond to the Left-factored from (LFF) of productions and to No-backup (NBU) grammars. A detailed examination of LFF and NBU is given in §2.1, to which the reader is referred. Below we describe the required forms but give no justification of their relevance.

2.2.1.1 Local Rules

If a production has the form

$$\langle x \rangle ::= \alpha_1 | \alpha_2 | \alpha_3 | \dots | \alpha_n$$

where α_i $(1 \leq i \leq n)$ are strings of terminal and non-terminal symbols; then we require that each α_i $(1 \leq i \leq n)$ beginswith a different terminal symbol with the possible exception of α_n which may be the null string, Λ.

[†] More properly, we use the rules for constructing the productions which define the syntax. The rules given do not explicitly govern the syntax but the way it is specified by a suitable grammar.

LLF is obtainable by using the identities of §2.2.2.1; however the resultant production will in general be long and unwieldy. The notation introduced in §2.2.4 will allow these lengthy productions to be cast into a more readable form.

2.2.1.2 Global Rule

The global rule corresponds to the set of conditions given by Knuth [59][†] and which, by virtue of the LFF of productions, reduce to:

$$\text{if } \exists \langle x \rangle : \langle x \rangle \rightarrow \Lambda$$

then we require that

$$\text{first } (\langle x \rangle) \cap \text{follow } (\langle x \rangle) = \emptyset$$

The functions 'first' and 'follow' are formally defined elsewhere in §2.1, but may intuitively be thought of as:

first (X) = the set of terminals which begin alternatives in the defining production rule

follow (X) = the set of all terminals which may follow an occurrence of the class 'X' in the grammar.

The location of all X : X → Λ and the construction of the sets first (X) are trivial.

The formation of follow (X) is, in general, non-trivial but is made considerably easier if lists of similar constructs within the syntax are defined in a uniform and useful way - see §2.2.3.

[†] It is conjectured that the conditions do not hold for some ambiguous grammars, however this last point does not affect the thesis since an ambiguous construct will only be recognised in one way by any well-defined parser.

2.2.2 Identities

An extensive set of transformation identities has been given [25] and is not repeated here. Most of those identities are applicable only in extremely unnatural constructs. The identities given below are considered to be the more commonly encountered and desirable ones.

2.2.2.1 Local identities

(a) Factorization: if $X : X ::= \alpha\beta | \alpha\gamma$

then $\Rightarrow$ $X ::= \alpha X'$
$X' ::= \beta | \gamma$

where α, β, γ are strings of terminals and non-terminals and $\alpha \neq \Lambda$.

(b) Direct left recursion:

if $X : X ::= X\alpha | \beta$

then $\Rightarrow$ $X ::= \beta X'$
$X' ::= \alpha X' | \Lambda$

where α, β are arbitrary strings.

(c) Simple BU-violation:

e.g.: $X : X ::= x\alpha X x\beta | \gamma | \Lambda$

(the first x is marked $\in$ first (X); the second x is marked $\in$ follow (X))

violations of this type can usually be resolved but meaningful equivalents can be constructed in a more systematic way as in §2.2.3.

2.2.2.2 Non-local Identities

(a) Back substitution of productions:

$$\text{if } X ::= \alpha_1|\alpha_2|\ldots|\alpha_n \quad \text{and}$$

$$\exists i\ (1 \le i \le n) : \alpha_i = C\beta_i$$

where C is a non-terminal, then the definition of C has to be substituted for this occurrence of C in α_i.

So if $C = \gamma_1|\gamma_2|\ldots|\gamma_m$

then X becomes

$$X ::= \alpha_1|\alpha_2|\ldots|\alpha_{i-1}|\gamma_1\beta_i|\gamma_2\beta_i|\ldots|\gamma_m\beta_i|\alpha_{i+1}|\ldots|\alpha_n$$

The order in which this substitution operation is applied to the productions is important (for details see p.26) moreover if a suitable starting point for the substitution cannot be found we must appeal to an alternative technique (b).

(b) Unlocking of productions:

should (a) above not be applicable and there exist productions which are not in terminal (i.e. LF) form; then we must use a technique analogous to solving systems of linear equations (see page 41). It is not described here since such interlinking of non-terminals is unlikely to occur in a 'real' language and should not occur synthetically.

(c) General BU-violations:

usually these are very tedious to detect and are the result of untidy syntax design. As in 2.2.2.1 (c) we refer the reader to §2.2.3.

2.2.3 General Remarks and Hints:

Of what follows most is 'obvious'. The points which are of special importance are marked by an asterisk.

2.2.3.1 Connectedness

All non-terminals (and terminals) of the grammar must be contained in at least one sentence of the derived language.

2.2.3.2 Finite Generation

Each sentence of the language must be of finite (but possibly unbounded) length and be generated in a finite number of steps.

2.2.3.3 Minimality

If two or more non-terminals generate the same set of sub-sentences, then replace all occurrences of such non-terminals by a single (possibly new) non-terminal and form a suitable defining production. Some of the original classes will now be disconnected and removed by 2.2.3.1.

*2.2.3.4 On the occurrence of Λ

As stated in §2.2.1.2 BU violations can only occur if we have a Λ-producing class.

$$\text{i.e.} \quad \exists X : X \xrightarrow{*} \Lambda$$

and by the restrictions imposed by LFF, this reduces to

$$\exists X : X \rightarrow \Lambda$$

$$\text{i.e.} \quad X : X ::= \alpha_1 | \alpha_2 | \cdots | \alpha_{n-1} | \Lambda$$

Naturally, the constructs present in the syntax must to a large extent reflect the constructs of the language and over these we have no control.

However the more commonly occurring instances of Λ-producing classes present in 'real' languages should not give rise to BU violations if the following guide-lines are adhered to.

*2.2.3.4.1 Factorization

BU violations can only arise from factorization if symbols are 'misgrouped'. In non-recursive constructs a regrouping is always possible.

Suppose $\exists A : A ::= \alpha | \alpha\beta$

then $A ::= \alpha A'$

and $A' ::= \beta | \Lambda$

Now $A' \rightarrow \Lambda$ and $\beta \in$ first (A')

Violation can occur iff $\beta \in$ follow (A'). However A' only occurs in the definition of A and Λ follows A' in A

so follow (A) = follow (A')

i.e. $\beta \in$ follow (A')

$\Leftrightarrow$ $\beta \in$ follow (A)

$\Leftrightarrow$ $\exists C : C ::= \gamma A \beta$

then $C ::= \gamma A \beta$

$\Rightarrow$ $C ::= \gamma\alpha\beta\beta | \gamma\alpha\beta$

$$\Rightarrow \begin{cases} C ::= \gamma\alpha\beta\beta | \gamma\beta C' \\ C' ::= \beta | \Lambda \end{cases}$$

Since in the above instance the language constructs were assumed to be non-recursive the process must ultimately end when we have re-grouped productions of X and $\beta \notin$ follow (X).

*2.2.3.4.2 Lists and Sequences

Recursion is often used to describe a list construction even though a list is fundamentally not a recursive entity.

Typically we may have:

(a)† <unsigned integer> ::= <digit>|<unsigned integer> <digit>

i.e. $X ::= d|Xd$

Here X yields a sequence of one or more 'd's.

(b)† <actual parameter list> ::= <actual parameter>|<actual parameter list> <parameter delimiter><actual parameter>

i.e. $Y ::= a|Yda$

Here Y is a proper (i.e. non-empty) list of 'a's delimited by 'd's.

(c) $X ::= Xd|\Lambda$

A (possibly empty) sequence of 'd's.

(d)
$$Y ::= aY_1|\Lambda$$
$$Y_1 ::= daY_1|\Lambda$$

Y generates an empty list or a proper list of 'a's delimited by 'd's.

The above cases (a-d) can be used to generate all lists and sequences, moreover if such combinations are terminated (explicitly or implicitly) by a suitable terminal symbol, i.e. one which is distinguishable from the components and/or the delimiters of the list or sequence - this is surely a most reasonable thing to ask of <u>any</u> meaningful readable language - then we may use the following (LFF) equivalents without risk of NBU violations.

(a)
$$\begin{cases} X ::= dX' \\ X' ::= dX'|\Lambda \end{cases}$$
$$d \notin \text{follow } (X)$$

(b)
$$Y ::= aY'$$
$$Y' ::= daY'|\Lambda$$
$$d \notin \text{follow } (Y)$$

---

† examples taken from Algol-60 revised report [82].

(c) $X ::= dX|\Lambda$

$d \notin$ follow (X)

(d) $Y ::= aY'|\Lambda$

$Y' ::= daY'|\Lambda$

$a,d \notin$ follow (Y)

2.2.3.5 Trivial Classes

Any connected class (which is not the root) with only a single option may be removed by substituting its definition for each occurrence of the class name. By §2.2.3.2 such a class cannot occur (explicitly or implicitly) in its own definition.

2.2.4 Abbreviated BNF

Constructing a syntax as described in §2.2.1-3 gives rise to class definitions which have many (similar) options each of which may be of considerable length. For example an Algol-60 identifier (restricted to an upper-case alphabet) may be represented thus:

```
<identifier> ::= A <id'>|
                 B <id'>|
                 :
                 Z <id'>

      <id'> ::= A <id'>|
                B <id'>|
                :
                Z <id'>|
                Ø <id'>|
                1 <id'>|
                :
                9 <id'>|
                Λ
```

Such productions tend to make the syntax unreadable. As a first step to making these productions more easily understood we utilize some set theoretic notation:

Write the set {a,b,c} as

{a|b|c}

then:

<identifier> ::= {A|B|...|Z} <id'>

<id'> ::= {A|B|...|Z|Ø|1|...|9} <id'>|Λ

Extending this notion a little further:

<A> ::= axy|bxy|cxy|dy

becomes <A> ::= {{a|b|c}x|d}y

Such notation will enable us to rewrite many class definitions in a more compact form. In simple cases, e.g. <identifier> as above, the definition will also be easily intelligible, however if nesting of sets occurs, e.g. <A> above, then readability is impaired. To avoid nested bracketing we introduce a new entity called a direct terminal class.

A direct terminal class (DTC) is written thus

<name of class>

and is a class such that each option of its defining production begins with a terminal symbol or a DTC[†]. A DTC is defined in the same manner as other classes, indeed a given class may occur in a syntax as both a DTC and an 'ordinary' class.

Using the examples as above we have:

[†] Note that no circularity can be incurred by such a definition since DTC's are all derived from LFF productions.

```
    <identifier> ::= <alpha> <id'>
           <id'> ::= <alphanumeric> <id'>|Λ
  <alphanumeric> ::= <alpha>|<digit>
         <alpha> ::= A|B|C|...|X|Y|Z
         <digit> ::= Ø|1|2|...|8|9

           <A>   ::= <A'>y
           <A'>  ::= <A">x|d
           <A">  ::= a|b|c
```

It is easily seen that the resultant syntax is comprehensible yet it is quickly transformed to a left-to-right recognisable form. Of course, if a stack is introduced we can parse directly left-to-right from such a grammar.

A grammar which includes DTC's will be termed an abbreviated BNF (ABNF).

The concept of ABNF is to be found in many other systems, notably SID ([46], [121]).

2.2.5 Consequences

Any grammar generated by SID is unambiguous, [46], and by virtue of the similarity in strategy, so is any grammar which is constructed in accordance with the rules of §2.2.

Formulation of a 'suitable' syntax for Algol 60 - which is possible by [46] - will need to resolve such syntactic (see p. 8) ambiguities as:

```
<primary>                  <primary>
    ↓                          ↓
<variable identifier>      <procedure identifier><actual parameter part>
    ↓                          ↓                               ↓
<identifier>               <identifier>                        Λ
```

Similarly for other occurrences of implicit type attributes related to identical syntactic constructs.

Any ambiguities detected by the formation of LFF productions will give rise to productions of the form:

$$X ::= \Lambda | \Lambda$$

These may be removed by replacing X by Λ wherever it occurs, any required semantic distinction being made later.

Ambiguities incorporated in NBU violations should be detected by or, better still, prevented by satisfaction of the '$x \in$ follow (y)' predicates of §2.2.3.4.

It may also be desirable, in order to keep the number of constructs within manageable limits, to widen the syntax so as to accept strings which will later be rejected by other (semantic) restraints. A typical example of this is in Algol 60 [82] where the conditional expressions:-

<if clause> <simple arithmetic expression> else <arithmetic expression>

and

<if clause> <simple Boolean> else <Boolean expression>

may be usefully widened and combined to give:-

<if clause> <simple expression> else <expression>

Indeed, when dealing with polymorphic objects (e.g. 'global' identifiers in Algol procedures) similar constructs may not be distinguishable, by any method, before execution of a particular occurrence of that construct.

2.2.6 Summary

The rules of §2.2 provide a framework for the construction of (context-free) syntax [16] of a large set of real programming languages: any context-sensitive restrictions now fall into the realm of semantics and must take the form of compile-time or run-time validation checks. This is further discussed in later chapters.

CHAPTER 3

THE LANGUAGE X; AN INTRODUCTION

3.1 Informal description

X is an Algol-60-like block-structured language. The syntax is given below in a variation of BNF. These modifications to BNF, which were fully explained in the previous chapter, aid parsing; however if the underlines of various non-terminals are ignored we revert back to standard BNF, familiarity with which is assumed. This standard BNF analogue is sufficient for the informal definition.

Language X Syntax

1. < program > ::= < block >
2. < block > ::= begin < decn >; < stmts > end
3. < decn > ::= let < proper id list > be < type >
4. < proper id list > ::= < id > < id list >
5. < id list > ::= , < proper id list > | Λ
6. < type > ::= real | int
7. < stmts > ::= < stmt > < stmts 1 >
8. < stmts 1 > ::= ; < stmts > | Λ
9. < stmt > ::= < block > | goto < label > | < id 1 >:=< exp > |
   L < special >

10. < id 1 > ::= A | B | C[†]

11. < id > ::= A | B | C | L[†]

12. < label > ::= L < digit > < rest of int. >

13. < special > ::= < digit > < rest of int. > < unlab. stmt > |
    := < exp >

14. < digit > ::= Ø | 1 | 2 | ... | 8 | 9

15. < rest of int. > ::= < digit > < rest of int. > | Λ

16. < unlab. stmt > ::= < block > | goto < label > | < id > := < exp >

17. < exp > ::= < token > < exp follower >

18. < exp follower > ::= + < exp > | Λ

19. < token > ::= < id > | . < digit > < rest of int. > |
    < digit > < rest of int. > < rest of number >

20. < rest of number > ::= . < rest of int. > | Λ

That this grammar for X satisfies the conditions laid down for direct left to right recognition is verified in §3.2.

We now give two examples of valid X programs.

I.
```
begin let A, B, C be real;
      A := 3.0 ;
      B := A + 1 ;
      C := B + 1.7 ;
      begin let B, L be int ;
            L := C ;
            B := A
      end
end
```

[†]Note: < id > could be the entire alphabet and < id 1 > ≡ < id > \ L, but this would add nothing to the exposition. However, notice the importance of including 'L' and the related parsing problems.

```
II.   begin let A be real;
          begin let B, C be int ;
                C := 7;
                B := 3;
          L2    A := B + 2.9;
                goto L7;
          L3    B := C;
                goto L2;
          L7 begin let B be real;
                B := A;
                C := C + 7;
                A := C + B
             end;
             goto L3
          end
      end
```

3.2 Verification that the given syntax is suitable for left-to-right parsing

Referring to [25] and [59] and the relations 'first' and 'follow' of Chapter 2 we recall that a necessary condition for an unambiguous language (and its associated grammar) to be recognisable directly from left to right is:-

$$\text{If } \exists < x > \; : \; < x > \rightarrow \Lambda$$

$$\text{then first } (< x >) \cap \text{follow } (< x >) = \emptyset$$

This condition may be checked by using the 'Produced Head Symbols' table and the 'C1 Matrix for Stacking Decision' produced by McKeeman's Compiler Generating System [77].

Before discussing the usage of the tables we note that _|_ denotes the end of input to the recogniser, and Λ is replaced by <empty>.

Now, first we find all non-terminals, N, such that

N → <empty>

This is detected by:-

PHS(N, <empty>) = Y

Having found such an N we use the PHS table to find all terminals T_i in first (N) and the C1 Matrix (C1M) to find all terminals T_j in follow (N):

i.e. first (N) = {x : PHS(N,x) = Y}

follow*(N) = {x : C1M(N,x) = Y,N or #}

[follow* is restricted to terminals because C1M is similarly restricted.]

We now check that these conditions hold in the syntax for language X, the tables for which are given at the end of this section.

Trivially we need only consider the non-terminals numbered 38, 43, 44, 45 and 47:

i.e. < iden list >

< statements 1 >

< rest of number >

< rest of integer >

< expression follower >

This is because

PHS (< iden list >, < empty >) = Y (see p. 91)
etc.

Now[†], first (38) = {2} = {,} (see p. 91)

first (43) = {1}

first (44) = {18}

first (45) = {7, 8, ..., 15, 16}

first (47) = {17}

Similarly, from C1M we have

follow (38) = {21} (see p. 92)

follow (43) = {22}

follow (44) = {17}

follow (45) = {1, 3, 4, 5, 6, 17, 18, 26, 27}

follow (47) = {1}

Notice that since XPL back-substitutes to the level of terminal symbols we may ignore the second Backup Condition since

---

[†] From hereon we restrict consideration of first and follow to their terminal elements $\setminus \Lambda$, this causes no loss of generality.

if $X \in f_o(C_i)$ and X non-terminal

then $a \in f_1(X) \Rightarrow a \in f_o(C_i)$

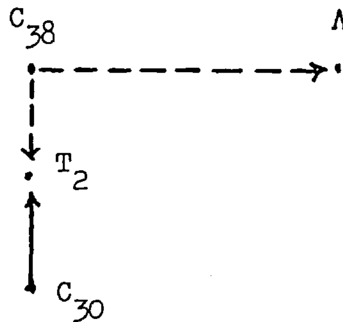
Now, writing C_{38} for <iden list> etc. (see tables), we examine in turn the above classes

$$f_1(C_{38}) = \{T_2, T_{28}\}$$

$$= \{T_2, \Lambda\}$$

$$f_o^{-1}(T_2) = \{T_3, \ldots, T_6, C_{30}\}$$

i.e.



Here there is a violation iff there is a 'following-sequence' from C_{38} to C_{30}. We consider such sequences for all non-terminals later on in this section.

$$f_1(C_{43}) = \{T_1, \Lambda\}$$

$$f_1(C_{44}) = \{T_{18}, \Lambda\}$$

$$f_1(C_{45}) = \{T_7 \ldots T_{16}, \Lambda\}$$

$$f_1(C_{47}) = \{T_{17}, \Lambda\}$$

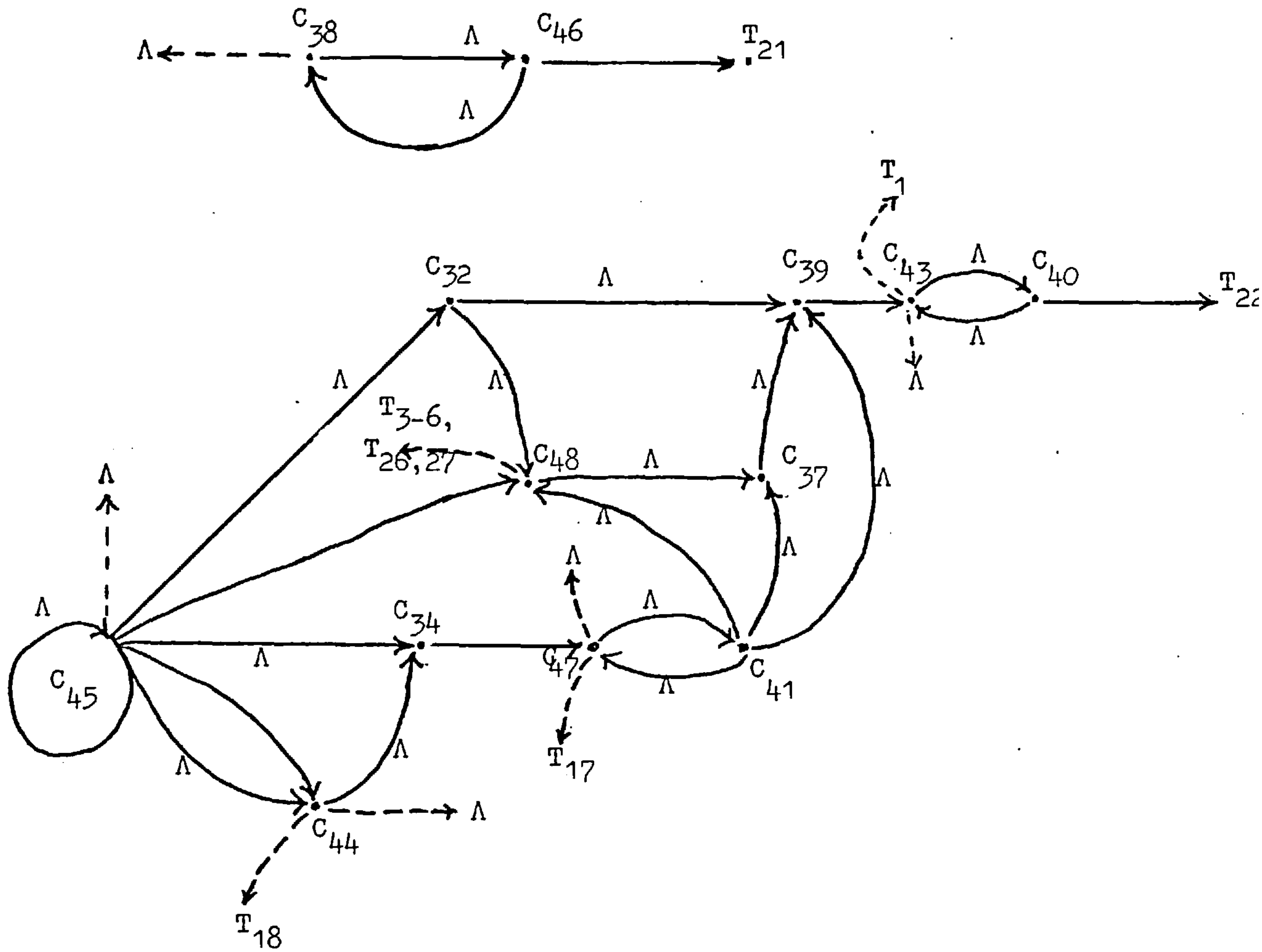
$$f_o^{-1}(T_1) \supseteq \{C_{29}, C_{31}, C_{32}, C_{37}, C_{39}, C_{41}, C_{42}, C_{45}, C_{47}, C_{48}\}$$

$$f_o^{-1}(T_{18}) \supseteq \{C_{45}\}$$

$$f_o^{-1}(T_7 \ldots T_{16}) \supseteq \{C_{33}\}$$

$$f_o^{-1}(T_{17}) \supseteq \{C_{30}, C_{34}, C_{44}, C_{45}\}$$

The relevant sections of the backup diagram are as follows.



From these segments it can be seen that there exist no "following-sequences",

from C_{38} to C_{30}

from C_{43} to C_{29}

C_{31}

etc.

from C_{44} to C_{45}

from C_{45} to C_{33}

nor from C_{47} to C_{30}

C_{34}

etc.

Hence $\nexists \ell : f_o \circ \ell (C_i) \cap (f_1(C_i) \setminus \Lambda) \neq \emptyset$

for any $C_i \in f_1^{-1} (\Lambda)$ where ℓ is a following sequence.

Therefore the NBU condition is satisfied.

The relevant XPL-generated information now follows:-

PRODUCTIONS

```
£ GRAMMAR FOR LANGUAGE X

 1  <PROGRAM>  ::=  <BLOCK>

 2  <BLOCK>  ::=  'BEGIN' <DECLARATION> ; <STATEMENTS> 'END'

 3  <DECLARATION>  ::=  'LET' <PROPER IDEN LIST> 'BE' <TYPE>

 4  <PROPER IDEN LIST>  ::=  <IDEN> <IDEN LIST>

 5  <IDEN LIST>  ::=  , <PROPER IDEN LIST>
 6                 |  <EMPTY>

 7  <TYPE>  ::=  'REAL'
 8            |  'INT'

 9  <STATEMENTS>  ::=  <STATEMENT> <STATEMENTS 1>

10  <STATEMENTS 1>  ::=  ; <STATEMENTS>
11                    | <EMPTY>

12  <STATEMENT>  ::=  <BLOCK>
13                 |  'GOTO' <LABEL>
14                 |  <IDEN 1> := <EXPRESSION>
15                 |  L <SPECIAL>

16  <IDEN 1>  ::=  A
17              |  B
18              |  C

19  <IDEN>  ::=  A
20            |  B
21            |  C
22            |  L

23  <LABEL>  ::=  L <DIGIT> <REST OF INTEGER>

24  <SPECIAL>  ::=  <DIGIT> <REST OF INTEGER> <UNLABELLED STATEMENT>
25               |  := <EXPRESSION>

26  <DIGIT> ::=  0
27            |  1
28            |  2
29            |  3
30            |  4
31            |  5
32            |  6
33            |  7
34            |  8
35            |  9

36  <REST OF INTEGER>  ::=  <DIGIT> <REST OF INTEGER>
37                       |  <EMPTY>
```

```
38  <UNLABELLED STATEMENT>  ::=  <BLOCK>
39                            |  'GOTO' <LABEL>
40                            |  <IDEN> := <EXPRESSION>

41  <EXPRESSION>  ::=  <TOKEN> <EXPRESSION FOLLOWER>

42  <EXPRESSION FOLLOWER>  ::= + <EXPRESSION>
43                          | <EMPTY>

44  <TOKEN>  ::=  <IDEN>
45             |  . <DIGIT> <REST OF INTEGER>
46             |  <DIGIT> <REST OF INTEGER> <REST OF NUMBER>

47  <REST OF NUMBER>  ::=  . <REST OF INTEGER>
48                      |  <EMPTY>
```

TERMINAL SYMBOLS

```
 1  ;
 2  ,
 3  L
 4  A
 5  B
 6  C
 7  0
 8  1
 9  2
10  3
11  4
12  5
13  6
14  7
15  8
16  9
17  +
18  .
19  :=
20  _|_
21  'BE'
22  'END'
23  'LET'
24  'INT'
25  'REAL'
26  'GOTO'
27  'BEGIN'
28  <EMPTY>
```

NONTERMINALS

```
29  <TYPE>
30  <IDEN>
31  <BLOCK>
32  <LABEL>
33  <DIGIT>
34  <TOKEN>
35  <IDEN 1>
36  <PROGRAM>
37  <SPECIAL>
38  <IDEN LIST>
39  <STATEMENT>
40  <STATEMENTS>
41  <EXPRESSION>
42  <DECLARATION>
43  <STATEMENTS 1>
44  <REST OF NUMBER>
45  <REST OF INTEGER>
46  <PROPER IDEN LIST>
47  <EXPRESSION FOLLOWER>
48  <UNLABELLED STATEMENT>
```

<PROGRAM> IS THE GOAL SYMBOL.

PRODUCED HEAD SYMBOLS:

```
                                        1111111 1112222222222333 3333333344444444
                              +1234567890123456+7890123456789012+3456789012345678+
 1  ;                         |Y               |                |                |
 2  ,                         | Y              |                |                |
 3  L                         |  Y             |                |                |
 4  A                         |   Y            |                |                |
 5  B                         |    Y           |                |                |
 6  C                         |     Y          |                |                |
 7  0                         |      Y         |                |                |
 8  1                         |       Y        |                |                |
 9  2                         |        Y       |                |                |
10  3                         |         Y      |                |                |
11  4                         |          Y     |                |                |
12  5                         |           Y    |                |                |
13  6                         |            Y   |                |                |
14  7                         |             Y  |                |                |
15  8                         |              Y |                |                |
16  9                         |               Y|                |                |
                              +----------------+----------------+----------------+
17  +                         |                |Y               |                |
18  .                         |                | Y              |                |
19  :=                        |                |  Y             |                |
20  _|_                       |                |   Y            |                |
21  'BE'                      |                |    Y           |                |
22  'END'                     |                |     Y          |                |
23  'LET'                     |                |      Y         |                |
24  'INT'                     |                |       Y        |                |
25  'REAL'                    |                |        Y       |                |
26  'GOTO'                    |                |         Y      |                |
27  'BEGIN'                   |                |          Y     |                |
28  <EMPTY>                   |                |           Y    |                |
29  <TYPE>                    |                |       YY   Y   |                |
30  <IDEN>                    |  YYYY          |             Y  |                |
31  <BLOCK>                   |                |          Y   Y |                |
32  <LABEL>                   |  Y             |               Y|                |
                              +----------------+----------------+----------------+
33  <DIGIT>                   |      YYYYYYYYYY|                |Y               |
34  <TOKEN>                   |  YYYYYYYYYYYYYY| Y           Y  |YY              |
35  <IDEN 1>                  |   YYY          |                |  Y             |
36  <PROGRAM>                 |                |          Y   Y |   Y            |
37  <SPECIAL>                 |      YYYYYYYYYY|  Y             |Y   Y           |
38  <IDEN LIST>               |Y               |           Y    |     Y          |
39  <STATEMENT>               |  YYYY          |         YY   Y |  Y   Y         |
40  <STATEMENTS>              |  YYYY          |         YY   Y |  Y   YY        |
41  <EXPRESSION>              |  YYYYYYYYYYYYYY| Y           Y  |YY      Y       |
42  <DECLARATION>             |                |      Y         |         Y      |
43  <STATEMENTS 1>            |Y               |           Y    |          Y     |
44  <REST OF NUMBER>          |                | Y         Y    |           Y    |
45  <REST OF INTEGER>         |      YYYYYYYYYY|           Y    |Y           Y   |
46  <PROPER IDEN LIST>        |  YYYY          |             Y  |             Y  |
47  <EXPRESSION FOLLOWER>     |                |Y          Y    |              Y |
48  <UNLABELLED STATEMENT>    |  YYYY          |         YY  YY |               Y|
                              +----------------+----------------+----------------+
```

C1 MATRIX FOR STACKING DECISION:

```
                                        1111111 111122222222
                               1234567890123456 789012345678
                              +----------------+------------+
 1  ;                         |  YYYY          |         YY |
 2  ,                         |  YYYY          |            |
 3  L                         | N    YYYYYYYYYY|N #        N|
 4  A                         | N              |N N        N|
 5  B                         | N              |N N        N|
 6  C                         | N              |N N        N|
 7  0                         |      NNNNNNNNNN|           N|
 8  1                         |      NNNNNNNNNN|           N|
 9  2                         |      NNNNNNNNNN|           N|
10  3                         |      NNNNNNNNNN|           N|
11  4                         |      NNNNNNNNNN|           N|
12  5                         |      NNNNNNNNNN|           N|
13  6                         |      NNNNNNNNNN|           N|
14  7                         |      NNNNNNNNNN|           N|
15  8                         |      NNNNNNNNNN|           N|
16  9                         |      NNNNNNNNNN|           N|
                              +----------------+------------+
17  +                         |  YYYYYYYYYYYYYY| Y          |
18  .                         |      YYYYYYYYYY|           Y|
19  :=                        |  YYYYYYYYYYYYYY| Y          |
20  _|_                       |                |          Y |
21  'BE'                      |                |       YY   |
22  'END'                     |N               |   N       N|
23  'LET'                     |  YYYY          |            |
24  'INT'                     |N               |            |
25  'REAL'                    |N               |            |
26  'GOTO'                    |  Y             |            |
27  'BEGIN'                   |                |      Y     |
28  <EMPTY>                   |N NNNN          |NN  NN   NNN|
29  <TYPE>                    |N               |            |
30  <IDEN>                    | Y              |N Y        #|
31  <BLOCK>                   |N               |   N       N|
32  <LABEL>                   |N               |           N|
                              +----------------+------------+
33  <DIGIT>                   |      YYYYYYYYYY|           Y|
34  <TOKEN>                   |                |Y          Y|
35  <IDEN 1>                  |                |  Y         |
36  <PROGRAM>                 |                |   N        |
37  <SPECIAL>                 |N               |           N|
38  <IDEN LIST>               |                |    Ⓝ       |
39  <STATEMENT>               |Y               |           Y|
40  <STATEMENTS>              |                |     #      |
41  <EXPRESSION>              |N               |           N|
42  <DECLARATION>             |Y               |            |
43  <STATEMENTS 1>            |                |     N      |
44  <REST OF NUMBER>          |                |N          N|
45  <REST OF INTEGER>         |N ####          |N#       ###|
46  <PROPER IDEN LIST>        |                |    #       |
47  <EXPRESSION FOLLOWER>     |N               |           N|
48  <UNLABELLED STATEMENT>    |N               |           N|
                              +----------------+------------+
```

3.3 Executive Semantics

The semantics of X are as follows:-

(i) the block structure is similar to that of Algol-60. Explicit transfers across block boundaries are forbidden.

(ii) Assignment is treated as in Algol-68, i.e. we can widen an <u>integer</u> to a <u>real</u> but not vice-versa. Similar types can be assigned in the usual way.

(iii) Addition between mixed node quantities is valid and integers are widened (see [108]) as required.

The problem of (semantic) validation is dealt with in chapter 7, however we need to know about the executable semantics before we can describe the initial CPS (Carabiner Program Space) for X, hence we give the following (tentative) set of productions with semantic injections. Note that these are only intended as a guide to the run-time semantics which govern a <u>valid</u> X program.

1. < program > ::= < <u>block</u> >$_{1.1}$

2. < block > ::= <u>begin</u> ⦃ block entry ⦄
   < decn >$_{2.1}$
   ;
   < stmts >$_{2.2}$
   <u>end</u> ⦃ block exit ⦄

3. < decn > ::= <u>let</u>
   < proper id list >$_{3.1}$
   <u>be</u>
   < type >$_{3.2}$ ⦃link list to type ⦄

4. < proper id list > ::= < <u>id</u> >$_{4.1}$ ⦃ save name ⦄
   < id list >$_{4.2}$

5. < is list > ::= ,
< proper id list >$_{5.1}$ |
Λ

6. < type > ::= real |
int

7. < stmts > ::= < stmt >$_{7.1}$ < stmts 1 >$_{7.2}$

8. < stmts 1 > ::= ; < stmts >$_{8.1}$ | Λ

9. < stmts > ::= < block >$_{9.1}$ |
goto
< label >$_{9.2}$ ≮ generate 'goto' ≯† |
< id 1 >$_{9.3}$
:=
< exp >$_{9.4}$ ≮ generate assignment ≯ |
L < special >$_{9.5}$

10. < id 1 > ::= A |
B |
C

11. < id > ::= A |
B |
C |
L

12. < label > ::= L
< digit >$_{12.1}$
< rest of int. >$_{12.2}$

---

†We merge: ≮ generate 'goto' ≯ into a translation routine which simplifies the whole control network within an X-block. A fuller discussion of this is given in the appendix.

13. < special > ::= < digit >$_{13.1}$
< rest of int. >$_{13.2}$
< unlab. stmt >$_{13.3}$ |
:=
< exp >$_{13.4}$ ≮ generate assignment ≯

14. < digit > ::= Ø | 1 | 2 | ... | 9

15. < rest of int. > ::= < digit >$_{15.1}$
< rest of int. >$_{15.2}$ |
Λ

16. < unlab. stmt > ::= < block >$_{16.1}$ |
goto
< label >$_{16.2}$ ≮ generate goto ≯ |
< id >$_{16.3}$
:=
< exp >$_{16.4}$ ≮ generate assignment ≯

17. < exp > ::= < token >$_{17.1}$ < exp follower >$_{17.2}$

18. < exp follower > ::= +
< exp >$_{18.1}$ ≮ generate addition ≯ |
Λ

19. < token > ::= < id >$_{19.1}$ ≮ take value ≯ |
.
< digit >$_{19.2}$
< rest of int. >$_{19.3}$ |
< digit >$_{19.4}$
< rest of int. >$_{19.5}$
< rest of number >$_{19.6}$

20. < rest of number > ::= .
< rest of int. >$_{20.1}$ |
Λ

<u>execution semantics</u>:

⊰ block entry ⊱

⊰ block exit ⊱

⊰ link list to type ⊱

⊰ save name ⊱ is incorporated within translation

⊰ generate 'goto' ⊱

⊰ generate assignment ⊱

⊰ generate addition ⊱

The above 'injections' (and others concerned with validation) will later be defined by actions in the program space specified by the Carabiner language.

CHAPTER 4

BASE FUNCTIONS OF THE SOURCE LANGUAGE

In order that our system should (a) be definitive and (b) use the notation of the high-level source language we use a derivative of Markov's normal algorithms [74] to specify the 'base functions'. These are the routines, details of which are often ignored by other systems, that perform such functions as integer addition, subtraction etc., similarly real operations, and also logic operations. Naïvely we may regard them as the 'hardware' functions of the system.

At this point we note that by allowing translations into binary representation a more pedantic set of definitions could be given by using the substitution operator S (see Chapter 6), however, such detail would be restrictive since by using different representation (or types of representation, e.g. binary, binary coded decimal, characters) equivalent processes would naturally give different results. We take the stand that any implementation of a given function is correct iff for each input the result (characterized as in the source language) is the same as that produced by the defining algorithm.

This chapter contains the elements of the Theory of Markov Algorithms together with some extensions. As we wish to present this as concisely as is practicable it tends to consist of a set of

disjoint statements, rather sparse and unelaborated. Anyone with a prior knowledge of Markov Algorithms may skip this chapter, merely referring to it as is found necessary.

In §4.1 we define the basic terminology associated with Markov Algorithms (MAs), giving two examples of such algorithms in §4.2. Extended MAs are defined in §4.3, and, in §4.4-5 we give details of how elementary MAs can be combined.

4.1 Markov Algorithms:

Our characterization of Markov's [74] scheme for the 'abstraction of potential realizability' follows closely that of Mendelson [78] but the notation is modified to embody the concept of the level of an algorithm to facilitate the formation of combinations of algorithmic schemata.

We define the fundamentals of a Markov algorithm and describe further notation as it becomes necessary.

Definition by equality is denoted thus:-

$$\text{'definiens'} \overset{\text{def}}{=} \text{'definiendum'}$$

Defn: The input to an algorithm, the algorithm itself, and the result of the algorithm are represented by sequences of atomic symbols which we call letters.

Defn: A non-empty set of letters is called an alphabet. We denote the set of all alphabets by Ab.

Given $A \in Ab$, the set $W(A)$ of words of A is the free monoid generated by A under the operation of concatenation with unit element Λ, defined below (see [64], [71]). To denote arbitrary words we use capital letters or barred small letters. E.g. A or $\bar{a}$.

If $a \in A \in Ab$ then we write a for a^1 and Λ for a^0, $\forall\, a \in A$. Λ is called the empty word and

$$\Lambda \notin B \quad \forall \quad B \in Ab$$

yet

$$\Lambda \in W(B) \quad \forall\, B \in Ab \,.$$

Defn: The product of two words is denoted and defined by juxtaposing them, and multiples of words are represented by:-

$$\bar{x}^n \,:\, \bar{x} \in W(A)$$

(the bar may be omitted if $\bar{x} \in A$).

Defn: Given $\bar{x} \in W(A)$, denote by $l(\bar{x})$, the length of the word $\bar{x}$, i.e. the number of letters occurring in it such that:

$$
\begin{array}{llll}
l(a) & \overset{\text{def}}{=} & 1 & a \in A \\
l(\Lambda) & \overset{\text{def}}{=} & 0 & \\
l(\bar{x}^n) & \overset{\text{def}}{=} & nl(\bar{x}) & \bar{x} \in W(A) \\
l(a^n) & \overset{\text{def}}{=} & n & a \in A \\
l(\bar{x}\bar{y}) & \overset{\text{def}}{=} & l(\bar{x}) + l(\bar{y}) & \bar{x},\bar{y} \in W(A)
\end{array}
$$

We assume the existence of a denumerable number of alphabets

$$A_i \,:\, i \in \mathbb{N} \overset{\text{def}}{=} \{0,1,2,\ldots\}$$

and

$$\{A_i \,:\, i \in \mathbb{N}\} \subset Ab$$

We write A for A_o and assume there are special symbols '$\rightarrow$' and '.' such that

$$\rightarrow\,,\ .\ \notin A \;\; \forall A \in Ab$$

Defn: Given $A,B \in Ab$ st. $A \subset B$ we say that A is a subalphabet of B, equivalently that B is an extension of A.

If we tentatively regard an algorithm as a function whose value for any given argument can be found by a finite (terminating) calculation; and if the domain of the function f is denoted by $\mathfrak{D}_f$; then, if $\mathfrak{D}_f \subset W(A)$ we say that f is an algorithm in A.

Defn: An algorithm over A is an algorithm in an extension of A.

Defn: An algorithmic schema, $\mathfrak{a}$, is a finite sequence of productions (or sets of productions) denoted by:

$$\begin{matrix} P_1 \rightarrow (.)\ Q_1 \\ P_2 \rightarrow (.)\ Q_2 \\ \vdots \quad \vdots \\ P_n \rightarrow (,)\ Q_n \end{matrix}$$

where, if $\mathfrak{a}$ represents an algorithm in $A(\in Ab)$, then:

$$P_i, Q_j \in W(A) \quad (1 \leq i, j \leq n) .$$

There are three types of production.

a) simple, written $P \rightarrow Q$

b) terminal, written $P \rightarrow .Q$

c) total,

written $P \rightarrow Q.$ for simple total

and $P \rightarrow .Q.$ for terminal total.

We shall show in §4.2 that total productions may be written in terms of simple and terminal productions and hence are

merely a notational convenience. The notation

$$P \rightarrow (.)Q$$

is used for the general representation of an arbitrary production of any of the above types a), b) or c).

<u>Defn</u>: Given $\bar{x} \in W(A)$, let $l = l(\bar{x})$ then define

$$(\bar{x})_i \quad (1 \leq i \leq l) \text{ to be the } \underline{i^{th} \text{ letter in } \bar{x}}$$

e.g.:

if $\qquad A = \{a,b,c\}$

and $\qquad \bar{x} \equiv ab^2ca^4b \in W(A)$

(here $\bar{x}$ is given in its canonical form) then

$$(\bar{x})_1 = a$$
$$(\bar{x})_2 = b$$
$$(\bar{x})_3 = b$$
....
$$(\bar{x})_6 = a$$
....
$$(\bar{x})_9 = b$$

For completeness, we stipulate

$$(\bar{x})_i \stackrel{def}{=} \Lambda \quad \forall i > l(\bar{x})$$

Then, given two words $\bar{x} \in W(A)$

$\bar{y} \in W(B)$

$$\bar{x} = \bar{y} \stackrel{def}{=} (\bar{x})_i = (\bar{y})_i \quad \forall i \in \mathbb{N}$$

and trivially, if $\bar{x} = \bar{y}$ then $\bar{x}, \bar{y} \in W(A \cap B)$.

<u>Defn</u>: If $\bar{a},\bar{b} \in W(A) : A \in Ab$ then we say that <u>$\bar{a}$ occurs in $\bar{b}$</u> if $\exists\ \bar{c},\bar{d} \in W(A)$ such that

$$\bar{b} = \bar{c}\bar{a}\bar{d}$$

and denote this by $\bar{a}\ o\ \bar{b}$ otherwise $\bar{a} \not{o}\ \bar{b}$.

If $\bar{a}\ o\ \bar{b}$ and $l = l(\bar{b})$, then let

$$\left.\begin{aligned} b &= \bar{c}_1\ \bar{a}\ \bar{d}_1 \\ &= \bar{c}_2\ \bar{a}\ \bar{d}_2 \\ &\ \vdots \\ &= \bar{c}_j\ \bar{a}\ \bar{d}_j \end{aligned}\right\} \quad \begin{aligned} \text{where}\quad & l(\bar{c}_i) < l(\bar{c}_{i+1}) \\ & 1 \leq i \leq j-1 \\ \text{and}\quad & j \leq l \end{aligned}$$

\- be all the representations of $\bar{b}$ in the form $\bar{c}_i\ \bar{a}\ \bar{d}_i$ with $\bar{c}_i,\ \bar{d}_i \in W(A)$.

<u>Defn</u>: Then we say that the word $\bar{c}_1$ specifies the <u>leading occurrence</u> of $\bar{a}$ in $\bar{b}$ and the $\bar{a}$ immediately following $\bar{c}_1$ is that occurrence.

The action of the algorithmic schema, $\mathfrak{a}$, defined by the sequence of productions:-

$$\left.\begin{aligned} & P_1 \rightarrow (.)Q_1 \\ & \vdots \\ & P_n \rightarrow (.)Q_n \end{aligned}\right\} \quad \begin{aligned} & P_i, Q_j \in W(A) \\ & (1 \leq i,\ j \leq n) \end{aligned}$$

over the alphabet A, may now be described.

(The analytical notation for mapping follows [60], [65].)

Let the input to $\mathfrak{a}$ be $P \in W(A)$

(i) if $P_i \not{o}\ P\ \forall i : 1 \leq i \leq n$ then $\mathfrak{a}$ has no effect on P

(i.e. the algorithm passes all its n stages without action)

So,

$$\alpha : P \mapsto P$$

which we denote by

$$\alpha : P \sqsupset$$

(ii) otherwise

$$\exists\, j : 1 \leq j \leq n$$

where

$$P_j \circ P$$

and

$$P_i \not\circ P \quad \forall i : 1 \leq i < j$$

The jth production is then to be applied to the input in the following way:-

Since $P_j \circ P$ there is a leading occurrence of P_j in P, let this be specified by R_1 such that

$$P = R_1 P_j R_2$$

(a) If $P_j \rightarrow Q_j$ (i.e. the jth production is simple) - then replace this occurrence of P_j by Q_j.

i.e.

$$\alpha_{(j)} : R_1 P_j R_2 \rightarrow R_1 Q_j R_2$$

where $\alpha_{(j)}$ is the jth production in the schema for α.

Then execute the algorithmic schema once again (from stage (i)) using this modified word as input.

(b) If $P_j \rightarrow .Q_j$ (terminal) then carry out the process (a) but instead of recycling the process, halt and exit with the transformed word as the result.

(c) If $P_j \to Q_j\cdot$

or $P_j \to \cdot Q_j\cdot$ then the whole of the 'input' word is replaced by Q_j and processing continued or terminated according to whether or not the production is simple.

If a simple production is applied and <u>transforms</u> P to R, say, then we write

$$\mathcal{a} : P \vdash R$$

If a terminal production is applied and transforms P to R, then the algorithm execution terminates and we write

$$\mathcal{a} : P \vdash \ .R$$

We will have cause in the sequel to refer to many different algorithms and we label these

$$\mathcal{a}_i \ : i \in \mathbb{N}$$

So $\mathcal{a}_{i(j)}$ represents the jth production of the ith algorithm.

<u>Defn</u>: An algorithmic schema,

$$\mathcal{a}_i = \{\mathcal{a}_{i(j)} \ : 1 \leq j \leq n_i \ \text{ for some } \ n_i \in \mathbb{N}\}$$

is a <u>Markov Algorithm</u>, MA, in the alphabet A if

(a) $$\mathcal{a}_{i(j)} \equiv P_j \to (.)Q_j, \ P_j, \ Q_j \in W(A) \qquad \forall j : 1 \leq j \leq n_i$$

(b) for any $P \in W(A)$, $\exists$ a sequence $R_o, R_1, \ldots, R_k$ such that:

$$R_o \equiv P$$

and $R_k \equiv R$ (say)

and where, for $0 \leq j \leq k-2$

$$\mathcal{a}_i : R_j \vdash R_{j+1}$$

and either

(b')

$$\mathcal{a}_i : R_{k-1} \vdash R_k$$

and

$$\mathcal{a}_i : R_k \sqsupset$$

or (b'')

$$\mathcal{a}_i : R_{k-1} \vdash .R_k$$

If (b') then we write $\mathcal{a}_i : P \models R$

If (b'') then we write $\mathcal{a}_i : P \models .R$

In either case we say that the application of $\mathcal{a}_i$ to P yields R.

4.2 Examples of two elementary algorithms

Having formalized the structure and action of MAs we now give examples of the simple MAs required to evaluate simple integer functions.

We may, as in recursive function theory, reduce the work involved in specifying the more complex operations by first defining the two basic functions $\mathfrak{S}$ (the successor function on $\mathbb{N}$) and $\mathfrak{P}$ (the predecessor function on $\mathbb{N}^+$). This we do by the MAs $\mathfrak{B}_1$ and $\mathfrak{B}_2$ in figs 4.1 and 4.2.

Fig. 4.1 $\mathcal{B}_1$ ($\mathfrak{S}$ on $\mathbb{N}$)

1)	$\alpha\xi$	$\to$	$\xi\alpha$	$(\xi \in \mathbb{N}_1)$
2)	$\alpha\eta$	$\to$	$.\Omega.$	$(\eta \notin \mathbb{N}_1)$
3)	α	$\to$	β	
4)	0β	$\to$	$.1$	
	1β	$\to$	$.2$	
	2β	$\to$	$.3$	
	3β	$\to$	$.4$	
	4β	$\to$	$.5$	
	5β	$\to$	$.6$	
	6β	$\to$	$.7$	
	7β	$\to$	$.8$	
	8β	$\to$	$.9$	
5)	9β	$\to$	$\beta 0$	
6)	β	$\to$	$.1$	
7)	Λ	$\to$	α	

Fig. 4.2 $\mathcal{B}_2$ ($\mathcal{P}$ on $\mathbb{N}^+$)

1)	$\bar{\alpha}\xi$	$\to$	$\xi\bar{\alpha}$	$(\xi \in \mathbb{N}_1)$
2)	$\bar{\alpha}\eta$	$\to$	$.\Omega.$	$(\eta \notin \mathbb{N}_1)$
3)	$\alpha 0\bar{\alpha}$	$\to$	$.\Omega.$	
4)	$\alpha P\bar{\alpha}$	$\to$	$P\beta$	$(P \in W(\mathbb{N}_1))$
5)	1β	$\to$	$.0$	
	2β	$\to$	$.1$	
	.			
	.			
	.			
	.			
	8β	$\to$	$.7$	
	9β	$\to$	$.8$	
6)	0β	$\to$	$\beta 9$	
7)	Λ	$\to$	$\alpha\bar{\alpha}$	

These examples involve further notation which is now given, as is the formal definition of a total production.

Notn: Let $\mathbb{N}_1 = \{0,1,2,\ldots,8,9\}$

Note: No Greek letters will be allowed in the input alphabets of our MAs and that any algorithm will be defined over the input alphabet extended by a subset of G, where

$$G = \{\alpha,\beta,\gamma,\delta,\zeta,\eta,\theta,\lambda,\mu,\nu,\xi,\pi,\rho,\sigma,\tau,\chi,\psi,\omega\} \cup \bar{G}$$

with

$$\bar{G} = \{\bar{g} : g \in G\}$$

and write $\overset{2}{\alpha}$ for $\bar{\bar{\alpha}}$, $\overset{3}{\alpha}$ for $\bar{\bar{\bar{\alpha}}}$ etc.

The characters θ,ψ,χ have each a special significance which will be described in §4.4 and §4.5.

Notn: We sometimes use an abbreviated notation of the form:

$$P\xi Q \rightarrow (.)R \qquad (\xi \in S)$$

to represent, in any required order, the set of productions derived from the subalphabet S, when the cardinality of S, $|S|$ is such that

(a) $|S| = n : n \in \mathbb{N}$

or (b) $|A \setminus S| = n : n \in \mathbb{N}$ and A is the alphabet of definition for the current MA

or (c) $|S| = \aleph_0$. i.e. S is bijective with $\mathbb{N}$.

Defn: We define the total production:-

$$\gamma \rightarrow .P.$$

to be equivalent to the schema.

i) $\bar{\delta}\xi \to \xi\bar{\delta}$ $\forall \xi \in A$ for suitable A

ii) $\delta P_1 \gamma P_2 \bar{\delta} \to .P$ $\left\{ \begin{array}{l} P_1, P_2 \in W(A) \\ P_1 \text{ specifies the leading occurrence of } \gamma \text{ in } P_1\gamma \end{array} \right.$

iii) $\Lambda \to \delta\bar{\delta}$

The corresponding definition of

$\gamma \to .P$ (i.e. simple and total)

is identical but with

ii)' $\delta P_1 \gamma P_2 \bar{\delta} \to P$ in place of ii)

With reference to $\mathfrak{B}_1$ (fig. 4.1)

Notice: (a) that the contraction of production (4) causes no ambiguity

(b) that production (2) is a total production

(c) that parameter type errors would usually not reach this stage, i.e. execution, because they would constitute syntax errors, however when we link values to identifiers (simple names) we will require the explicit type checks of the kind incorporated in $\mathfrak{B}_1$,

and (d) that 'Ω' is a special symbol which may be thought of as a (universal) error flag.

$\mathfrak{B}_2$ (fig 4.2) represents the function $\wp$ on the set $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$.
Since $n \in \mathbb{N} \Rightarrow n \geq 0$ we can have no negation function $\mathbb{N} \to \mathbb{N}$.

This concludes the introduction to MAs proper; in the next section we consider EMAs.

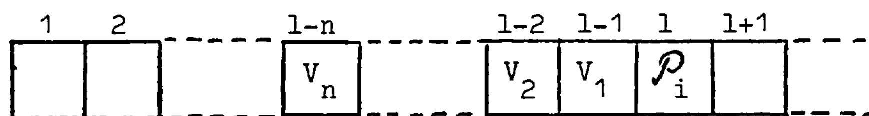
4.3 Extended Markov Algorithms (EMAs)

We now turn to our first diadic operator, the diadic plus. In Polish notation this would be denoted by xy+ or x, y, $\mathcal{P}_3$ (say).

We give a uniform technique for the manipulation of k-adic operators ($\forall\ k \geq 0$). Suppose that the triple given above occupies cells numbered

$$l-2,\ l-1,\ l$$

Formally we say that the function (here $\mathcal{P}_3$) acts on the preceding $l-1$ cells. To illustrate the manipulation more fully, consider the more general case with a list:-



Let $\mathcal{P}_i$ be an n-adic function yielding m values ($\mathcal{P}_i$ may be thought of as a vector function or as a subroutine).

(a) Input: strictly speaking each production in the schema for evaluating $\mathcal{P}_i$ is of the form:-

$$\underline{R} \rightarrow (.)\underline{Q}$$

where $\underline{R},\underline{Q}$ are vectors of length $l-1$, each co-ordinate of which is a word of the appropriate alphabet. We may illustrate the input mapping by fig. 4.3.

(b) Execution: since the operation $\mathcal{P}_i$ is n-adic all the components of any production

$$\underline{R} \rightarrow (.)\underline{Q}$$

of the schema for $\mathcal{P}_i$ which operate on co-ordinates $j : j > n$ may be thought of as:-

$$\Lambda \rightarrow \Lambda$$

Fig. 4.3

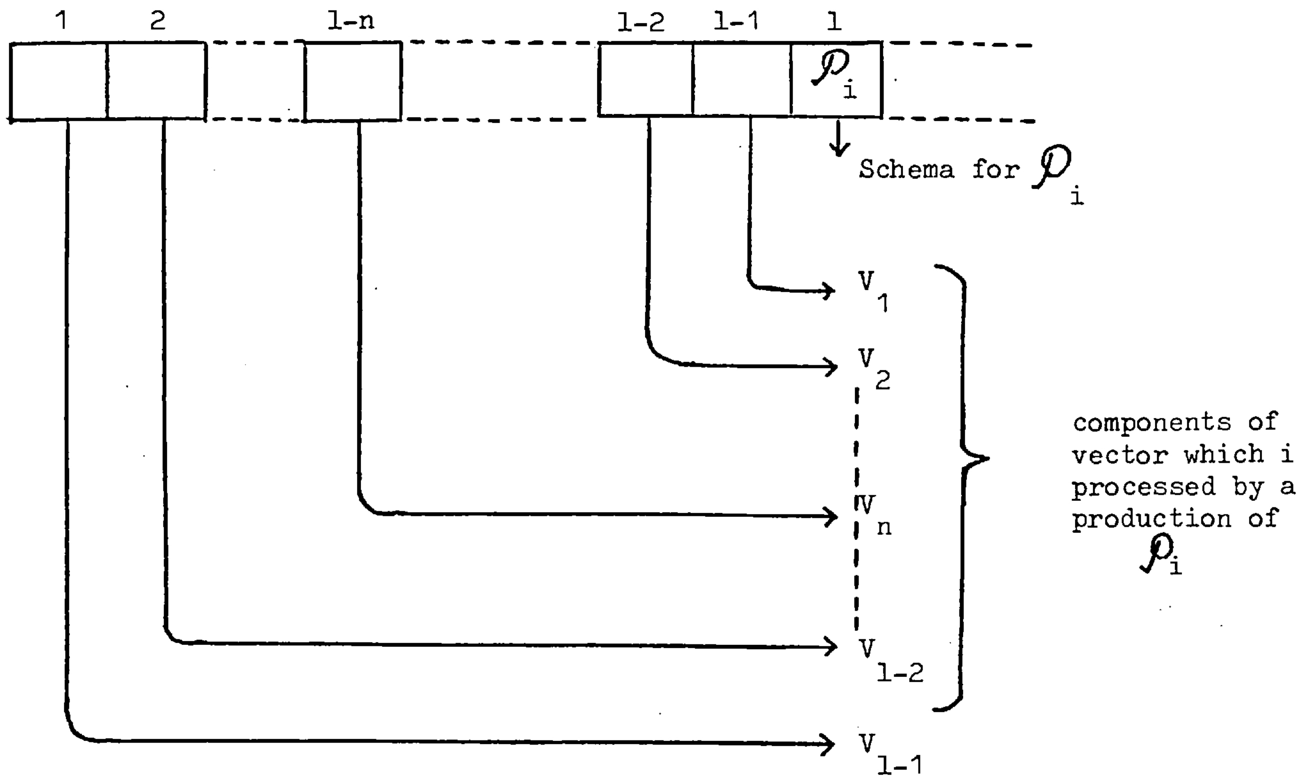
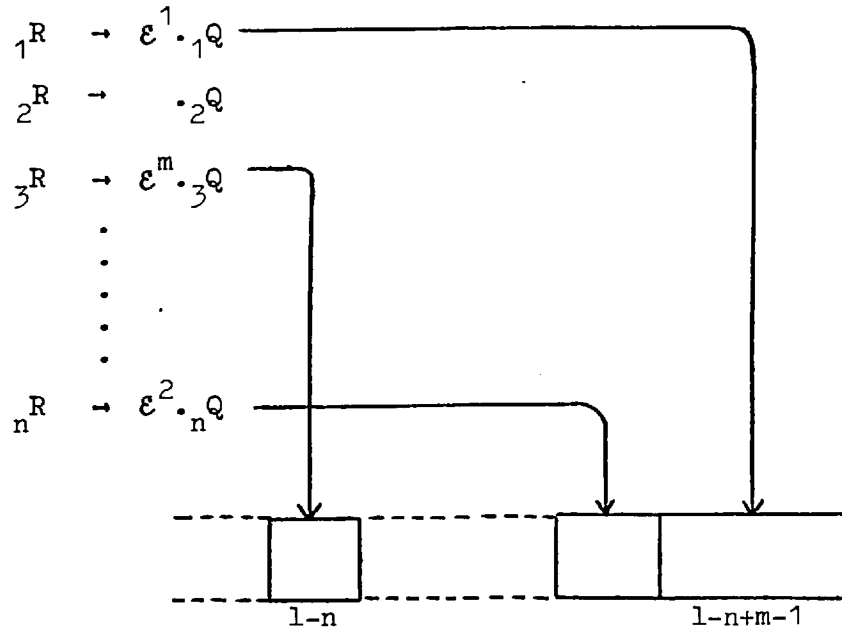


Fig. 4.4



and, because such a production is always applicable and has no effect, it is omitted from the schema. Any void co-ordinate production within the schema (i.e. acting on the jth co-ordinate where $1 \leq j \leq n$) must be shown but may be indicated by a long dash.

For a production (e.g.:-

$$\underline{R} \rightarrow (.)Q \equiv \begin{array}{l} {}_1R \rightarrow (.)_1Q \\ {}_2R \rightarrow (.)_2Q \\ \vdots \\ {}_nR \rightarrow (.)_nQ \end{array} \quad)$$

to be applied to a vector $\underline{V} = (V_1, \ldots, V_n)$ we require that

$${}_iR \circ V_i \quad \forall i : 1 \leq i \leq n$$

and that no earlier production is applicable. Execution stops when either no vector production is applicable or when every co-ordinate of an applied vector production is terminal.

(c) Output: when $n > 1$ and $m = 1$ we use the symbol $\mathcal{E}$ to indicate the exit value of the function: we do this by appending $\mathcal{E}$ to a co-ordinate of the terminal vector production e.g.

$$P \rightarrow \mathcal{E}.Q$$

When $m > 1$ we use $\mathcal{E}^i : (1 \leq i \leq m)$ in a similar way. The action of the output mapping is illustrated by the example in diagram in fig. 4.4.

To illustrate how these extended algorithms (EMAs) work we give, in fig. 4.5, the schema for a function, f, which takes as input three words in W(A), and combines them to produce two new words subject to the rule:-

$$f : (\bar{a}, \bar{b}, \bar{c}) \rightarrow (\bar{b}\bar{c}, \bar{a}\bar{b})$$

Fig. 4.5

The function f:

$$1)\ \begin{cases} \bar{\gamma}\xi \to \xi\bar{\gamma} & (\xi \in A) \\ \underline{\hspace{3cm}} & \\ \underline{\hspace{3cm}} & \end{cases}$$

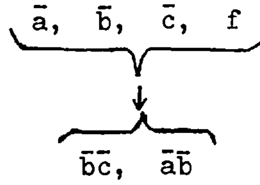
$$2)\ \begin{cases} \underline{\hspace{3cm}} & \\ \bar{\beta}\xi \to \xi\bar{\beta} & (\xi \in A) \\ \underline{\hspace{3cm}} & \end{cases}$$

$$3)\ \begin{cases} \underline{\hspace{3cm}} & \\ \underline{\hspace{3cm}} & \\ \bar{\alpha}\xi \to \xi\bar{\alpha} & (\xi \in A) \end{cases}$$

$$4)\ \begin{cases} \gamma P\bar{\gamma} \to .\Lambda & \\ \beta Q\bar{\beta} \to \varepsilon^{2}.QP & (R,\ P,\ Q \in W(A)) \\ \alpha R\bar{\alpha} \to \varepsilon^{1}.RQ & \end{cases}$$

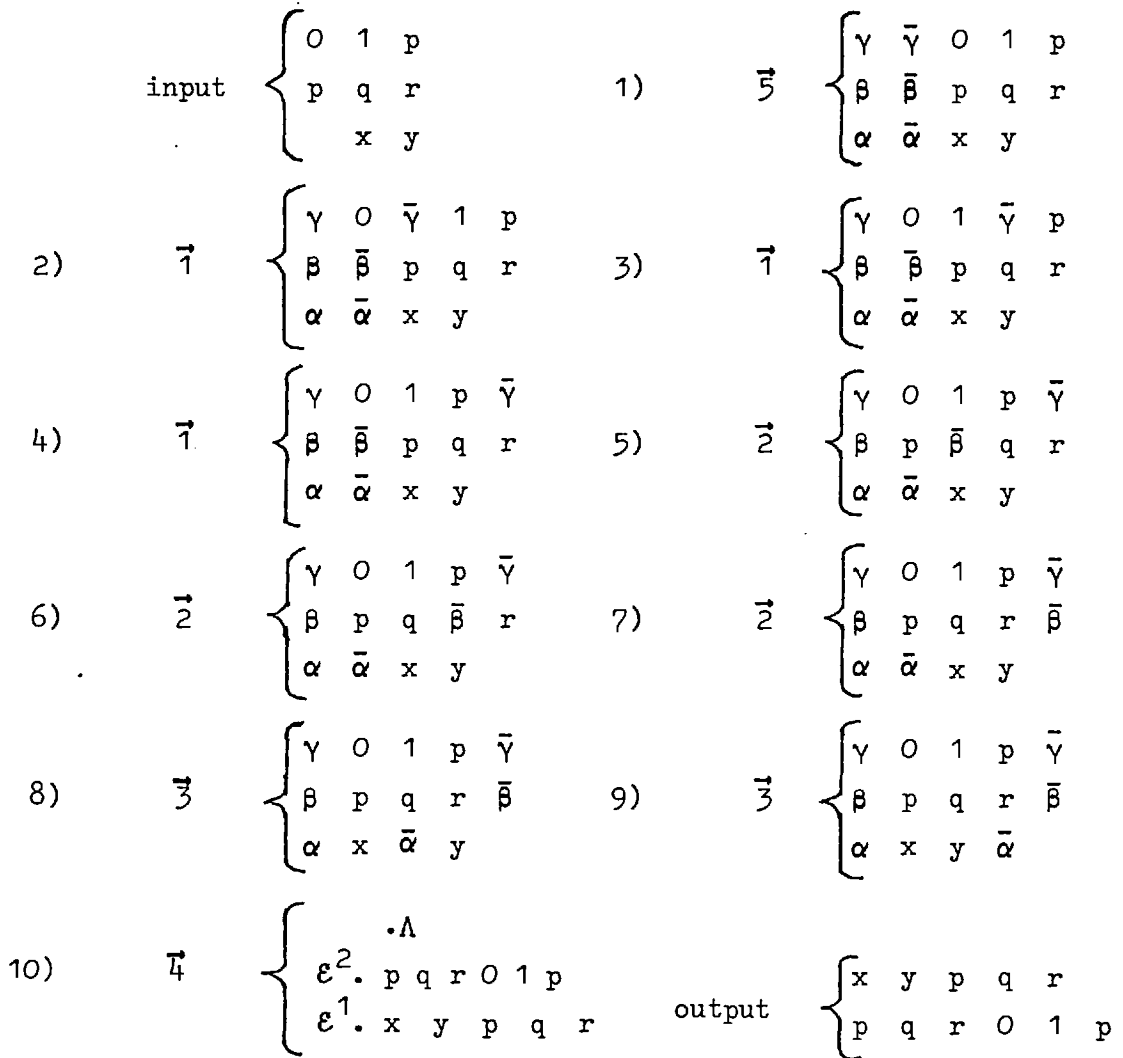
$$5)\ \begin{cases} \Lambda \to \gamma\bar{\gamma} \\ \Lambda \to \beta\bar{\beta} \\ \Lambda \to \alpha\bar{\alpha} \end{cases}$$

equivalently, in terms of lists



To exemplify further, we trace the execution of . f with input list

xy, pqr, 01p, f



Notice also, that if we require work space during the execution of an algorithm, or we have a function f':

$$f' : X^n \to X^m \qquad m > n$$

extra components can be added to the vectors with indices < 0

i.e.

$$\begin{array}{l} V_{-3} \\ V_{-2} \\ V_{-1} \\ \cdot \\ \cdot \\ \cdot \\ V_n \end{array}$$

Then we may initialize these cells to Λ and henceforth treat them in the same way as before, keeping the same input mapping technique so as not to interfere with these cells.

4.4 Composition of Algorithms and Embedding

4.4.1 Sequential Composition

To formalize the notions of composing and nesting MAs and EMAs (or more correctly their schemata, though we usually ignore this distinction) we need more notation.

Defn: Given algorithms $\mathcal{A}, \mathcal{B}$ over $A \in Ab$, then if

$$\mathcal{A}(P) = \mathcal{B}(P) \quad \forall P \in W(A)$$

[inclusive of the case when neither is applicable to P and hence $\mathcal{A}(P) = P$] we say that $\mathcal{A}, \mathcal{B}$ are fully equivalent, and write $\mathcal{A} \approx \mathcal{B}$.

Defn: A normal algorithm $\mathcal{A}_i$ is closed iff $\exists$ a production of the form:

$$\mathcal{A}_{i(j)} \equiv \Lambda \to .Q$$

Given any arbitrary MA, $\mathcal{A}$, let $\mathcal{A}\cdot$ denote the closure of $\mathcal{A}$ formed by adding the production

$$\Lambda \to \cdot\Lambda$$

to the end of the schema for $\mathcal{A}$.

Trivially

$$\mathcal{A}\cdot \approx \mathcal{A}$$

Now we consider the problem of defining the composition

$$\mathcal{B} \circ \mathcal{A}$$

of the MAs, $\mathcal{A}$ (over $A \in Ab$) and $\mathcal{B}$ (over $B \in Ab$). We usually have the case when $A \cap B \neq \emptyset$ or even $A = B$ and in order to deal with these cases we define a special procedure which creates an isomorphic copy of the required alphabet each time the MA is activated.

For any given $i \in \mathbb{N}$ and $A \in Ab$, define

$$C_i : A \to A^i$$

where A^i is the Ith IM copy of A and

$$C_i : a \to a_i \quad (a \in A)$$

[The stimulus behind this is the requirement to attach a suffix to every letter, except $\to$ and . used in any MA; whenever a new MA is 'called' we increment the suffix of the current string and of the new schema by one.]

Defn: An algorithm acting on an alphabet subscripted by i is said to act at level i.

C_i can be defined by the abbreviated schema

$$\xi \rightarrow \xi_i \qquad (\xi \in A)$$

$$\Lambda \rightarrow \cdot\Lambda$$

Obviously C_i is bijective and if we denote its inverse by C_i^{-1} then the transfer function required when entering a nested schema is the following:

$$T_{i,j} \approx C_j \circ C_i^{-1}$$

and for exit:

$$T_{i,j}^{-1} \approx C_i \circ C_j^{-1} \approx T_{j,i}$$

At this stage it may seem that we are producing a circular construction by attempting to achieve composition by defining new functions which themselves have to be composed; however, as will be seen, the transition functions behave differently from arbitrary MAs and their composition is more easily defined.

We refer to C_i and $T_{i,j}$ as <u>subalgorithms</u>; these never stand alone nor do they have input or output in the usual sense; they are used to formalize change of alphabets within algorithmic schemata.

Since we have not yet defined the symbol 'o' between MAs, the preceding equivalence for $T_{i,j}$ is vacuous; we give a schematic definition:-

$T_{i,j}$ $(i < j)$

1) $\xi_i \theta_i \rightarrow \theta_i \xi_i$
2) $\theta_i \xi_i \rightarrow \xi_j \theta_i$
3) $\theta_i \rightarrow \Lambda_j$

where $\xi \in A$, $\theta \notin A$ and A is the union of the alphabets of the component algorithms in the composition schema (see below)

$T_{i,j}$ $(i > j)$

$$\begin{array}{lll} 1) & \xi_i \psi_i \rightarrow \psi_i \xi_i & \text{where} \\ 2) & \psi_i \xi_i \rightarrow \xi_j \psi_i & \xi \in A, \psi \notin A \\ 3) & \psi_i \rightarrow \Lambda_j & \end{array}$$

$\dot{T}_{i,j}$ is as $T_{i,j}$ $(i > j)$ but with production 3) replaced by

$$3') \ \psi_i \rightarrow .\Lambda_j$$

N.B. We even index the empty word and require that

$$\Lambda_i \not\subset P_j \quad \text{if} \quad i \neq j \qquad \forall\ P \in W(A)$$

N.B. In the schemata for T, ξ is a dummy variable but θ, ψ are not; they are linkage markers and are used as illustrated in an example given below.

By their nature it is impossible to compose arbitrary MAs sequentially; what we do is define a new schema which embodies isomorphic (slightly modified but still isomorphic) copies of the constituent schemata which is fully equivalent to their sequential evaluation.

Given algorithms $\mathfrak{a}$ on A and $\mathcal{B}$ on B , we use the functions C_i and C_{i+1} to give:

$$\bar{\mathfrak{a}}^i \stackrel{def}{=} \mathfrak{a}\cdot \quad \text{on } A^i$$

$$\bar{\mathcal{B}}^{i+1} \stackrel{def}{=} \mathcal{B}\cdot \quad \text{on } B^{i+1}$$

We are going to define composition iteratively in pairs. The next modification to the schemata is to ensure the correct order of evaluation; if we are currently at level i then $\bar{\mathcal{B}}^{i+1}$ cannot be executed and any combination would terminate on the completion of $\bar{\mathfrak{a}}^i$.

We replace all termination 'dots' of the schema

$$\bar{\mathcal{A}}^{i} \text{ by } \theta_i$$

Call this the θ_i termination of $\bar{\mathcal{A}}^{i}$ and write it as:

$$\bar{\mathcal{A}}^{i}_{\theta_i}$$

Similarly for $\mathcal{B}$ form;

$$\bar{\mathcal{B}}^{i+1}_{\psi_{i+1}}$$

Now define $\mathcal{B} \circ \mathcal{A}$ at level i by the schema given below, which consists of four ordered sets of productions.

1) $T_{i,\ i+1}$

2) $\dot{T}_{i+1,\ i}$

3) $\bar{\mathcal{A}}^{i}_{\theta_i}$

4) $\bar{\mathcal{B}}^{i+1}_{\psi_{i+1}}$

Multiple composition is defined by the following:

if $\mathcal{A}, \mathcal{B}, \mathcal{C}$, are MAs, then so is

$$\mathcal{A} \circ \mathcal{B} \circ \mathcal{C}$$

where $\mathcal{A} \circ \mathcal{B} \circ \mathcal{C} = ((\mathcal{A} \circ \mathcal{B}) \circ \mathcal{C})$

At level 6 this yields the schema:-

1) $T_{6,7}$

2) $\dot{T}_{7,6}$

3) $\overline{\mathcal{A} \circ \mathcal{B}}^{7}_{\psi_7}$

4) $\mathcal{C}^{6}_{\theta_6}$

where $\mathcal{A} \circ \mathcal{B}$ at level 7 is:

1) $T_{7,8}$

2) $\dot{T}_{8,7}$

3) $\bar{\mathcal{A}}^{8}_{\psi_8}$

4) $\bar{\mathcal{B}}^{7}_{\theta_7}$

The derived schema may be simplified by applying the reduction rule:-

$$\left.\begin{matrix} \dot{T}_{i,j} \\ \\ \dot{T}_{j,k} \end{matrix}\right\} = \dot{T}_{i,k} \qquad \text{where} \quad i>j>k$$

in such a way as to minimize the total number of T's in the resultant schema.

Defn: Consistent with our aim of trying to simplify notation and since we are justified in introducing any abbreviations provided only that we have a well-defined procedure for expanding them uniquely into an EMA; we denote a sequence of (sub) algorithms as in the preceding example by:-

$$\mathcal{A}_i : \mathcal{A}_j : \mathcal{A}_k : \ldots : \mathcal{A}_n$$

Defn: Now we can define the composition of $\mathcal{A}$ and $\mathcal{B}$ level i by:-

$$\overline{\mathcal{B} \circ \mathcal{A}}^{i} \overset{\text{def}}{=} T_{i,i+1} : \dot{T}_{i+1,i} : \bar{\mathcal{A}}^{i}_{\theta_i} : \bar{\mathcal{B}}^{i+1}_{\psi_{i+1}}$$

where $\mathcal{A}$ is simple.

An algorithm $\mathcal{A}$ is simple iff

$$\mathcal{A} \neq \mathcal{P} \circ \mathcal{Q}$$

for any non-trivial $\mathcal{P}$, $\mathcal{Q}$. The definition thus covers all multiple compositions, since:-

$$\overline{\mathcal{C} \circ \mathcal{B} \circ \mathcal{A}}^{i} = T_{i,i+1} : \dot{T}_{i+1,i} : \bar{\mathcal{A}}^{i}_{\theta_i} : \overline{\mathcal{C} \circ \mathcal{B}}^{i+1}_{\psi_{i+1}}$$

where

$$\overline{\mathcal{C} \circ \mathcal{B}}^{i+1} = T_{i+1,i+2} : \dot{T}_{i+2,i+1} : \bar{\mathcal{B}}^{i+1}_{\theta_{i+1}} : \bar{\mathcal{C}}^{i+2}_{\psi_{i+2}}$$

Examples of the mechanics of sequential combination are given at length elsewhere [23].

4.4.2 Embedding

Having defined the functions $\mathcal{S}$ and $\mathcal{P}$ on $\mathbb{N}$ and $\mathbb{N}^+$ by the schemata $\mathcal{B}_1$ and $\mathcal{B}_2$ (given in figs. 4.1 and 4.2) we wish to use these to calculate 'x+y' by increasing x and reducing y until y=0; the current value of x is then the required sum. To do this we will need to achieve the effect of embedding one EMA within another, i.e. to have productions of the form:-

$$P \rightarrow (.)Q\,\mathcal{A}(n)R \quad \text{where } \mathcal{A} \text{ is an EMA}$$

Without formality, we dictate that when such a production is invoked we evaluate the embedded schemata ($\mathcal{A}$) on the given input (n) and then replace $\mathcal{A}(n)$ with the result: the evaluation of $\mathcal{A}$ being carried out separately from the rest of the computation, i.e. we do not embed the schema for $\mathcal{A}$, only a 'call' to $\mathcal{A}$ which is denoted by the occurrence of $\mathcal{A}$.

Now: z = (x+y) = (a) <u>if</u> (x=0) <u>then</u> z:= y
<u>else</u> (x := x-1;
y := y+1;
<u>goto</u> (a))

From this definition we may extract an MA, $\mathcal{B}_3$ given in fig. 4.6

<u>Fig. 4.6</u>

$\mathcal{B}_3$, addition on $\mathbb{N}$

$$
\begin{array}{ll}
1) & \left\{\begin{array}{lcl} ____ & & \\ \bar{\alpha}\xi & \rightarrow & \xi\bar{\alpha} \end{array}\right. \\
2) & \left\{\begin{array}{lcl} \bar{\beta}\xi & \rightarrow & \xi\bar{\beta} \\ ____ & & \end{array}\right.
\end{array}
\qquad (\xi \in \mathbb{N}_1)
$$

$$
\begin{array}{ll}
3) & \left\{\begin{array}{lcl} ____ & & \\ \bar{\alpha}\eta & \rightarrow & \varepsilon.\Omega. \end{array}\right. \\
4) & \left\{\begin{array}{lcl} \bar{\beta}\eta & \rightarrow & \varepsilon.\Omega. \\ ____ & & \end{array}\right.
\end{array}
\qquad (\eta \notin \mathbb{N}_1)
$$

$$
\begin{array}{ll}
5) & \left\{\begin{array}{lcl} \beta O\bar{\beta} & \rightarrow & .O \\ \alpha P\bar{\alpha} & \rightarrow & \varepsilon.P \end{array}\right. \\
6) & \left\{\begin{array}{lcl} \beta Q\bar{\beta} & \rightarrow & \beta\mathcal{B}_2\,(Q)\bar{\beta} \\ \alpha P\bar{\alpha} & \rightarrow & \alpha\mathcal{B}_1\,(P)\bar{\alpha} \end{array}\right.
\end{array}
\qquad (P,\ Q \in W\ (\mathbb{N}_1\ ))
$$

$$
7) \quad \left\{\begin{array}{lcl} \Lambda & \rightarrow & \beta\bar{\beta} \\ \Lambda & \rightarrow & \alpha\bar{\alpha} \end{array}\right.
$$

Proceeding in this way the other basic arithmetic functions can be defined [23], however it is easier to appreciate what is happening within an (E)MA if we define only the most primitive operations by such means; using higher-level operations to determine which sequences of basic operations should be executed next. These are fully defined (via MAs) in §4.5.1 and, in §4.5.2 we use them to specify the fundamental arithmetic operations.

4.5 Iteration and Ramification

In this section we give two fundamental control routines. These model the Algol-60 constructs:-

if ... **then** ... **else**

and

label: ... **if** ... **then** **goto** label; - and

may be defined in terms of MAs as we show in §4.5.1. Also, one of the operations may be equivalently defined in terms of S as in §6.

In §4.5.2 these control routines are used to define some of the less primitive arithmetic operations.

4.5.1 Formal derivation of RAM and PTL.

We wish to define the ramification of processes α and β by the predicate γ:-

i.e. **if** γ **then** α **else** β

- and the iteration of α controlled by the predicate γ :-

label: α ; **if** γ **then** **goto** label.

In order to simplify the exposition we use a continuous indexing scheme for denotation of levels, instead of using a system based on

primes (cf Gödel). This means that the constructions given below are technically incorrect. However, if (at the stages marked by an asterisk) we replace the construction by an equivalent algorithm which acts at only one level, then the construction is valid.

We now give the constructions. A commentary, to aid comprehension of the assembly, is included at the end of this section.

Defn: If $\mathcal{D}$ is a MA on A and $A \subset B$, prefix the schema for $\mathcal{D}$ with

$$b \rightarrow b \quad (b \in B \setminus A)$$

Call the new schema $\mathcal{D}_B$.

Then $\mathcal{D}_B(P) \approx \mathcal{D}(P) \quad \forall\, P \in W(A)$

$\mathcal{D}_B$. is called the propagation of $\mathcal{D}$ onto B

Defn: Given $\mathcal{A}$ on A and $\mathcal{B}$ on B with $A \cup B = C$. Let $\mathcal{A}_C$ and $\mathcal{B}_C$ be the propagations of $\mathcal{A}$ and $\mathcal{B}$ onto C.

Then $\mathcal{B}_C \circ \mathcal{A}_C$ is the normal composition of $\mathcal{A}$ and $\mathcal{B}$, and is written $\mathcal{B} \circ \mathcal{A}$.

In what follows we shall pay no particular attention to the compatability of alphabets; any inconsistency may be easily removed by using appropriate propagations.

Defn: Given $A, B \in Ab$ with $A \subset B$

Then

$$\Pi_B^A : \xi \rightarrow \Lambda \quad (\xi \in B \setminus A)$$

is the projection of B onto A.

Defn: A predicate is a MA which yields (depending on its data) either the logic value T or the logic value F.

Defn: A void predicate is formed by the composition of the MA ν with a predicate, where

$$\nu : \begin{cases} T \to .\Lambda \\ \Lambda \to .\Lambda \end{cases}$$

i.e. if $\mathscr{C}_1 = \nu \circ \mathscr{C}$ where $\mathscr{C}$ is a predicate, then

$$\mathscr{C}(P) = T \Rightarrow \mathscr{C}_1(P) = \Lambda$$

$$\mathscr{C}(P) = F \Rightarrow \mathscr{C}_1(P) = F \ (\neq \Lambda)$$

Defn: We define the n-tuple algorithm at level i

$$T_n : P_i \to P_{i+1}\, P_{i+2} \cdots P_{i+n} \ : P \in W(A)$$

- by the schema of fig. 4.7

Trivially its inverse T_n^{-1} is defined by the abbreviated schema:

$$T_n^{-1} \text{ (at i)} : \xi_j \to \xi_i \quad i<j\leq i+n$$

Fig. 4.7

T_n:

1) $\alpha \xi_i \to \xi_{i+1}\, \xi_{i+2} \cdots \xi_{i+n}\, \alpha \quad (\xi \in A)$

2) $\alpha \to \Lambda$

3) $\xi_l \eta_j \to \eta_j\, \xi_l \qquad (i<j<l\leq n) \quad (\xi, \eta \in A)$

4) $\Lambda \to \alpha$

Defn: Using the tupling algorithms T_n we form the juxtaposition of a set of MAs

$$\mathfrak{a}_i : 1 \leq i \leq n \quad \text{by}$$

$$\mathscr{B} = T_{i+n,i} : T_{i+n-1,i} \cdots : T_{i+2,i} : T_{i+1,i} : \bar{\mathfrak{a}}_{1_{\psi_{i+n}}}^{i+n} : \cdots : \bar{\mathfrak{a}}_{n_{\psi_{i+1}}}^{i+1} : T_n$$

and write

$$\mathscr{B} = \mathfrak{a}_1\, \mathfrak{a}_2 \cdots \mathfrak{a}_n$$

The major operation at which we are aiming is the ramification of two MAs governed by a third (void predicate) MA. In the construction of a ramification we use χ as a selector flag and develop the concept by means of two lemmas.

Lemma 1: Given a void predicate $\mathcal{G}$ over A then $\exists$ an algorithm $\mathcal{D}$ over $A \cup \{\chi\} = B$, such that:

$$\mathcal{D}(P) \approx \begin{cases} \chi P & \text{if } P \in W(A) \text{ and } \underline{e}\,\mathcal{G}(P) = \Lambda \\ P & \text{if } P \in W(A) \text{ and } \underline{e}\,\mathcal{G}(P) \neq \Lambda \end{cases}$$

(For a description of e see §6.3. Informally $\underline{e}\,\mathcal{G}(P)$ means the result of $\mathcal{G}$ on P.)

Construction: Take $\beta \notin B$ and let $C = B \cup \{\beta\}$. Define $\mathfrak{H}_1$ over C by the schema

1) $a \rightarrow \beta \quad (a \in B)$

2) $\beta^2 \rightarrow \beta$

3) $\beta \rightarrow .\Lambda$

4) $\Lambda \rightarrow .\chi$

(*) set $\mathfrak{H}_2 = \mathfrak{H}_1 \circ \mathcal{G}$

Now if $\mathcal{I}$ is the identity function defined by the schema

$$\Lambda \rightarrow .\Lambda$$

Then:

$$\mathcal{D} = \mathfrak{H}_2 \mathcal{I}$$

Lemma 2: Given MAs (*)$\mathcal{A}$, (*)$\mathcal{B}$ on A at level i and $\chi \notin A$ then $\exists \mathcal{G}$, an MA over $A \cup \{\chi\}$ such that:

$$\left.\begin{array}{l} \mathcal{G}(\chi P) \approx \mathcal{A}(P) \\ \mathcal{G}(P) \approx \mathcal{B}(P) \end{array}\right\} \quad P \in W(A)$$

Construction: is defined by the schema:

1) $\chi \to \theta_i$

2) $T_{i,i+1} : \dot{T}_{i+1,i} : \bar{\mathcal{a}}^{i+1}_{\psi_{i+1}} : \bar{\mathcal{B}}^i$

Defn: Taking (*)$\mathcal{G}$ and (*)$\mathcal{D}$ as in the Lemmas and setting

$$\mathcal{F} = \mathcal{G} \circ \mathcal{D}$$

then $\mathcal{F}$ is called the ramification of $\mathcal{a}$ and $\mathcal{B}$ governed by $\mathcal{C}$.

i.e.

$$\mathcal{F}(P) \approx \begin{cases} \mathcal{a}(P) \text{ if } P \in W(A) \text{ and } \underline{e}\,\mathcal{C}(P) = \Lambda \\ \mathcal{B}(P) \text{ if } P \in W(A) \text{ and } \underline{e}\,\mathcal{C}(P) \neq \Lambda \end{cases}$$

Defn:† Using the derivation of $\mathcal{F}$ and regarding P as the common (or total) data for $\mathcal{a}$, $\mathcal{B}$ and $\mathcal{C}$ (via propagations) we define:

$$\text{RAM } (C, A, B) \approx \mathcal{F}$$

where P is understood.

Defn: If we now set $\mathcal{B} = \mathcal{I}$ and denote the new ramification by $\mathcal{F}_o$, then:

$$\mathcal{F}_o(P) = \begin{cases} \mathcal{a}(P) \text{ if } P \in W(A) \text{ and } \underline{e}\,\mathcal{C}(P) = \Lambda \\ P \text{ if } P \in W(A) \text{ and } \underline{e}\,\mathcal{C}(P) \neq \Lambda \end{cases}$$

In this case we say that $\mathcal{F}_o$ is equivalent to $\mathcal{a}$ controlled by the (void) predicate (*)$\mathcal{C}$

i.e.

$$\mathcal{F}_o(P) := \text{ if } \underline{e}\,\mathcal{C}(P) = \Lambda \text{ \underline{then} } \mathcal{a}(P) \text{ \underline{else} } P$$

Defn: If for a given $P_o \in W(A)$ we have a sequence $P_1, P_2, P_3, \ldots, P_n$ with $n \in \mathbb{N}$ such that

---

† This is not the same as in [23].

$$\begin{array}{ll}
\underline{e}\,\mathcal{A}(P_0) = P_1 & \underline{e}\,\mathcal{B}(P_1) = \Lambda \\
\underline{e}\,\mathcal{A}(P_1) = P_2 & \underline{e}\,\mathcal{C}(P_2) = \Lambda \\
\underline{e}\,\mathcal{A}(P_2) = P_3 & \underline{e}\,\mathcal{C}(P_3) = \Lambda \\
\vdots & \vdots \\
\underline{e}\,\mathcal{A}(P_{n-2}) = P_{n-1} & \underline{e}\,\mathcal{C}(P_{n-1}) = \Lambda \\
\underline{e}\,\mathcal{A}(P_{n-1}) = P_n & \underline{e}\,\mathcal{C}(P_n) \neq \Lambda
\end{array}$$

then this may be viewed as the <u>iteration</u> of $\mathcal{A}$ controlled by the (void) predicate $\mathcal{C}$. We now formalize such an iterative scheme:-

Given a MA (*)$\mathcal{A}$ and a void predicate $\mathcal{C}$ over A, then

$$\mathcal{I}_{\mathcal{A},\mathcal{C}}$$

the iteration of $\mathcal{A}$ controlled by $\mathcal{C}$ is an MA; we give its construction:-

Take $\chi \notin A$ and let $B = A \cup \{\chi\}$ then $\exists\,\mathcal{D}$ over B such that:

$$\mathcal{D}(P) = \begin{cases} \chi P & \text{if } P \in W(A) \text{ and } \underline{e}\,\mathcal{C}(P) = \Lambda \\ P & \text{if } P \in W(A) \text{ and } \underline{e}\,\mathcal{C}(P) \neq \Lambda \end{cases}$$

let $\mathcal{F} = \mathcal{D} \circ \mathcal{A}$ and (*)$\mathcal{F}$ is in $F \supset B$, form the linked closure of $\mathcal{F}$, $\bar{\mathcal{F}}_{\bar{\chi}}$ where $\bar{\chi} \notin F$. Define $\mathcal{G}$ by the schema:

1) $\xi\bar{\chi} \to \bar{\chi}\xi \qquad (\xi \in F)$

2) $\bar{\chi}\chi \to \Lambda$

3) $\bar{\chi} \to .\chi$

4) $\bar{\mathcal{F}}_{\bar{\chi}}$

Then $$\mathcal{I}_{\mathcal{A},\mathcal{C}} = \Pi_F^{F \setminus \{\chi\}} \circ \mathcal{G}$$

<u>Defn</u>: From the algorithm $\mathcal{I}_{\mathcal{A},\mathcal{C}}$ we define

PTL(A, C)

i.e. Process A (given by schema $\mathcal{A}$), Test the predicate C (evaluated by $\mathcal{C}$) and if True repeat processing from A, otherwise exit.

Commentary:

RAM:- For input x; form xx

Process the leading x by $\#_2$: $\#_2(x) = \chi$ if $\mathcal{C}(x) = \Lambda$ (i.e. $C(x) \Rightarrow T$).

This gives $\mathcal{D}(x) = \chi x$

Then $\mathcal{G} \circ \mathcal{D}(x) = \mathcal{A}(x)$ where $\mathcal{A}$ evaluates A

Alternatively $\mathcal{C} \neq \Lambda$ so $\mathcal{D}(x) = x$

and $\mathcal{G} \circ \mathcal{D}(x) = \mathcal{B}(x)$

PTL:- Given x do $\mathcal{A}(x)$ giving result y, if $C(y) = $ True then $\mathcal{D}(y) = \chi y$ otherwise $\mathcal{D}(y) = y$

So if $C(y) = $ True

then $\mathcal{F}_{\bar{\chi}}(x)$ contains $\bar{\chi}$ and χ hence by $\mathcal{G}$ we get $\bar{\chi}\chi\alpha$ (say)

and $\bar{\chi}\chi\alpha \mapsto \alpha$ and $\mathcal{F}_{\bar{\chi}}$ is repeated.

Otherwise, if $C(y) = $ False

then $\mathcal{F}_{\bar{\chi}}(x)$ does not contain χ

hence by $\mathcal{G}$ $\bar{\chi} \to \cdot\chi$

and we get $\chi\beta$ (say) on exit.

Finally $\Pi_F^{F \setminus \{\chi\}}$ erases χ giving the required result.

4.5.2 Examples:

Given $\mathcal{B}_1$ ($\mathcal{S}$ on $\mathbb{N}$) and $\mathcal{B}_2$ ($\mathcal{P}$ on $\mathbb{N}$) then $\mathcal{B}_3$ (addition on $\mathbb{N}$) may be redefined by:-

$$\mathcal{B}_3(x,y) \equiv \text{RAM}\ (x = o,\ y,\ (\text{PTL}((x := \mathcal{B}_2(x),\ y := \mathcal{B}_1(y)),\ (x \neq o)),\ y))$$

Similarly, $\mathcal{B}_4$ (multiplication on $\mathbb{N}^+$)

$$\mathcal{B}_4(x,y) \equiv \text{RAM}\ (x = 1,\ y,\ (\ z := y,\ \text{PTL}(\ (\ x := \mathcal{B}_2(x),\ z := \mathcal{B}_3(y,\ z)),\ (x \neq 1))\ z))$$

Explicit details of the order evaluation through control constructs are given in chapter 6.

CHAPTER 5

THE PROGRAM SPACE AND THE OPERATOR k

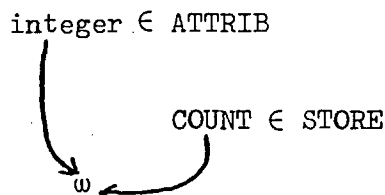
5.1 Informal Discussion

In the Carabiner Program Space (CPS) we represent all directly addressable components of a computing system as elements of a list which we call the STORE. These elements are referred to as Key nodes.

Another list, called ATTRIButes, holds the (abstract) properties which may be used to characterize elements of the system.

Other components of the system must be linked in some way to STORE, and it is via these key nodes in STORE, that they are accessed.

As a trivial example consider an Algol-60 identifier, COUNT, which denotes an integer quantity. We wish to represent this information by the relations:-

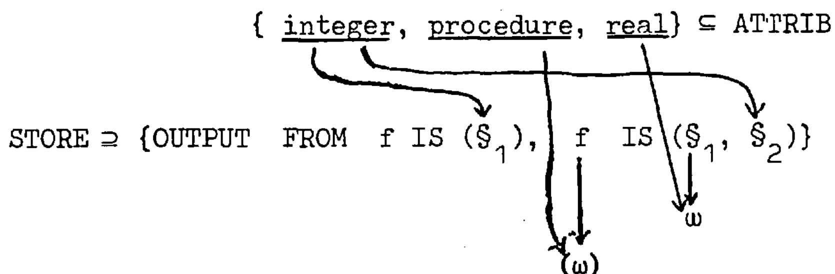


- where 'ω' is used to denote an unassigned value.

A more complex example is given by the following (Algol-60) procedure declaration heading:-

<u>integer</u> <u>procedure</u> f(x,y); <u>real</u> x;
<u>integer</u> y;
<u>value</u> y;

This may be represented thus:-



The meaning of 'OUTPUT FROM', 'IS' and ($§_i$) will be explained in depth in subsequent sections; however, informally:-

> 'OUTPUT FROM f' denotes the output from f,
>
> 'IS' is a list definition operator, and
>
> '$§_i$' denotes the ith element in a list

Also it may be noted that a function is considered as a pair; the name of the function being identified with its input (formal) parameter list and the body, 'ω' in the present example since we only know about the procedure head, which is 'possessed' by the name. We shall formalise these relations when we have defined the CPS rigorously.

5.2 Some Mathematical Definitions

5.2.1 Lists: Our notation for lists is the usual one [45] but the operations defined are tailored to meet our special requirements. The definitions given below are based on set-theory but we note that they can all be specified by Extended Markov Algorithms as in Chapter 4 or by S as in Chapter 6.

Defn: A (double-ended) pseudo-list is a finite set, L, that is well ordered† [72].

i.e. there is an associated order relation, which we write $\underset{L}{<}$, such that if

$$x, y \in L$$

and

$$x \neq y$$

then

$$x \underset{L}{<} y \quad \text{or} \quad x \underset{L}{>} y \quad \text{(not both).}$$

Moreover, since L is finite, there exists an element O_1 of $L : O_1 \underset{L}{<} x \quad \forall x \in L \setminus \{O_1\}$.

Also, if the cardinality of L, $|L| = n \in \mathbb{N}$, then there is an element O_n of L, such that

$$O_n \underset{L}{>} x \quad \forall x \in L \setminus \{O_n\} .$$

Selecting elements inductively by the rule

$$O_i : O_i \underset{L}{<} x \quad \forall x \in L \setminus \{O_1, O_2, \ldots, O_i\}$$

with $1 \leq i \leq n$, and $O_i \in L$

---

†By the 'linearity' of a computer's store any set held within the store has an implicit ordering.

or equivalently

$$O_j : O_j \underset{L}{>} x \quad \forall\, x \in L \setminus \{O_j, O_{j+1}, \ldots, O_n\}$$

with $1 \leq j \leq n$ and $O_j \in L$

we have:-

$$O_1 \underset{L}{<} O_2 \underset{L}{<} O_3 \underset{L}{<} \cdots \underset{L}{<} O_{n-1} \underset{L}{<} O_n \ .$$

Hence, we denote the set L by:-

$$(O_1, O_2, O_3, \ldots, O_{n-2}, O_{n-1}, O_n)$$

Defn: Since a pseudo-list, L, is a set, we use $\in$ to denote inclusion. i.e. $x \in L$ iff $L = (O_1, \ldots, x, \ldots, O_n)$

Defn: Given a pseudo-list, L, where

$$L = (O_1, O_2, \ldots, O_{n-1}, O_n)$$

we define the left-augmentation of L with an object x by the operation AUGL such that:

AUGL (L, x) : $L \mapsto (x, O_1, O_2, \ldots, O_n)$.

Similarly for right-augmentation:

AUGR (L, x) : $L \mapsto (O_1, O_2, \ldots, O_n, x)$.

Note that if M is a pseudo-list, then

AUGL (L, M) : $L \mapsto ((M), O_1, \ldots, O_n)$.

Also, to augment a pseudo-list by a set of elements, the order of which is irrelevant, we may use normal set notation, since no ordering is implied, so:-

AUGL (L, {a, b, c}) : $L \mapsto (a, b, c, O_1, \ldots, O_n)$

i.e.

AUGL (L, {a, b, c}) = AUGL (L, c)

AUGL (L, b)

AUGL (L, a)

but AUGL (L, (a, b, c)) : $L \mapsto ((a, b, c), O_1, \ldots, O_n)$

Notice that when using AUG's, the result may not be a pseudo-list; this leads to:-

Defn: A list is an element of the recursively defined set $\mathcal{L}$, where:

$\mathcal{L}$ = {l:l is a pseudo-list or

AUGL (L, x) : $L \mapsto l$ or

AUGR (L, x) : $L \mapsto l$

where $L \in \mathcal{L}$ and x is a set}

examples: (a) (x, y, x, x, y)

x is a pseudo-list

AUGL ((x), {x, y}) : $(x) \mapsto (x, y, x)$

AUGR ((x, y, x), {x, y}) : $(x, y, x) \mapsto (x, y, x, x, y)$

(b) (x, (x), y) is a pseudo-list.

Note that the elements of $\mathcal{L}$ are not at all uniquely defined, e.g. (a) above.

Defn: To (re)initialize a list we use the definitional operator IS.

e.g. L IS(a, b, c) .

To insert an element into a list, in a specified position, we use the following two operators:-

Defn: Given a list L , where

L IS $(O_1, O_2, \ldots, O_n)$

then INSERTL (L, O_i, p) : $L \mapsto (O_1, O_2, \ldots, p, O_i, \ldots, O_n)$

provided $O_i \notin (O_{i+1}, \ldots, O_n)$

and INSERTR (L, O_i, p) : $L \mapsto (O_1, \ldots, O_i, p, \ldots, O_n)$

provided $O_i \notin (O_1, \ldots, O_{i-1})$

These may be read: insert p into L to the immediate left of O_i,

and: insert p into L to the immediate right of O_i.

If $O_i \notin L$ then

INSERTL (L, O_i, p) $\equiv$ AUGL (L, p)

and

INSERTR (L, O_i, p) $\equiv$ AUGR (L, p).

Defn: Given a list L IS ($O_1, \ldots, O_n, x, p_1, \ldots, p_m$) we may duplicate a portion of L by the routines

e COPYR (L, x) = ($p_1, \ldots, p_m$) if $p_i \neq x$ $(1 \leq i \leq m)$

e COPYL (L, x) = ($O_1, \ldots, O_n$) if $O_i \neq x$ $(1 \leq i \leq n)$

The list L is unchanged.

Similarly, to copy the whole list we use COPY(L).

Notice: The copy routines define an object not a mapping, and e denotes activation of the routine (see Chapter 6).

Defn: To delete an element, x, of a list L, e.g.

L IS ($O_1, \ldots, O_n, x, p_1, \ldots, p_m, x, q_1, \ldots, q_e$)

– where $x \notin \{O_1, \ldots, O_n, q_1, \ldots, q_e\}$ –

we use the two operations:-

DELL (L, x): L $\rightarrow$ ($O_1, \ldots, O_n, p_1, \ldots, p_m, x, q_1, \ldots, q_e$)

DELR (L, x): L $\rightarrow$ ($O_1, \ldots, O_n, x, p_1, \ldots, p_m, q_1, \ldots, q_e$).

If $x \notin L$ then

$$DELR\ (L, x) : L \mapsto L$$

and

$$DELL\ (L, x) : L \mapsto L$$

DELR ≡ delete rightmost specified element

DELL ≡ delete leftmost specified element.

Defn: To delete portions of a list L, where

L IS $(O_1, \ldots, O_n, x, p_1, \ldots, p_m)$,

delimited by x we define:

$$TRIMR\ (L, x) : L \mapsto (O_1, \ldots, O_n) \quad \text{if } p_i \neq x \quad (1 \leq i \leq m)$$

$$TRIML\ (L, x) : L \mapsto (p_1, \ldots, p_m) \quad \text{if } O_i \neq x \quad (1 \leq i \leq n)$$

One final point with reference to inclusion in lists. By the construction of a list, an object of that list may occur in several different places,

e.g. (x, y, x).

Defn: We postulate a selection operator 'OF' which denotes the right-most entry of an object in a list;

So if L IS (x, y, x, z)

then:

x OF L denotes the 3rd object in L.

Moreover, we may use the concept of the formal parameter to yield an explicit selector function i.e.

$$\S_n \text{ OF } (O_1, \ldots, O_m) = O_n \quad \text{if } 1 \leq n \leq m$$
$$= \omega \quad \text{otherwise.}$$

<u>Defn</u>: Given a list L IS $(0_1, \ldots, 0_n)$

and i: $(1 \le i \le n)$

then $\S_i$ OF L yields an element of L.

Moreover, <u>Next</u> (L, $\S_i$) yields $\S_{i+1}$ OF L

<u>Prev</u> (L, $\S_i$) yields $\S_{i-1}$ OF L.

When these operations take extreme parameters i.e.

Next (L, $\S_n$)

Prev (L, $\S_1$)

then the results are defined to be:

$\omega : \omega$ is both an atom and an anti-atom (see §5.3.3).

5.2.2 Digraphs

As in §5.2.1 we presuppose a knowledge of elementary set theory and familiarity with the concept of a relation such as is to be found in most introductory texts in modern analysis (e.g. [80]).

<u>Defn</u>: A (finite) <u>graph</u> G consists of :

(a) a (finite) set of <u>points</u> (or <u>nodes</u>) P(G)

(b) a (finite) set of <u>lines</u> (or <u>edges</u>, or <u>links</u>) L(G)

such that:

$$L(G) \subset P(G) \times P(G)$$

<u>Defn</u>: A <u>relation</u> ρ, on a set A is a subset, A_ρ of $A \times A$

i.e. $y \in \rho\, x$ iff $(x, y) \in A_\rho$

<u>Defn</u>.(i) A <u>digraph</u> D is a graph G,

such that $(x, y) \in L(G) \Rightarrow x \neq y$,

together with an ordering mapping

$$b : L(G) \rightarrow P(G)$$

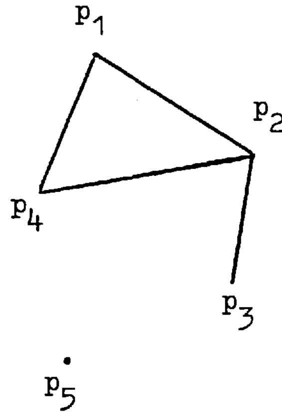
where,

if $z = (x, y) \in L(G)$

then

$b(z) \in \{x, y\}$

e.g. if G is



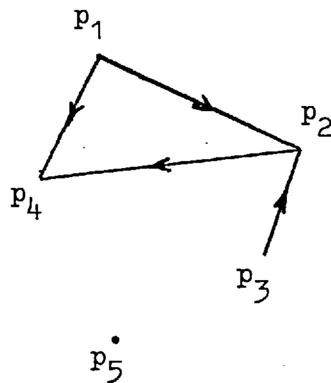
and $b : (p_1, p_2) \mapsto p_1$

$(p_4, p_2) \mapsto p_2$

$(p_2, p_3) \mapsto p_3$

$(p_1, p_4) \mapsto p_1$

- then the digraph D generated by G and b is



(In other words, the elements of $L(G)$ are ordered and b selects the base points of pairs in $L(G)$.)

equivalently (ii) a digraph D is

(a) a (finite) set of points P

(b) an irreflexive relation ρ on P.

Notation: following defn.(ii) for a digraph

if $D = \{P, \rho\}$, then:

if $x, y, \in P$, $x \neq y$ and $y \in \rho x$

(i.e. $x \rho y$), then we denote this by:

$$x \xrightarrow[\rho]{} y$$

or, more simply, by

$$x \longrightarrow y$$

Defn: Given a digraph, $D = \{P, \rho\}$, then a ρ-link (or ρ-path) joining x_1 to x_n, is a sequence in P,

$$x_1, x_2, \ldots, x_n \quad : \quad n > 1$$

such that $x_{i+1} \in \rho x_i : \; \forall \; i : \; 1 \leq i \leq n-1$

Defn: A cycle in D (as above) is a ρ-link from x_1 to x_n such that $x_1 \in \rho x_n$

Defn: An acyclic digraph is a digraph with no cycles.

Defn: Given $D = \{P, \rho\}$, then associated with ρ is its pseudo-inverse $\rho^{-1}x = \{ y : y \in P$ and $x \in \rho \, y \}$

Defn: If $D = \{P, \rho\}$ and $x \in P$, then $\rho \, x$ is the outbundle of x in D. The cardinality of $\rho \, x$ is called the outdegree of x, written $|\rho \, x|$.

Similarly, $\rho^{-1}x$ is the <u>inbundle</u> of x in D and $|\rho^{-1}x|$ is the indegree of x.

<u>Notn</u>: When $|\rho\, x| = 1$ we write $\rho\, x = y$ instead of $\rho\, x = \{y\}$, and similarly when $|\rho^{-1}x| = 1$ we write $\rho^{-1}x=y$.

5.2.3 Finite Disjoint Unions

Given a collection of sets $S_i : i \in \mathbb{Z}$, form an isomorphic set of cartesian products by the mapping,

$$d : x \to (x, i) \quad \forall\, x \in S_i \quad \forall\, i \in \mathbb{Z}$$

Then, given any two sets S_j, S_k $(j \neq k)$ the derived sets dS_j, dS_k are disjoint and we write:

$$dS_j \cup dS_k$$

as

$$S_j \sqcup S_k \ .$$

This is called the <u>disjoint union</u> of S_j, S_k.

e.g. if $S_1 = \{a, b, c\}$

$S_2 = \{a, c, d\}$

then $S_1 \cup S_2 = \{a, b, c, d\}$

and $S_1 \sqcup S_2 = \{(a, 1), (a, 2), (b, 1), (c, 1), (c, 2), (d, 2)\}$.

Extending this notation in a natural way, we obtain

$$\bigsqcup_{i \in A} S_i = S_{j_1} \sqcup S_{j_2} \sqcup \ \ldots \ \sqcup S_{j_n}$$

where $A = \{j_1, \ldots, j_n\} \subset \mathbb{Z}$

$$\bigsqcup_{i \in \mathbb{N}} S_i = S_1 \sqcup S_2 \sqcup \ \ldots \ \sqcup S_n \sqcup \ldots$$

and $$\bigsqcup_{i\in\mathbb{Z}} S_i = \ldots \sqcup S_{-m} \sqcup \ldots \sqcup S_{-1} \sqcup S_o \sqcup S_1 \sqcup \ldots \sqcup S_n \sqcup \ldots$$

If now S_i is such that $\exists\, m, n \in \mathbb{Z}$ and

$$S_i = \emptyset \;\forall\; i < m$$

and $$S_i = \emptyset \;\forall\; i > n \quad \text{with} \quad m \leq n$$

then we abuse the notation by regarding A as an abbreviation† for $A \sqcup \emptyset$ and hence write

$$\bigsqcup_{i\in\mathbb{Z}} S_i \quad \text{for} \quad S_m \sqcup S_{m+1} \sqcup \ldots \sqcup S_{n-1} \sqcup S_n$$

We call $\bigsqcup_{i\in\mathbb{Z}} S_i$ the finite disjoint Union of S_i.

---

†Clearly, what we really need is an embedding not unlike the identification of finite polynomials with formal sums, of the form $\sum_{i=o}^{\infty} a_i x^i$, see e.g. [47] or rings of formal power series [73].

5.3 Description of the Space

5.3.1 Non-Mathematical Description

The structure of CPS is language dependent as are the transformations which can take place within the structure. We believe that CPS is capable of representing any high-level language, including Algol-68. Since Carabiner is extensible, it may seen that we require a metalanguage similar in power to that of the Algol-68 report [108]. However, we take the view that there must be a simpler way of describing the facilities of such a language, e.g. along the lines of Algol-N [55, 104, 124].

We envisage the initial mode-structure to be a simplified representation of the diagram in [126]. In our list notation this is:-

MODE IS (mood, union of $MODE^{n+1}$)

mood IS (type, stowed)

type IS (Plain, format, proc)

etc.

Ideally we should be able to start from a set of empty lists, augmenting and linking these lists as the syntax analysis and semantic validation of the program progresses; further extensions being made as the (possibly recursive) declarations are executed.

The one major stumbling block of this philosophy is that all high-level languages have an Algol-68-like prelude which needs to be embedded within CPS. We shall consider this prelude to be part of every program written in a particular language; hence the basic set of empty lists together with the extensions defined by the prelude constitute the initial CPS state for that language. A detailed description of the full initial state for the language X is given in chapter 7.

The fundamental elements of CPS are:-

(a) The lists: STORE and

ATTRIB (i.e. attributes)

and

(b) The operator k (and hence k^{-1})

Defn: STORE is a list which represents both the heap (of global entities) and the stack of local quantities.
Heap material is augmented to the left of STORE and local material to the right. A special symbol, $\updownarrow$, is used in STORE to denote Carabiner (data) block entry.

Defn: ATTRIB is a list of attributes (usually types or modes) used in a program. It contains the standard modes of the source language and of Carabiner. ATTRIB may be augmented by mode declarations as in Algol 68 and may also hold temporary information required at compile time.

Notation: In the sequel we shall frequently refer to elements of the list STORE, e.g.

x OF STORE .

Where no ambiguity arises, we denote this simply by the name of the element i.e. x.

The initialization of STORE consists, in general, of (left) augmentation of standard procedures, system constants and a 'nameless'† procedure pα, the body of which is undefined, but which will eventually comprise the Carabiner 'object' program produced by the source program.

---

† Here p generates a 'name' for the procedure α. A formal definition is given in §5.3.2.

Moreover:

$\S_i$ OF $p\mathcal{a}$ = $\underline{k}$ ($\underline{\text{string}}$ of ATTRIB)

and if $\mathcal{a}$ IS $(\alpha_1, \alpha_2, \alpha_3, \alpha_4)$

where † α_1 = INSERTL(STORE, $p\mathcal{a}$, OUTPUT FROM $p\mathcal{a}$)

α_2 = OUTPUT FROM $p\mathcal{a}$ IS $(\S_1, \ldots, \S_m)$

α_3 = LINK($\underline{\text{string}}$ OF ATTRIB, $\{\S_1, \ldots, \S_m$ OF OUTPUT FROM $p\mathcal{a}$ })

(α_4 = BLOCK (τ) - where τ denotes the procedure (program) body.)

where ($1 \leq i \leq n$, $1 \leq j \leq m$) for some suitable $n, m \in \mathbb{N}$.

[We claim that this is a reasonable way of representing a software system based on a stored program computer. After translation of a source program into Carabiner, the model is self-contained and only requires $\underline{e}$ to operate on $\mathcal{a}$ to activate the computational process.]

$\mathcal{a}$ acts on a row of strings and yields a row of strings. These strings are the I/O buffers.

String is a Carabiner mode not necessarily distinct from the language defined modes.

Loading and unloading of buffers is outside the scope of the program being modelled.

5.3.2 Mathematical formulation

We define CPS constructively from a void finite disjoint union α, where

$$\alpha = \bigcup \alpha_i \quad , \quad \alpha_i = \emptyset \quad \forall\, i \in \mathbb{Z} .$$

α is equivalent to STORE, with α_i $(i < 0)$ corresponding to global entities and α_i $(0 < i)$ denoting local (stack) quantities, α_o is the component of α which represents the name of the program.

---

† For discussion see §7.2.

In general, each non-null component of α is a $\underline{k}$-directed graph. This may, however, consist only of a trivial (single point and no line) graph.

The key node of each graph, i.e. the node which represents the entry in the list STORE, may be referred to by its own name or (unambiguously) by the corresponding α_i. This is a further abuse of terminology but no side effects arise.

For any 'nameless' procedure, e.g. the program $\mathfrak{a}$, we define

$$p\mathfrak{a} = \underline{k}^{-1}\ \mathfrak{a} \cap \text{STORE}.$$

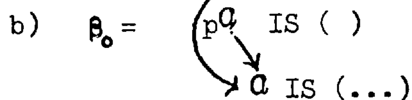
$p\mathfrak{a}$ possesses $\mathfrak{a}$ and usually is equivalent to $\mathfrak{a}$'s input parameter list.

According to the definition of α so far given, a typical CPS (to avoid confusion we will call the STORE β) may be defined thus:-

a) $\beta = \emptyset$

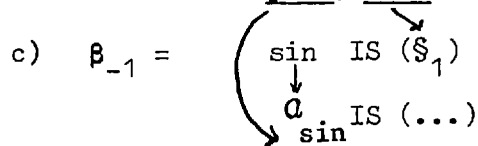
ATTRIB IS (proc)

b) $\beta_0 =$



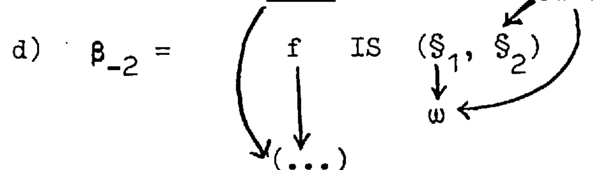
ATTRIB IS (proc, real)

c) $\beta_{-1} =$



ATTRIB IS (proc, int)

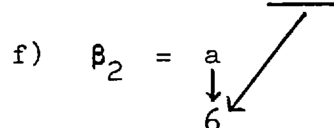
d) $\beta_{-2} =$



e) $\beta_1 = \Phi$

ATTRIB IS (int)

f) $\beta_2 =$



ATTRIB IS (real)

g) β_3 = b → 7.1 ← (real)

Hence β is an ordered set of six disjoint k-digraphs (the unnamed digraph relation is, as always, taken to be the k. In particular, notice that

$$\underline{k}^{-1}\,a = \underline{\text{proc}}$$

and

$$\underline{k}^{-1}\,a_{\text{sin}} = \underline{\text{proc}}$$

but

$$\underline{k}^{-1}\,a \neq \underline{k}^{-1}\,a_{\text{sin}} .$$

We require that these mode indicators should be identified with the same characterizations and hence we define an equivalence relation over a subset of P where $P = \bigsqcup P_i$ and P_i is the underlying point set of the digraph α_i. Denoting the relation by $\simeq$ we say that

$$\underline{\text{amode}} \text{ of } \alpha_i \simeq \underline{\text{amode}} \text{ of } \alpha_j$$

whenever

(i) $\underline{\text{amode}} \in$ ATTRIB

and (ii) $i \neq j$.

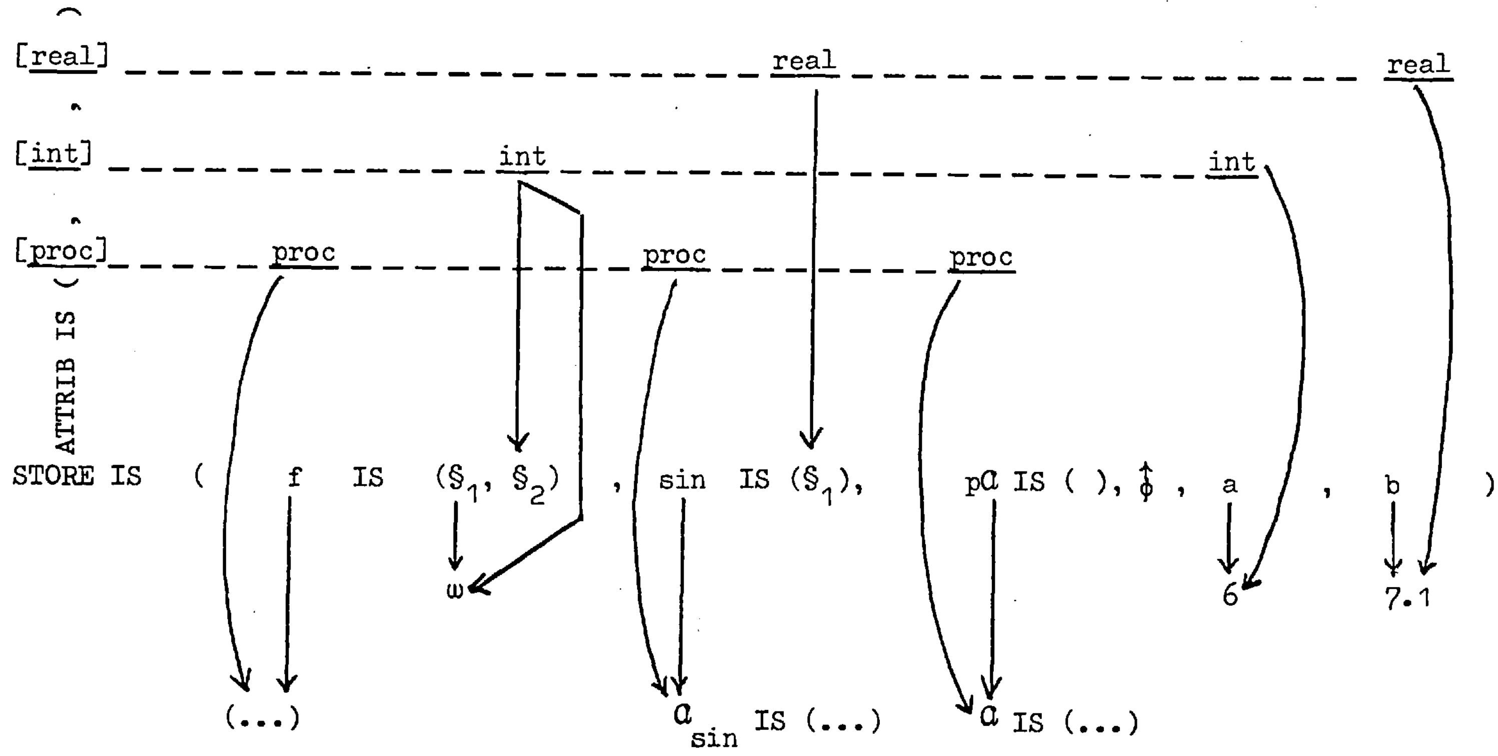
Equivalently, let ATTRIB be an ordered set of attributes (modes etc.) each of which is represented by its own equivalence class;

e.g. ATTRIB IS ([amode], [anothermode], ...) .

With this characterization of ATTRIB part (i) of the definition of $\simeq$ should be:-

'whenever [amode] $\in$ ATTRIB' .

If we now use solid lines and arrows to denote k-links and broken lines to indicate equivalence relations in ATTRIB, then the CPS based on β as defined above may be represented by the diagram:-



[real]
[int]
[proc]
ATTRIB IS (
real
real
int
int
proc
proc
proc
STORE IS (f IS (§1, §2) , sin IS (§1), pα IS ( ), ϕ , a , b)
ω
6
7.1
(...)
α sin IS (...)
α IS (...)

We further stipulate that, in order for $\underline{k}$ to traverse STORE if [$\underline{\text{amode}}$] ∈ ATTRIB then

$$\underline{k}\ [\ \underline{\text{amode}}\ ] = \{\ \underline{k}\ \underline{\text{amode}} : \underline{\text{amode}} \in [\ \underline{\text{amode}}\ ]\ \}.$$

In our example, this yields:-

$$\underline{k}\ [\ \underline{\text{int}}\ \text{OF ATTRIB}\ ] = \{\ \S_2\ \text{OF f},\ \underline{k}\ (\S_1\ \text{OF f}),\ \underline{k}\ \text{a}\ \}$$

etc.

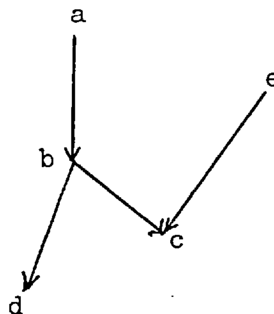
5.3.3 The $\underline{k}$-completion† of CPS

Defn: A CPS is $\underline{k}$-complete iff each component of the underlying finite disjoint union (i.e. STORE) is $\underline{k}$-complete.

Each component of STORE is an acyclic digraph, and hence we need to define the $\underline{k}$-completion of such a graph but before doing so we consider an example.

Take the digraph D with the relation $\underline{k}$:-

D



† the notion of completion defined here is NOT the same as in traditional graph theory [7, 52, 53]; however, notice that any sequence $\underline{x}$ on D : $x_{i+1} \in \underline{k}\ x_i$ is well defined and achieves its limit after a finite number of steps; hence it is Cauchy and the space of such sequences is complete.

D may be fully described by either

$$
\begin{aligned}
\text{(i)}\quad & \underline{k}\ a = b \\
& \underline{k}\ b = \{\ c,\ d\ \}\ ,\ \underline{k}\ c = \phi \\
& \underline{k}\ d = \emptyset \\
& \underline{k}\ e = c
\end{aligned}
$$

$$
\begin{aligned}
\text{or (ii)}\quad & \underline{k}^{-1}a = \emptyset \\
& \underline{k}^{-1}b = a \\
& \underline{k}^{-1}c = \{\ b,\ e\ \} \\
& \underline{k}^{-1}d = b \\
& \underline{k}^{-1}e = \emptyset \quad .
\end{aligned}
$$

The nodes a and e, and c and d are peculiar in that either their inbundles or their outbundles are empty. These facts are easily characterised by the predicates

$$\underline{k}\ x -= \emptyset$$

and $$\underline{k}^{-1}x = \emptyset\ .$$

However, the evaluation of this predicate does, in a sense, lead us outside the domain of definition, because:

$$\underline{k}\ x = \emptyset \text{ iff } y \notin \underline{k}\ x \quad \forall\ y \text{ in CPS.}$$

Now since D is acyclic we are able to modify it into what we shall call its $\underline{k}$-completion or $\underline{k}$-complete form. Before defining this, we need some notation and terminology.

Defn: Given points a, b in a set, we may define one or two $\underline{k}$-semi-links between them. Suppose $a \notin \underline{k}^{-1}b$ and $b \notin \underline{k}\ a$, then (a into b) is a non-invertible† relation denoted by

† in the sense that the inverse relation is not defined.

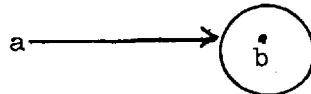
with the effect that:

$$a \in \underline{k}^{-1} b$$

but $\quad b \notin \underline{k}\, a$.

<u>Analogously</u>:

(b outof a) is denoted by



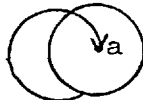
and has the effect that:-

$$a \notin \underline{k}^{-1} b$$

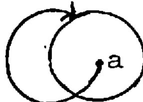
while $\quad b \in \underline{k}\, a$.

Now since a $\underline{k}$-semilink is non-invertible we may define (a into a) and (a outof a) without creating cycles.

<u>Defn</u>: An <u>in-k-semiloop</u> at a point 'a' is equivalent to the $\underline{k}$-semilink (a into a) and is represented by



Similarly, the <u>out-k-semiloop</u> at 'a' is equivalent to (a outof a) and denoted



Given an acyclic digraph D, we form its <u>k-completion</u> by the following process:-

(a) $\forall\ x \in D : \underline{k}^{-1} x = \emptyset$

add an in-$\underline{k}$-semiloop to D at x.

(b) $\forall\ x \in D : \underline{k}\ x = \emptyset$ add an out-$\underline{k}$-semiloop to D at x.

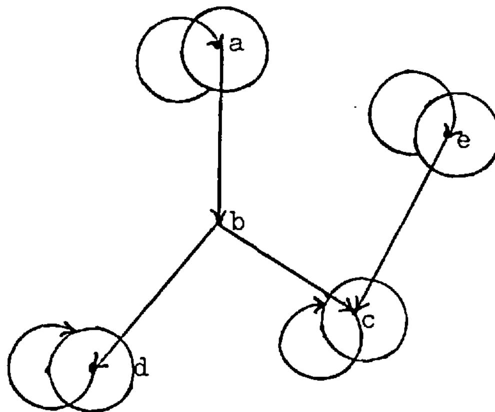
Then:

$$\underline{k}\, x \neq \emptyset \neq \underline{k}^{-1} x \qquad \forall\, x \in D.$$

The nodes which previously had void inbundles and/or outbundles now have inbundles and/or outbundles which consist of that same node and are defined to have indegree and/or outdegree 1.

Defn: An element x of a $\underline{k}$-complete (acyclic) digraph such that $\underline{k}\, x = x$ is called an atom and an element y such that $\underline{k}^{-1} y = y$ is called an anti-atom.

The $\underline{k}$-completion, D_k, of D may now be represented



Now:- $\underline{k}^{-1} c = \{b, e\}$

$\underline{k}^{-2} c = \{a, e\}$

$\underline{k}^{-3} c = \{a, e\}$ etc.

and

$\underline{k}\, a = b$

$\underline{k}^{2} a = \{c, d\}$

$\underline{k}^{3} a = \{c, d\}$ etc.

By introducing the natural convention that $\underline{k}^{0} x = x \;\forall\, x$ we then have a system which gives a valid and meaningful result for $\underline{k}^{n} x$ when $x \in CPS$ and $n \in \mathbb{Z}$.

Although not used in language X, we remark here that the completion of CPS allows the definition of fixed points (relative to $\underline{k}$ and $\underline{k}^{-1}$) derived from any element x (OF STORE), thus:-

Defn: Fixed point operators K^+ and K^- can be defined for any x in a $\underline{k}$-complete CPS.

st.

$$K^+x = \{ y : y \in \underline{k}^n x \text{ and } \underline{k}\, y = y \text{ for a suitable } n \geq 0\}$$

and

$$K^-x = \{ y : y \in \underline{k}^{-n} x \text{ and } \underline{k}^{-1} y = y \text{ for a suitable } n \geq 0\}$$

CHAPTER 6

THE CARABINER LANGUAGE

The language is built on the three basic operators, S, k, and e, which are described in the sections 6.1 to 6.3 below. These primitive operations may be gathered together in various ways to form procedures, functions and macros as described in §6.4 and §6.8.

Formalization of the order in which sequences of operations are performed (and hence the central structure of Carabiner) is given in §6.5. This is closely related to the way in which e is used to model the OBEY command of a computing system and is defined in §6.6

The inter-relationship between k and e may be used to give a formalization of left- and right-hand modes of evaluation as in CPL [6, 15]. This leads to more precise characterization of S particularly when it is used to emulate assignment commands as in §6.7.

The order in which the material is presented may be challenged, however, after reading it will be obvious that S, k and e are so closely inter-dependent that any order of presentation would necessitate both forward and backward referencing between the subsections.

6.1 The Operator S

Related to Markov's normal Algorithms [74] is a substitution operator Σ, defined on three words x, y, z by the Markov algorithm thus:-

$$\Sigma(x,\ y,\ z)\ \equiv\ y \rightarrow\ .z$$
$$\Lambda \rightarrow\ \Lambda$$

Σ denotes the replacement of the leading occurrence of y in x by z; if $y \not\subset x$ then it is undefined.

Wesselkamper [116] defined an operator S (which we shall call S_w) that performed a similar operation upon substructures of his CRAMPON machine, and which was defined to be the identity if not applicable:

$S_w\, x\, y\, z\ \equiv$ if y 'bt' x then replace it by z
otherwise do nothing

Here 'bt' means 'belongs to' in the sense of 'being a substructure of' the current state of the machine. S_w was also defined when x was not a structure but a 'value', in which case:-

$$S_w\, a\, b\, c\ =\ \begin{cases}\text{if } a{=}b \text{ then } c\\ \text{else } a\end{cases}$$

Notice that the distinction between the execution of S_w and Σ and their respective results is vague. We shall return to this in §6.3.

The Carabiner S is developed from the 'value' definition of S_w. The action of S involves (i) an equality test, and possibly (ii) a replacement.

By means of the operators k and e (§6.2, §6.3) we are able to specify positions and extract values; these may then be manipulated by S. We make the assertion that position specifiers and values are incomparable.

Using 'v' and 'p' to denote objects of type 'value' and 'pointer' we may fully define S as follows:

$Sv_1v_2v_3$ yields, if $v_1 = v_2$ then v_3 else v_1

$Sv_1v_2p_3$ yields, if $v_1 = v_2$ then p_3 else v_1

$Sv_1p_2v_3$, $Sv_1p_2p_3$ $\Rightarrow v_1$ — equivalent to the identity operation

$Sp_1v_2v_3$, $Sp_1v_2p_3$ $\Rightarrow p_1$ — equivalent to the identity operation

$Sp_1p_2v_3$, $Sp_1p_2p_3$ — if $p_1 = p_2$ then substitutes v_3 or p_3 into p_1 otherwise it is equivalent to identity operation.

Examples of the usage of S to model conditionals and (generalised) assignment are given in subsequent sections when other supporting operators have been defined.

Notice, that if we denote by γ the current state of the CPS and redefine 'bt' to relate to this space†, then each Carabiner assignment:-

$$S\alpha_1\alpha_2\alpha_3$$

can be more correctly written in the form:-

$$S_w \; \gamma\alpha_1 \; S\alpha_1\alpha_2\alpha_3$$

---

†This could be done by explicitly redefining S to act on sets and lists, but as will be seen this is not necessary.

Moreover, if we have a sequence of such statements:-

$$S\alpha_{1,1}\ \alpha_{1,2}\ \alpha_{1,3}$$

$$S\alpha_{2,1}\ \alpha_{2,2}\ \alpha_{2,3}$$

$$\vdots$$

$$S\alpha_{n,1}\ \alpha_{n,2}\ \alpha_{n,3}$$

and we denote by γ_i the state of CPS before the execution of the ith statement then we have:-

$$\gamma_1$$

$$\gamma_2 = S_w\ \gamma_1\ \alpha_{1,1}\ S\ \alpha_{1,1}\ \alpha_{1,2}\ \alpha_{1,3}$$

$$\gamma_3 = S_w\ \gamma_2\ \alpha_{2,1}\ S\ \alpha_{2,1}\ \alpha_{2,2}\ \alpha_{2,3}$$

$$\vdots$$

$$\gamma_n = S_w\ \gamma_{n-1}\ \alpha_{n-1,1}\ S\alpha_{n-1,1}\ \alpha_{n-1,2}\ \alpha_{n-1,3}$$

Here the final state of CPS is

$$\begin{aligned}\gamma &= S_w\ \gamma_n\ \alpha_{n,1}\ S\alpha_{n,1}\ \alpha_{n,2}\ \alpha_{n,3}\\ &= S_w\ S_w\ \gamma_{n-1}\ \alpha_{n-1,1}\ S\alpha_{n-1,1}\ \alpha_{n-1,2}\ \alpha_{n-1,3}\\ &\qquad\qquad \alpha_{n,1}\ S\alpha_{n,1}\ \alpha_{n,2}\ \alpha_{n,3}\\ &\ \vdots\\ &= S_w\ S_w\ \cdots\ S_w\ \gamma_1\\ &\qquad \alpha_{1,1}\ S\alpha_{1,1}\ \alpha_{1,2}\ \alpha_{1,3}\\ &\qquad \alpha_{2,1}\ S\alpha_{2,1}\ \alpha_{2,2}\ \alpha_{2,3}\\ &\qquad \vdots\\ &\qquad \alpha_{n,1}\ S\alpha_{n,1}\ \alpha_{n,2}\ \alpha_{n,3}\end{aligned}$$

All this means is that each assignment changes the current state of the space and hence the nth assignment will act on the space as left by the $(n-1)^{th}$ assignment.

Put mathematically, each assignment represents a function $S_i : \Gamma \rightarrow \Gamma$ where Γ is the set of all possible states of CPS. If this function symbolises the assignment

$$S_w \; \gamma_i \; \alpha_{i,1} \; S\alpha_{i,1} \; \alpha_{i,2} \; \alpha_{i,3}$$

then the above sequence represents the functional composition

$$S_n \circ S_{n-1} \circ \cdots \circ S_2 \circ S_1 : \Gamma \rightarrow \Gamma$$

Now, by virtue of the list format of STORE and ATTRIB (see chapter 5) and the associated operators, specification of the subject of S via the 'bt' relation is superfluous. Also, since

$$\mathcal{R}_s \subseteq \mathcal{D}_s \quad \text{(range} \subseteq \text{domain)}$$

we need only know the order of the operations and hence write them as a list. i.e. $S_1, S_2, \ldots, S_n$. We shall return to this in §6.4

6.2 The Operator k

k (or κ) was introduced into CRAMPON as a way of relating an identifier to the value it possessed. Each identifier was regarded as a name-value pair and k was a projection from this pair onto its second component.

e.g. $\underline{k}\; x \;=\; \underline{k}\;(x,\;3) \;=\; 3$ (say)

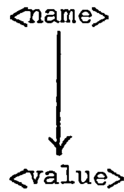
Formally:

$$\underline{k}\;(\langle name\rangle) = \underline{k}\;(\langle name\rangle : \langle value\rangle) = \langle\, value \,\rangle$$

and

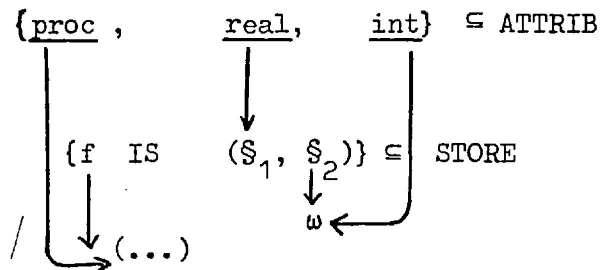
$$\underline{k}\;\langle value\rangle = \langle value\rangle$$

Depicting the (<name> : <value>) pair as -



- the extension of k to be a general graph traversing operator is quite natural, as is the extraction of its pseudo-inverse $\underline{k}^{-1}$. This was done in §5.3.

We are now able to describe fully the states of CPS which represent more complicated constructs such as:-

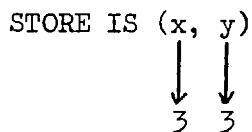


- by $\underline{k}^{-1}\ \underline{k}\ f \cap$ ATTRIB = proc OF ATTRIB

$\underline{k}^{-1}$ ($§_1$ OF f) = real OF ATTRIB

etc.

Notice, however, that we may have ambiguities such as:-



now $\underline{k}x = 3$ and $\underline{k}y = 3$, but do we intend that $y \in \underline{k}^{-1}\underline{k}x$? The operator $\underline{e}$ resolves these conflicts as will be shown in §6.7.

6.3 The Operator $\underline{e}$

Up to now we have only been concerned with describing the execution of a validated and translated program and hence the vital distinction between (a) a procedure, and (b) its effect, was ignored. However, in, for example, the translation phase, we must be able to distinguish between these notationally similar concepts. e.g.:

Translate (f(5))

could mean (i) evaluate f at 5 and Translate the result, or

(ii) Translate the program segment representing the function f acting on 5, or

(iii) the program segment representing the translation routine acting on the value of f at 5, or

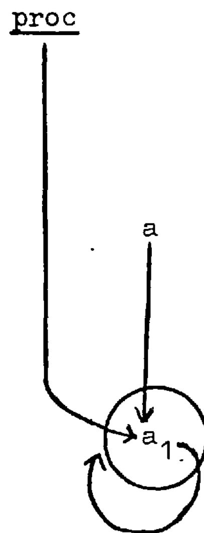
(iv) the program segment representing the translation routine acting on f acting on 5.

A first thought on a method of distinguishing these cases was to extend the role of $\underline{k}$. This would seem very desirable since there is much similarity between the concept of a variable delivering a value and that of a function (+ parameters) delivering a result. However $\underline{k}$ is itself an operator and hence difficulties arise.

To illustrate the problem, consider the expression:-

$$\underline{k}^2 a$$

acting on the structure:-



i.e. 'a' is the name of a procedure

- then $\underline{k}^2$a (= $\underline{k}$ $\underline{k}$ a) could mean:-

(i) $\underline{k}^2$a (i.e. do no activation)

(ii) $\underline{k}$ ($\underline{k}$ a) (activate $\underline{k}$ acting on a)

= a_1

(iii) $\underline{k}$ ($\underline{k}$ a) ($\underline{\text{do}}$ a_1)

= $\underline{k}$ a_1

etc.

The trouble here is that $\underline{k}$ is being used for two distinct (but indistinguishable) purposes; namely as a graph-traversing operator and as an activiation operator; moreover, the graph operation may be the operand of the activation operator.

If we denote activation by the operator $\underline{e}$ (and use $\underline{k}$ merely for graph traversing) then the above cases could be characterised by:-

(i)' $\underline{k}^2$a

(ii)' $\underline{e}$ $\underline{k}$ a

(iii)' $\underline{e}$ ($\underline{e}$ $\underline{k}$ a)

Notice that in examples (ii) and (iii) the leftmost k is replaced by e but the second k in (ii) denotes the operation to be performed, while in (iii) the same token is activated (by e in (iii)') to yield the operation to be performed, namely a_1.

The departure from using k to relate a function to its result seems to break with the underlying philosophy (see pages 10 and 11) that S and k are sufficient to describe all computational processes. If we are only attempting to define the (final) result of applying an arbitrary Boolean function to a vector of Boolean variables, as is the case in generating the result of executing a computer program; then only S is necessary [116]; but when talking about the result we (implicitly) need k to be able to say e.g.:

k < bit n> = 0 etc.

Moreover, if we regard a computing process as being defined by a collection of functions acting on such a vector, and these functions are combined to form a program - represented as a vector of Boolean values - it would seem reasonable that we require not only an explicit activation operator but also a formalism to describe which functions should be used at any given state. This matter is elaborated further in §6.5 and §6.6.

The operator e may also be used to distinguish between (node) values and position specifiers within the program space as shown in §6.7.

Finally we note a special property of e when acting upon a procedure defined by a list of more elementary operations. In this

case $\underline{e}$ distributes over that list.

i.e. given $\underline{x} = (x_1, \ldots, x_n)$

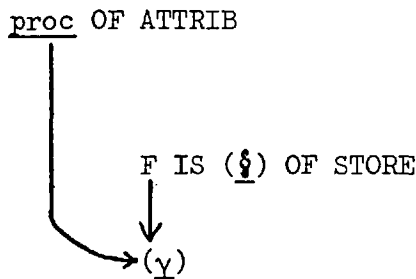
and $\underline{x} \in \underline{k}$ ($\underline{\text{proc}}$ OF ATTRIB)

then $\underline{e}\ \underline{x} \equiv (\underline{e}x_1, \ldots, \underline{e}x_2)$

If the list $\underline{x}$ is derived from a 'sequence' of embedded functions (cf. S in §6.1) then this property of $\underline{e}$ is not merely convenient but indeed necessary. Moreover this property is required to execute 'dormant' procedures as we shall see in §6.5.

6.4 Procedures and Functions

The fundamental structure associated with a procedure is its template. Essentially this is a representation of the code sequence required to evaluate the function, together with the mechanism for loading its parameters and determining its order of evaluation. Diagramatically it is represented thus:-



i.e. F is the name of an entity of type proc (procedure), $\underline{\S}$ denotes the formal parameter vector, and $\underline{\gamma}$ denotes the sequence of operations required to do the evaluation after $\underline{\S}$ has been replaced by the actual parameter vector $\underline{\S}'$.

Any procedure used within a program has to have a template defined either (a) by the system, i.e. in the language prelude, or (b) by the execution of appropriate definitions within the program.

Now consider a subroutine procedure call:

$\underline{\text{do}}\ F(\underline{v})$,

if we denote the formal parameter vector of F by $\underline{\S}$, then the execution of the above statement goes as follows:

1.[†] Create a copy of the procedure template of F.

2. Load $\underline{\S}'$ by either substituting actual parameters, or linking[††] $\underline{\S}$ to actual parameters.

3. Activate the body of the procedure. (This acts upon $\underline{\S}$).

4. Delete F.

If now F is a 'function' procedure, so we may have:

$\underline{y} := F(\underline{v})$,

then the operation is as before but with the extra step of creating and loading a vector, [†††]OUTPUT FROM F, which is not deleted with F. These extra steps can be included in the function definition. (See §9.4).

The assignment to $\underline{y}$ can then be carried out as if it were:

$\underline{y}$:= OUTPUT FROM F

[†]In practical cases F (i.e. its template) might be stored in a library file.

[††]This preserves side effects but is dependent on the type of parameter handling (name, value, reference etc.). By default we assume call by value (but see chapter 9).

[†††]This phrase is used so as to avoid confusion with the list operator OF (§5.2.1).

Using the operator $\underline{k}$, we may describe the situation in a more rigorous way.

A procedure body is an entity γ such that:

$$\underline{\text{proc}} \text{ OF ATTRIB} \in \underline{k}^{-1} \gamma$$

i.e. $\gamma \in \underline{k}$ ($\underline{\text{proc}}$ OF ATTRIB).

Now γ may be defined by either an Extended Markov Algorithm (EMA) or by a list of more elementary operations or procedures. EMA's are described at length in chapter 4 and herein are regarded as atomic.

If the function is defined by a list of operations then the sequence of operations properly constituting the procedure F is:-

$$\begin{array}{l} S\ \Lambda,\ \Lambda,\ \text{STRUCT (F)}^{\dagger} \\ S\ \underline{\S} \text{ OF F},\ \underline{\S} \text{ OF F},\ \underline{x} \\ \underline{e}\ \underline{k}\ F \\ S\ \text{STRUCT (F), STRUCT (F)},\ \Lambda \end{array}$$

We shall denote this by F($\underline{x}$). Activation is then caused by sequential execution of this list and is written $\underline{e}$ F($\underline{x}$). Note that this is purely shorthand notation for

$$\begin{array}{l} \underline{e}\ S\ \Lambda,\ \Lambda,\ \text{STRUCT (F)} \\ \underline{e}\ S\ \underline{\S} \text{ OF F},\ \underline{\S} \text{ OF F},\ \underline{x} \\ \underline{e}\ \underline{e}\ \underline{k}\ F \\ \underline{e}\ S\ \text{STRUCT (F), STRUCT (F)},\ \Lambda \end{array}$$

the third term of this is then expanded to give (say):-

$$\underline{e}\ \underline{e}\ \underline{k}\ F = \underline{e}\ \gamma = \underline{e}(\gamma_1, \ldots, \gamma_n) = (\underline{e}\ \gamma_1,\ \ldots,\ \underline{e}\ \gamma_n)$$

---

† STRUCT(x) is defined in §8.1 and gives the structure within CPS which refers only to the object x. In this context we may think of it as the function template.

It is usual in mathematics for f(x) to mean the <u>result</u> of applying a function f to the argument x, the actual process of evaluating the function being taken for granted; in the 'abstract' world of mathematical evaluations this causes no problems, however when we are concerned with the actual mechanism of evaluation (as we are in describing computing machines and processes), it is sometimes necessary to talk about this process explicitly and hence the distinction between a process and its result needs to be made.

This has been done by the operator <u>e</u>.

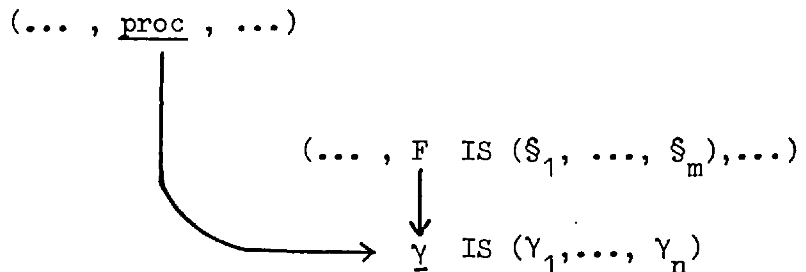
Now the classical meaning of f(x) is represented by <u>e</u> f(x). This apparent clash in the use of f(x) reflects an inconsistency of the traditional notation and makes the description of computing processes easier and more explicit.

6.5 On Orders of Evaluation and Control Functions

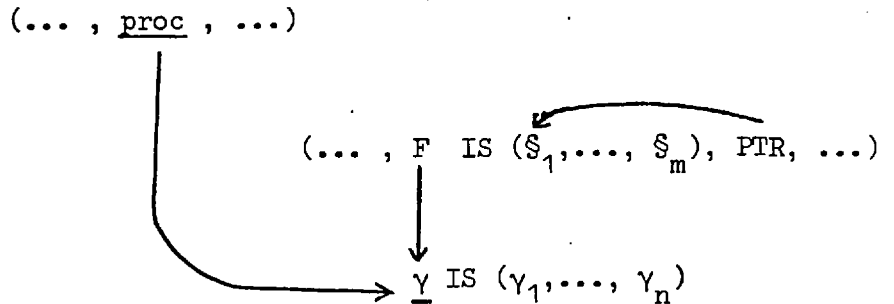
6.5.1 Non-control Functions

So that we may precisely define which function should be executed at any given point of the 'program' we specify the order in which actual parameters are created and (non-control) functions (defined by sequences of more elementary functions) are elaborated.

Given a 'procedure template' as in §6.4, i.e.:-



If we append a pointer, PTR, to this template so that $\underline{k}$PTR = $\S_1$ OF F, we have:-



Before a procedure is executed, by $\underline{e}$, the template is copied into a local area of STORE and the actual parameters are copied into position; the procedure is then activated. In mathematics, these processes are implicit and hence the order in which they are carried out is ignored; however in computing processes it is possible for the 'evaluation' of one parameter to affect the 'evaluation' (or even the specification) of another, hence this needs to be formalised. The pointer, PTR, allows us to do not only this but also to dictate the order of evaluation of sequences of functions.

Very simply, the process for dealing with the above procedure, F, may be defined in 'pidgin Algol' as follows:-

```
(while k PTR ∈ {§1 OF  F,..., §m-1 OF  F}
 do  (evaluate k PTR;
      set k PTR to next item in list);
 evaluate k PTR;
 set k PTR to γ1 OF γ;
 while k PTR ∈ {γ1 OF γ ,..., γn-1 OF γ }
 do (evaluate k PTR;
      set k PTR to next item in list);
 evaluate k PTR)
```

Notice that in general γ_n will be of the form TRIMR (STORE,F) which means erase F and its PTR, hence returning control to the calling function and another PTR.

To facilitate description of pointer manipulation in Carabiner we use the routines:

Advance Parameter Pointer (APP)

and Advance List Pointer (ALP)

Trivially these routines increment by one the pointer k PTR in either the parameter list or the defining statement list. e.g.

Given

..., F IS (a , b , c , d) , PTR , ...

then

APP ⇒

..., F IS (a , b , c , d) , PTR , ...

So that none of our arguments become circular we explicitly state that no pointer mechanism is associated with these two routines†.

Hence we have the Carabiner program:-

---

†This may seem somewhat arbitrary and indeed it is. If all instructions (functions, procedures, operators) of the underlying system had to be implemented by software then the system would never work and the sequence of 'set up' functions would regress infinitely since these 'set up' functions would need themselves to be 'set up' etc. etc. In practical cases we bootstrap up from hardware instructions which take zero or one parameter and are executed by a single (atomic) action, hence they need no pointer mechanism. Any process of 'semantic refinement' if taken to a meaningful limit must reach such basic hardware actions.(cf. Woodger's traversing of semantic levels [120] until he eventually 'strikes bottom'.)

$\underline{e}$(TPL[†] ($\underline{e}$($\underline{ek}$PTR $\in$ {δ_1 OF F,..., δ_{m-1} OF F }), $\underline{e}$($\underline{e}$ $\underline{k}$ PTR, APP)),
$\underline{e}$ $\underline{k}$ PTR,
BREAK[††] (PTR, $\underline{k}$ PTR),
LINK[††] (PTR, γ_1 OF $\underline{k}$ F),
TPL($\underline{e}$ ($\underline{e}$ $\underline{k}$ PTR $\in$ {γ_1 OF $\underline{k}$ F, ... γ_{n-1} OF $\underline{k}$ F}), $\underline{e}$($\underline{e}$ $\underline{k}$ PTR, ALP)),
$\underline{e}$ $\underline{k}$ PTR)

- this assumes that $n,m \geq 1$ and that the function and all its parameters are fully evaluated. If this is not so then conditional actions have to be embedded within the function definition, we return to this point in §6.6.

6.5.2 Control Routines

Regardless of whether the high-level source language is block-structured or not, we may require the resultant Carabiner program to be 'structured' (by which we mean free of explicit 'goto' statements - but see the appendix) and this usually requires the creation of Carabiner blocks. To model this we use the two 'routines':

BLOCK (<block body>)

and EXIT (< exit depth >)

Here <block body> is the list of statements to be executed within the BLOCK and <exit depth> is a strictly positive integer, which specifies the number of nested blocks to be exited from, or 'T' for terminate.

---

† TPL(X, Y) ~ <u>while</u> X <u>do</u> Y
(defined in §6.5.2).

†† Defined in §6.8.1.

Within blocks, jumps may be either (logically) forward or backward. These two cases give rise to '<u>if</u>-<u>then</u>-<u>else</u>' constructs or (potential) loops, and are modelled by the operations RAM (ramification) and PTL (Process, test, loop) which were formally defined in §4.5.1. Recall:

RAM(a, b, c) ≡ <u>if</u> a <u>then</u> <u>do</u> b <u>else</u> <u>do</u> c

PTL(a, b) ≡ <u>do</u> a <u>if</u> b <u>then</u> <u>do</u> PTL(a,b)

Although, not strictly necessary, for completeness we also give:

TPL(a, b) ≡ <u>if</u> b <u>then</u> <u>do</u> PTL(a, b)

The templates of these routines are similar in structure to those of non-control routines, however the associated pointer manipulations are very different, as we shall see below.

6.5.3 Activation of Control Routines

Relating the distributivity of <u>e</u> to the control operations:

BLOCK, EXIT, RAM, PTL, TPL

demands special consideration because, in contrast to non-control procedures, the parameters constitute part of the sequence of instructions to be obeyed in executing the procedure. Because of arguments which will be given in §6.6, these parameters cannot be preceded by the activation operator <u>e</u> and hence it must be inserted when the control routine is activated.

Strict formulation of <u>e</u>-manipulations relative to BLOCK-EXIT constructs requires the specification of associated pointer handling. The same is true of the loop routines PTL and TPL, when modelled in a sequential environment†.

---

† While this is characteristic of current machine architecture, it may not always be so in the future.

Informally:-

given $\underline{\alpha} = (\alpha_1, \ldots, \alpha_n)$

and $\underline{\beta} = (\beta_1, \ldots, \beta_m)$

then:- (1) BLOCK ($\underline{\alpha}$) ~ BLOCK IS ($\underline{\alpha}$)

and $\underline{e}$ BLOCK ($\underline{\alpha}$) ~ $\underline{e}(\underline{\alpha})$

$= (\underline{e}\alpha_1, \ldots, \underline{e}\alpha_n)$

(2) Given BLOCK ($\underline{\alpha}$) as above with

α_i = EXIT (1) for some $(1 \le i \le n)$

then $\underline{e}\,\alpha_i \Rightarrow$ skip α_j $(i < j \le n)$ (i.e. execute as if $\alpha_j = \omega\ (i < j \le n)$)

(3) Given RAM ($p, \underline{\alpha}, \underline{\beta}$)

then $\underline{e}$ RAM ($p, \underline{\alpha}, \underline{\beta}$)

~ $\underline{e}$ $\underline{e}$ ($\underline{\text{if}}$ $\underline{e}$p $\underline{\text{then}}$ α $\underline{\text{else}}$ β)

(4) Given PTL ($\underline{\alpha}$, p)

then $\underline{e}$ PTL ($\underline{\alpha}$, p)

~ $\underline{e}$ ($\underline{\alpha}$, $\underline{e}$($\underline{\text{if}}$ $\underline{e}$p $\underline{\text{then}}$ PTL ($\underline{\alpha}$, p)))

and

(5) Given TPL ($\underline{\alpha}$, p)

then $\underline{e}$TPL ($\underline{\alpha}$, p)

~ $\underline{e}$ $\underline{e}$ ($\underline{\text{if}}$ $\underline{e}$p $\underline{\text{then}}$ PTL ($\underline{\alpha}$, p))

We may achieve these effects by the following 'next instruction' pointer manipulations:-

(1) On creation of a BLOCK, create simultaneously a pointer BPTR (cf. BLOCK template) linked to the first element of the associated list of instructions. Whilst this pointer exists, after execution of any statement ($\underline{k}$BPTR) then advance the pointer by the procedure ALP.

(2) On reaching EXIT(n), TRIMR STORE so as to remove n BPTR's. This automatically leaves the rightmost pointer remaining in STORE pointing to the next instruction to be executed. Trimming removes any intermediate pointers associated with other control constructs as described below.

(3) On encountering a RAM statement we must create a pointer CPTR. If the result of evaluating the condition is True then we set kCPTR to the first element of the 'then-clause' otherwise set kCPTR to the first element of the 'elseclause'. If now we terminate both clauses by an instruction to delete CPTR (and its related STRUCTure) the required activation sequence results.

(4)/(5) The required execution sequences are achieved by using the activation of RAM ((3) above) to model 'if-then-else' constructs with the definitions given above.

6.5.4 The 'Next Instruction'

Although it was originally intended that there shall be no mechanism in Carabiner for explicitly mirroring 'GOTO's; a by-product of the formalism given for flow of control within blocks has (by virtue of any Carabiner program being a procedure and a block) given us the facilities to do just this.

Loosely: e k BPTR ≡ the 'Next Instruction'

Modelling of a 'goto' statement is thus easily achieved since, if we model labels thus:-

ATTRIB IS(..., label, ..., L9, ...)

α IS (..., ω, ..., x, ...)

where x is the translation of 'goto' L9, then:-

x ≡ S k BPTR, k BPTR, e k(L9 OF ATTRIB)

Processing then continues from the point immediately after the null statement labelled by L9.

Note: If access to these pointers is forbidden (by the designer) and hence the Carabiner program has no explicit 'goto', then PTL has to be included as a primitive operation just like S, k and e.

6.6 Dormant Procedures

Any function (or procedure), f, which is not immediately preceded by e is not (at the current stage of execution of the system) activated and is called a dormant procedure. A dormant k acts as a pointer (or reference) and will be discussed further in §6.7.

Except when we are dealing with functionals [105] and in general we are not, dormant functions would seem to be of little use unless we can arrange for them to be 'awakened' at some later point in the program evaluation. The operator e, described earlier, already caters for this by virtue of the distributivity of e over any function-defining list in the strict order dictated by k PTR. Below we give examples of the use of dormant procedures.

6.6.1 Examples

The first example concerns the modes of initialization in CPL [6, 15]. Consider the declaration statements:

$$\underline{\text{let}}\ g\,[x] \;=\; A[i]x^2\ ;$$

$$\underline{\text{let}}\ h\,[x] \;\ddot{\simeq}\; B[j]x;$$

- the first being initialized by value, the second by reference. In terms of $\underline{e}$ and $\underline{k}$ we have:

$$\underline{e}\ \underline{k}\ (\underline{e}\ \underline{k}\ i \text{ OF } A) \quad \rightarrow \quad \text{for } A[i]$$

$$\underline{k}\ (\underline{e}\ \underline{k}\ j \text{ OF } B) \quad \rightarrow \quad \text{for } B[j]$$

Obviously the above declarations represent function definitions and to this end procedure templates must be created; in these creation processes the parameters of $\underline{e}$'s are evaluated. Now, on execution of h in the body of the program a specific value will need to be extracted from the array B, hence $\underline{k}B$ will need to be evaluated by $\underline{e}$. We cannot write $\underline{e}\ \underline{k}\ (\underline{e}\ \underline{k}\ j \text{ OF } B)$ since this would derive a value <u>immediately</u>; we need a notation which dictates that certain specified $\underline{e}$'s should be activated at one time and others later. One solution would be to replace each $\underline{e}$ by either $\underline{e}_1$ or $\underline{e}_2$ (say), so that:-

$$\underline{e}\ \underline{k}\ (\underline{e}\ \underline{k}\ i \text{ OF } A) \quad \text{becomes} \quad \underline{e}_1\ \underline{k}\ (\underline{e}_1\ \underline{k}\ i \text{ OF } A)$$

and

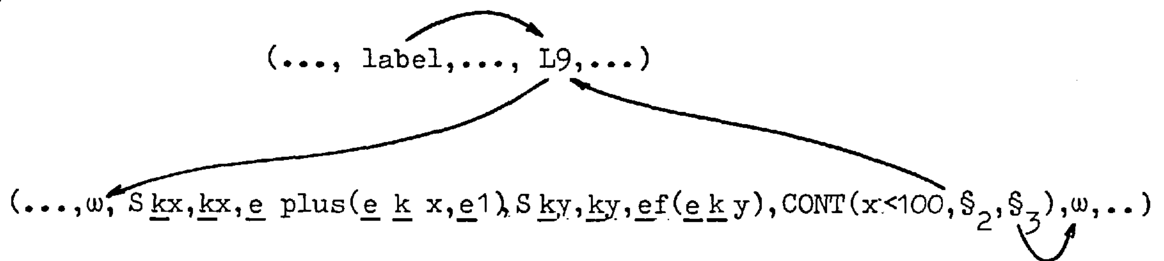
$$\underline{k}\ (\underline{e}\ \underline{k}\ j \text{ OF } B) \quad \text{becomes} \quad \underline{e}_2\ \underline{k}\ (\underline{e}_1\ \underline{k}\ j \text{ OF } B)$$

The distinction created here needs to be interpreted. Intuitively this is not difficult but we leave formalization until after we have given another example.

Consider the following:-

```
L9 : x : = x + 1 ;
     y : = f(y);
     if (x < 100) then goto L9;
```

If we wish to transform this segment of program into unoptimised 'goto-less' form (see Appendix, also [28, 29]) then, before control modifications, the situation may be represented in the following way†:-



In translating this into a post-check loop we have to take the two S-operations and form the body of the loop; extract the first parameter of CONT and use it for the post-check predicate; and replace the section

$$\omega, \ldots, \omega$$

by the resultant loop operation. Again we need to execute some operations but not all. Using $\underline{e}_1$ to denote translation activation, $\underline{e}_2$ to denote activation associated with execution of the object program, and informal English to describe the operations, we have:-

---

† Currently irrelevant details have been ignored or simplified. CONT(x,y,z) is used to represent a 'semi-translated' control statement which will later give rise to a RAM or PTL.

```
          e1 (insert (/) to left of (/),
                      /             /
                     /             /
                    /          e1k e1k (§2 OF CONT)
                   /
           e2 (repeat (/) until (/))
                       /       /
                      /     e1 (copy (§1 OF CONT))
                     /
                    /
             e1 (from (/) to (/))
                       /    /
                      /    /
                     /   e1 left of (CONT)
                    /
                   /
          e1 right of (e1k e1k (§2 OF CONT))
```

delete from ($\underline{e}_1$ left of repeat ( ) until ( )) to CONT))

Here, the stroke '/' is used to demonstrate the embedded structure and has no semantic meaning.

6.6.2 Discussion

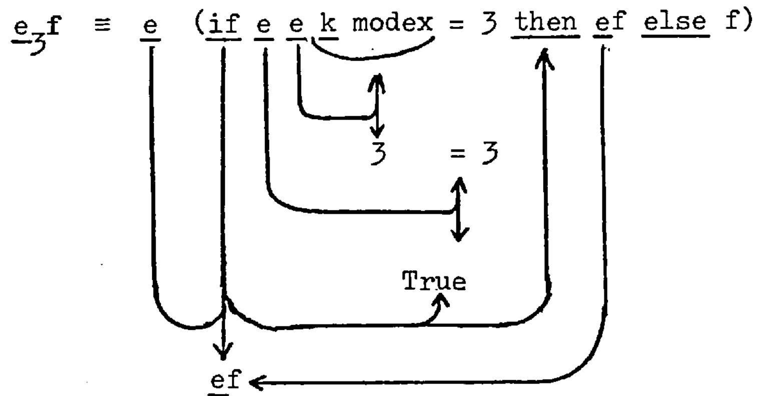
In the above cases it is possible to replace $\underline{e}_1$ and $\underline{e}_2$ by other constructs to achieve the same effect [31]. This is done by 'move' operations which when activated do not evaluate their argument (parameter); however, the techniques used in doing this are ad hoc. As our purpose is strictly definitional we shall not utilize these non-uniform methods but formally define $\underline{e}_i$.

Defn: Given an indicator modex which stipulates the mode of execution of the system, given as a positive integer than for any f

$$\underline{e}_i f \equiv \underline{e}\ (\underline{\text{if}}\ \underline{e}\,\underline{e}\,\underline{k}\ \text{modex} = i\ \underline{\text{then}}\ \underline{e}f\ \underline{\text{else}}\ f)$$

We explain this by two examples:

(i) Suppose 'modex = 3', then the evaluation of $e_3 f$ goes:-



The action of the e's in the definition is as follows:-

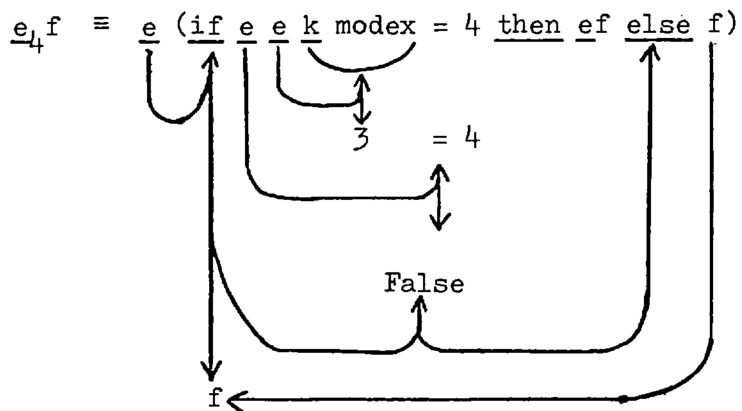
(i) 1st e activates if-then-else

(ii) 2nd e activates = (the predicate)

(iii) 3rd e activates k to yield a value

(iv) 4th e (conditionally) activates f

(ii) Under the same conditions $e_4 f$ is:-



As it stands, the definition of $e_i f$ given above yields either the value (result) of f or the identifier f. This we do not want - we require either the result of f or the string $e_i f$. Modifying the definition would lead to a circular argument so instead we

stipulate that the elaboration of the definition of $\underline{e}_i f$ is re-executed whenever necessary, i.e. we always consider $\underline{e}_i f$ and never the result of a previous elaboration.

Defining $\underline{e}_i$ allows the generation of a countable infinity of activation operators; however, recent work on a set of test languages [33, 34] has shown that only 2 or 3 such operators are needed in order to specify the semantics of a wide range of commonly occurring high-level language features.

6.7 On Position Specification and Value Selection in CPS

Taking up from the assertion at the beginning of the previous section, that e generates a value, and k, if not immediately preceded by e is interpreted as a position specifier (pointer or reference) within the program space, we return to our fundamental operator S. Recall that S may be, depending on its arguments, either a substitution operator or a selector.

As a selector:

Sxyz ≡ e if e (x = y) then z else x.

As a substitution operator:

Sxyz ≡ e if e (x = y) then replace x by z.

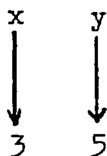
- here e(x = y) is the (boolean) result of the equality predicate on parameters x and y, and e if ... then ... denotes the result of the 'if' statement. Notice in particular that the result of the substitution S is either 'ω' or 'replace x by z' but not 'e replace x by z'.

Now, using the above definition we cannot, in general, distinguish between the two different usages of S. We need to know whether the arguments of S are 'values' or 'pointers'.

Consider the assignment:

$$x := y$$

This means, substitute the value of y for the value of x. If x and y have initial values 3 and 5 and are represented thus:



So, $\underline{k}x = 3$ and $\underline{k}y = 5$, but also $\underline{e}\,\underline{k}\,x = 3$ and $\underline{e}\,\underline{k}\,y = 5$. We have a paradox. We need to define precisely what the above assertions mean.

$\underline{k}\,x = 3$ means (i) there is a position related to x by $\underline{k}$

i.e.

and (ii) the position specified contains a representation of 3.

$\underline{e}\,\underline{k}\,x = 3$ means that the <u>result</u> of traversing the structure from x by $\underline{k}$ is 3; and this may henceforth be manipulated as a (single) value which no longer is related to x.

Clearly what is needed to model the effect of the assignment in Carabiner is

$$S\,\underline{k}\,x,\ \underline{k}\,x,\ \underline{e}\,\underline{k}\,y$$

or more generally

S position, position, value.

In the case of selection operations we must have a value as the first parameter, moreover for any comparison between the first two parameters to be meaningful, the second should also be a value (although when any parameter is conditional this may not be so, as we see below). The third parameter may be of any type. This justifies the definition of S given in §6.1.

To complete the formal distinction between pointers and values recall that values and position specifiers are incomparable and in the case where a specifier and a value have the same form we use $\underline{e}$ to 'break' a value from a position. The rationale for this is:-

$$\underline{e}\ x = \underline{ek}^{o}\ x = x \quad \text{as a value}$$

Recall the question posed in §6.2; given the CPS substate:-

$$\text{STORE IS } (\underset{\substack{\downarrow \\ 3}}{x}\ ,\ \underset{\substack{\downarrow \\ 3}}{y})$$

then $\underline{k}\,x = 3$ and $\underline{k}\,y = 3$ but is $y \in \underline{k}^{-1}\,\underline{k}\,x$?

Clearly, by the above discussion, it is not.

$\underline{k}^{-1}\underline{k}\,x = \{x\}$ because $\underline{k}\,x = 3$ only asserts that the position $\underline{k}$ from x contains the value 3, i.e.

$$\begin{array}{c} x \\ \downarrow \\ 3 \end{array}$$

hence $\underline{k}^{-1}$(this 3) = {x}. Similarly for y. In fact

	$x \neq y$	locations have different names
and	$\underline{k}\ x \neq \underline{k}\,y$	locations are different
but	$\underline{e}\,\underline{k}\ x = \underline{e}\,\underline{k}\,y$	contents of the location are equal

Using CPL [6] terminology a value is a right-hand value and a position specifier is a generalised left-hand value.

As a final example, consider the bi-conditional assignment statement:-

$$(a \rightarrow b,\ c) := (d \rightarrow e,\ f)$$

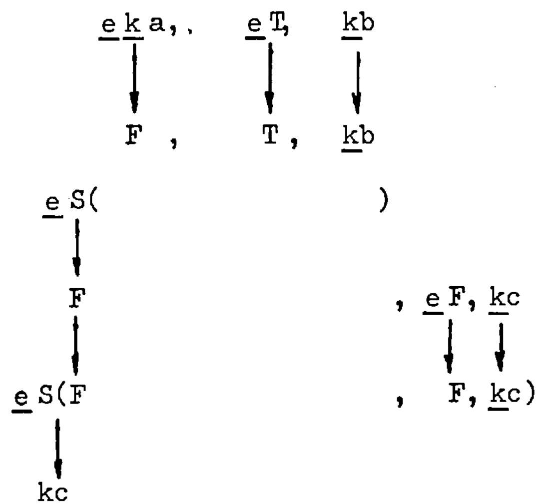
or, in Algolese

$$(\underline{if}\ a\ \underline{then}\ b\ \underline{else}\ c) := (\underline{if}\ d\ \underline{then}\ e\ \underline{else}\ f)$$

The left-hand side yields:-

$$\underline{e}S(\underline{e}S(\underline{e}\,\underline{k}\,a,\ \underline{e}T,\ \underline{k}b),\ \underline{e}F,\ \underline{k}c)$$

supposing that the value of a is F(alse) then



The hand side yields

$$\underline{e}S(\underline{e}S(\underline{e}\,\underline{k}\,d,\ \underline{e}T,\ \underline{k}f),\ \underline{e}F,\ \underline{k}g)$$

Now if we write

$$Sx,\ x,\ y \quad \text{as} \quad S_1x\ ,\ y$$

the whole expression can be assembled:

$$\underline{e}S_1(\underline{e}S(\underline{e}S(\underline{e}\,\underline{k}\,a,\ \underline{e}T,\underline{k}b),\ \underline{e}F,\underline{k}c),\underline{ee}S(\underline{e}S(\underline{e}\,\underline{k}\,d,\underline{e}T,\underline{k}f),\underline{e}F,\underline{k}g))$$

The operators $\underline{k}^{-1}$ and OF may also be used as position specifiers. $\underline{k}^{-1}$ may be used together with $\underline{e}$ in the same fashion as $\underline{k}$; however OF always (without $\underline{e}$!) yields a position, $\underline{e}$ being available to break the contained value from the structure, should this be required.

6.8 Macros and Set Theory.

So far we have formally defined only three basic Carabiner operations, and described the denotations used in representing other (language - or program-defined) procedures. Any other operations used in Carabiner, apart from those explicitly related to a given source language, we shall call macros. These include all the set and list operations, and some topological functions (e.g. STRUCT etc.) which we define, together with other operations that act upon the CPS, in §6.8.1.

In the subsequent section we examine the relationship between S and set theoretic operations hence we show that the list operations of §5.2.1 are a natural consequence of our fundamental operations and do not contradict our claim that these basic procedures are sufficient for the description of computing systems.

6.8.1 Operation on CPS.

Apart from the operations of §5.3 which act on the constituent lists and graphs (and sets) we may define new operations which act on the CPS, as follows.

To create a new link between two elements (nodes) of CPS, we augment the in/out-bundles of the relevant nodes i.e.:-

Defn:† LINK (x,y) : $\left\{ \begin{array}{l} \underline{k}x \mapsto \underline{k}x \cup \{y\} \\ \underline{k}^{-1}y \mapsto \underline{k}^{-1}y \cup \{x\} \end{array} \right\}$

Similarly:

Defn: BREAK (x,y): $\left\{ \begin{array}{l} \underline{k}x \mapsto \underline{k}x \setminus \{y\} \\ \underline{k}^{-1}y \mapsto \underline{k}^{-1}y \setminus \{x\} \end{array} \right\}$

Using the operations LINK and BREAK we can easily define any number of ADJustment operators which can be tailored to suit specific needs. Here we define only one but suggest how two others can be derived and used. Given four nodes, a, b, c, d in CPS such that $b \in \underline{k}a$

i.e.

Now define ADJ (a, b, c, d) $\equiv$ $\left\{ \begin{array}{l} \text{BREAK (a, b)} \\ \text{LINK (c, d)} \end{array} \right\}$

=

---

† Trivially either operand of LINK may be extended from a singleton to a larger set. This may be viewed as projecting a graph operation onto an operation on a related hypergraph [8].

Trivially, to reset the destination of a k link we could define -

ADJ1 (a, b, c) ≡ ADJ (a, b, a, c)

- or to reset the source -

ADJ2 (a, b, c) ≡ ADJ (a, c, b, c) etc.

One direct use of ADJ is in the specification of pointer (PTR) manipulation described in §6.5. The routines used therein, namely

Advance Parameter Pointer (APP)

and

Advance List Pointer (ALP)

can be defined thus:-

Given an active function template, F (say), then

APP ≡ ADJ(PTR, kPTR OF F, PTR, e(Next(F, kPTR OF F)))

ALP ≡ ADJ(PTR, kPTR OF kF, PTR, e(Next(kF, kPTR OF kF)))

Notice here that the 4th parameter could actually modify the 2nd if it was elaborated before the 2nd; hence the need for a definite order of elaboration. Without formality this is defined to be strictly from left to right.

Any element of CPS which is not in the lists STORE or ATTRIB must be accessed by reference to these lists and the use of the operations k, k^{-1} and OF (together with associated selector names). In general, this specifier is not unique, and if further we allow the use of the equivalence relations across ATTRIB then the set of items in CPS specifiable from a single element of one of the fundamental lists can be very large and is of little interest. However, we can extract some subsets of this collection of related items which will make manipulation of the CPS easier. Conceptually these constructs are simple but their formal definition is somewhat complex.

What we finally require are substructures of CPS such that given x OF STORE the related substructure is the part of CPS which has to be added to CPS in order to give the whole CPS from a state without x (or, equivalently, the substructure to be deleted if we wish to remove x); obviously this depends on the current state of CPS.

<u>Defn</u>. Given $x \in$ CPS

$$\text{deriv}(x) = \underline{k}^{-1}(x) \cup \{z : z \in \underline{k}(x) : x \notin \text{ATTRIB}\}$$
$$\cup \{z : z \in \underline{k}(y_i) \cup \underline{k}^{-1}(y_i),\ (1 \leq i \leq m) \text{ if } x \text{ IS } (y_1,\ldots,y_m) : x \notin \text{ATTRIB}\}$$

The above set comprises all elements of CPS which are one ($\underline{k}$) step away from x without traversing the equivalences of ATTRIB.

Now, from $x \in$ STORE we may construct

$$D_o(x) = \text{deriv}(x)$$

and $$D_n(x) = \{z : z \in \text{deriv}(y) : y \in D^*_{n-1}(x)\}$$

†where $D^*_n(x) = D_n(x) \setminus \text{STORE}^*$ for all $n \in \mathbb{N}$

Then $$\text{Rel}(x) = \bigcup_{n=o}^{\infty} D^*_n(x) \cup \{x\}$$

Rel(x) is then all the nodes and lists in CPS which are related to x via the relations $\underline{k}$, $\underline{k}^{-1}$ and OF (but not ≈) without returning to STORE.

Now form $\text{Rel}^*(x)$:

$$\text{Rel}^*(x) = \text{Rel}(x) \cup \{ (y,z) : y,z \in \text{Rel}(x) \text{ and } z = \underline{k}y\}$$

i.e. $\text{Rel}^*(x)$ yields all the elements of Rel(x) and their interconnecting structure (given by $\underline{k}$).

Obviously, given $x_1,x_2 : x_1 \neq x_2$, and $x_1,x_2 \in$ STORE, the related sets $\text{Rel}^*(x_1)$, $\text{Rel}^*(x_2)$ are in general not disjoint, however from $\text{Rel}(x_1)$ we can abstract a suitable structure thus:- for $x \in$ STORE

let $$\text{Nhd}(x) = \text{Rel}(x) \setminus \bigcup_{y \in \text{STORE} \setminus \{x\}} \text{Rel}(y)$$

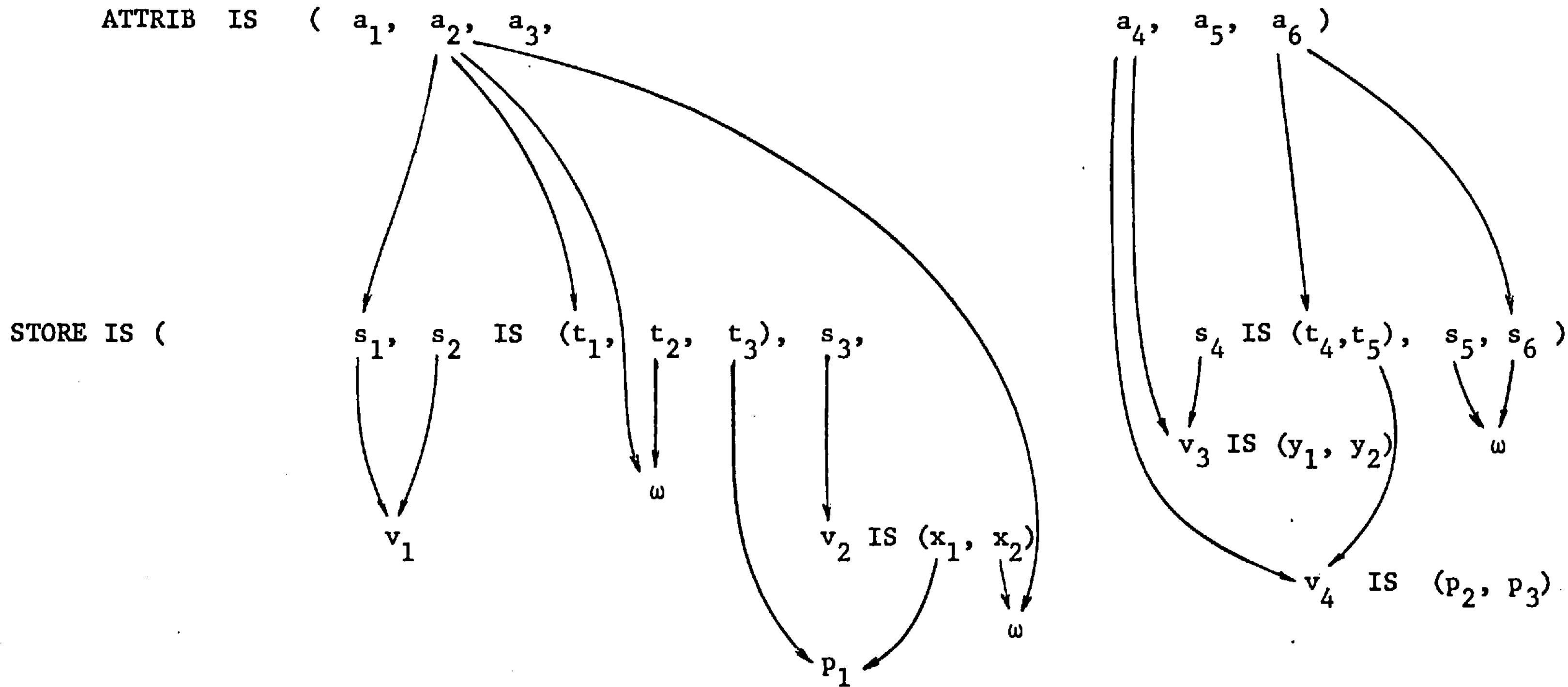
---

† $\text{STORE}^* = \{y : y \in y_1 \in \ldots \in y_n \in \text{STORE}\}$ for all $n \in \mathbb{N}$

then

$$STRUCT(x) = Nhd(x) \cup \{ (a,b) : (a,b) \in Rel^{*}(x) \text{ and } (a \in Nhd(x) \text{ or } b \in Nhd(x)) \}$$

STRUCT yields 'open' sets around each element of STORE. (This fact may be used to define a topological basis for CPS - see Chapter 8). In practical terms, manipulation of STRUCT corresponds to the duplication of a subgraph or deletion of an element in STORE; the mathematical formulation does, nevertheless, look somewhat over-powering. In an attempt to illustrate its inherent simplicity we give an example. The example is not intended to represent a specific point in the execution of a program and hence names etc. are meaningless except for STORE and ATTRIB.


ATTRIB IS (a1, a2, a3, a4, a5, a6)
STORE IS (s1, s2 IS (t1, t2, t3), s3, s4 IS (t4,t5), s5, s6)
ω
v1
v2 IS (x1, x2)
ω
p1
v3 IS (y1, y2)
ω
v4 IS (p2, p3)

$deriv(a_i \text{ OF ATTRIB}) = \emptyset \quad i \in (1,\ldots,6)$

$deriv(s_1) = \{a_2 \text{ OF ATTRIB}, \underline{k}s_1\}$

$deriv(s_2) = \{\underline{k}s_2, a_2 \text{ OF ATTRIB}, \underline{k}(t_2 \text{ OF } s_2), \underline{k}(x_1 \text{ OF } \underline{k}s_3)\}$

$deriv(s_3) = \{\underline{k}s_3\}$

$deriv(s_4) = \{\underline{k}s_4, a_6 \text{ OF ATTRIB}, \underline{k}(t_5 \text{ OF } s_4)\}$

$deriv(s_5) = \{\underline{k}s_5\}$

$deriv(s_6) = \{a_6 \text{ OF ATTRIB}, \underline{k}s_6\}$

Before proceding further, we note that the node names (other than selections and elements of ATTRIB and STORE) $v_1,\ldots,v_4$ and p_1 are added for convenience and ω denotes a void entity. The derived sets given above are in their strict form but for ease of exposition we shall write, e.g.

$deriv(s_4) = \{v_3, a_6, v_4\}$

Using this principal where applicable, and also disregarding specifiers for arguments of 'deriv', we have:-

$deriv(\underline{k}s_1) = deriv(v_1) = \{s_1, s_2\}$

$deriv(v_2) = \{s_3, p_1, \underline{k}x_2\}$

$deriv(v_3) = \{a_4, s_4\}$

$deriv(v_4) = \{a_4, t_5\}$

$deriv(t_1) = \{a_2\}$

$deriv(t_2) = \{\underline{k}a_2\}$

$deriv(t_3) = \{p_1\}$

$deriv(t_4) = \{a_6\}$

$deriv(t_5) = \{v_4\}$

$deriv(p_1) = \{t_3, x_1\}$

$deriv(p_2) = deriv(p_3) = \emptyset$

$deriv(y_1) = deriv(y_2) = \emptyset$

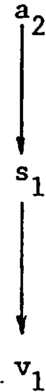
$$\text{deriv}(x_1) = \{p_1\}$$
$$\text{deriv}(x_2) = \{\underline{k}x_2\}$$
$$\text{deriv}(\underline{k}t_2) = \{t_2, a_2\}$$
$$\text{deriv}(\underline{k}x_2) = \{x_2, a_2\}$$
$$\text{deriv}(\underline{k}s_5) = \text{deriv}(\underline{k}s_6) = \{s_5, s_6\}$$

Then:-

$$D_o(s_1) = \{a_2, v_1\}$$
$$D_o^*(s_1) = D_o(s_1) = \{a_2, v_1\}$$
$$D_1(s_1) = \emptyset \cup \{s_1, s_2\}$$
$$= \{s_1, s_2\}$$
$$D_1^*(s_1) = \emptyset$$

hence: $\text{Rel}(s_1) = \{s_1, a_2, v_1\}$

and $\text{Rel}^*(s_1)$ is :-

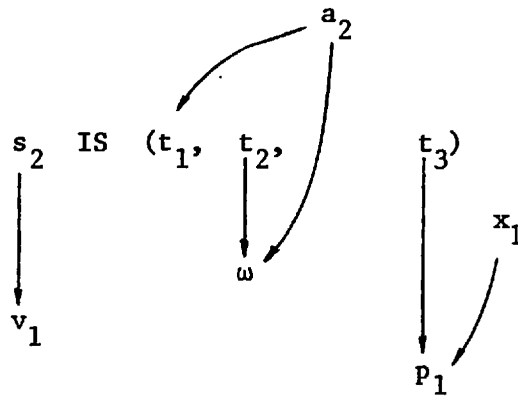


Similarly:

$$D_o(s_2) = \{v_1, a_2, \underline{k}(t_2), p_1\} = D_o^*(s_2)$$
$$D_1(s_2) = \{s_1, s_2, t_2, a_2, t_3, x_1\}$$
$$D_1^*(s_2) = \{a_2, x_1\}$$
$$D_2(s_2) = \{p_1\} \subset D_o(s_2)$$

$\therefore$ $\text{Rel}(s_2) = \{s_2, v_1, a_2, \underline{k}t_2, p_1, x_1\}$

and $Rel^*(s_2)$ is



$D_o(s_3) = \{v_2\} = D_o^*(s_3)$

$D_1(s_3) = \{s_3, p_1, \underline{k}x_2\}$

$D_1^*(s_3) = \{p_1, \underline{k}x_2\}$

$D_2(s_3) = \{t_3, x_1, x_2, a_2\}$

$D_2^*(s_3) = \{x_1, x_2, a_2\}$

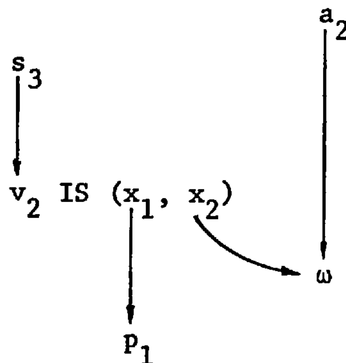
$D_3(s_3) = \{p_1, \underline{k}x_2\} = D_3^*(s_3)$

$D_4(s_3) = \{t_3, x_1, x_2, a_2\}$

$D_4^*(s_3) = \{x_1, x_2, a_2\} = D_2^*(s_3)$

$\therefore$ $Rel(s_3) = \{s_3, v_2, p_1, \underline{k}x_2, a_2\}$

$Rel^*(s_3)$ is:-

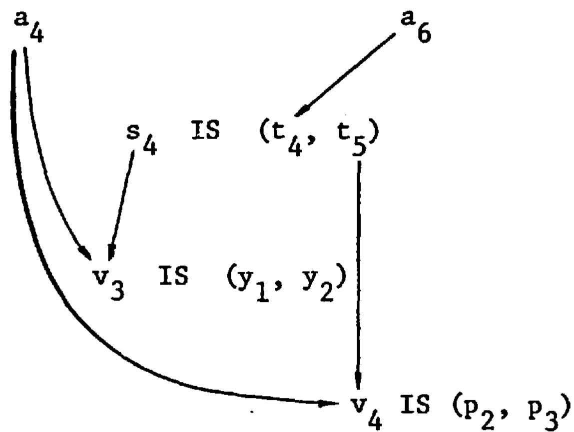


Similarly:

$Rel(s_4) = \{s_4, v_3, a_4, a_6, v_4\}$

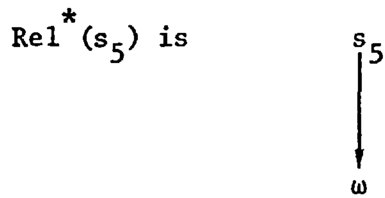
and

$Rel^*(s_4)$ is:-

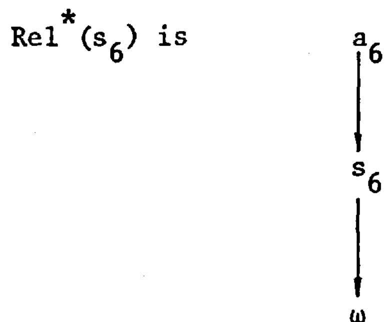


and trivially:

$Rel^*(s_5)$ is



$Rel^*(s_6)$ is



Then:

$Nhd(s_1) = \{s_1\}$

$Nhd(s_2) = \{s_2, \underline{kt}_2\}$

$^{\dagger}Nhd(s_3) = \{s_3, v_2, \underline{k}x_2\}$

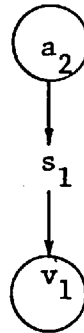
$Nhd(s_4) = \{s_4, v_3, v_4, a_4\}$

$Nhd(s_5) = \{s_5\}$

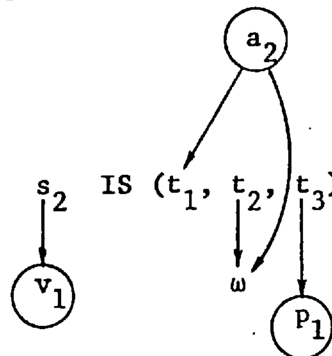
$Nhd(s_6) = \{s_6\}$

Now, finally, if we adopt the convention that elements not in STRUCT but which must be specified so as to indicate a link (pair) in STRUCT are depicted by encircled symbols then:-

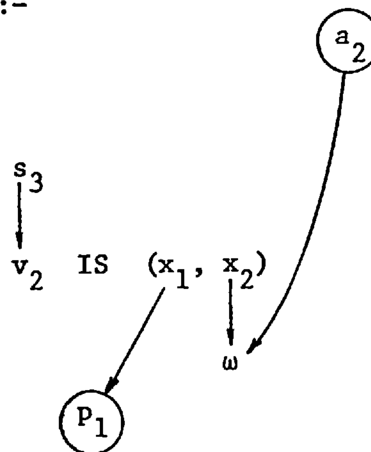
$STRUCT(s_1)$ is:-



$STRUCT(s_2)$ is:-

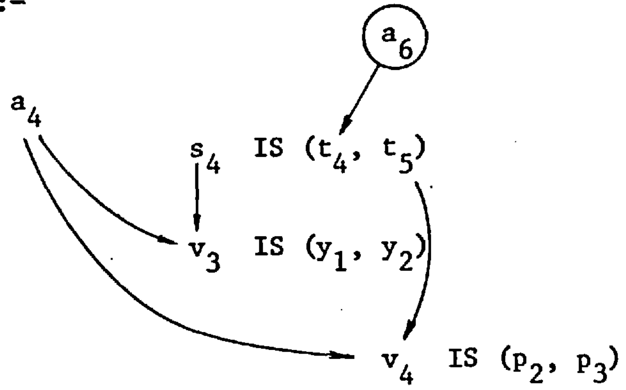


$STRUCT(s_3)$ is:-



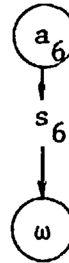
† Notice $v_2 \in Nhd(s_3)$ and by definition v_2 IS (x_1, x_2); this implies $x_1 \in Nhd(s_3)$ although $x_1 \in Rel(s_2)$.

STRUCT(s_4) is:-



STRUCT(s_5) is:-

STRUCT(s_6) is:-



6.8.2 The S definition of set operations and predicates.

From the definition of S we can easily model the conditional statement:

$\underline{\text{if}}\ \alpha\ \underline{\text{then}}\ \beta\ \underline{\text{else}}\ \gamma$

by $S(\underline{e}S\ \underline{e}T,\ \underline{ek}\alpha,\beta),\ \underline{e}T,\ \gamma$

where α is a Boolean value, expression or function and β, γ are not Boolean values; however, if we are in a situation where such Boolean values can arise then we need to use the rather more complex form:

$S\underline{e}(\underline{e}S\ \underline{e}T,\ \underline{ek}\alpha,\beta),\ \underline{e}T,\ \underline{e}(S\ \underline{ek}\alpha,\ \underline{e}F,\ \gamma)$

The validity of these forms can be checked by examination of truth-tables for S or by a simple proof [96].

The conditional expression (or statement) given above may also be written as a 'McCarthy conditional', i.e.

$(\underline{ek}\alpha \rightarrow \beta,\ \underline{e}T \rightarrow \gamma)$

or, less formally:-

$(p_1 \rightarrow q_1,\ T \rightarrow q)$

where p_1 is a Boolean value and q, q_1 can be anything.

More complex conditionals can be built up by embedding constructs of this type, e.g.

$\underline{\text{if}}\ p_1\ \underline{\text{then}}\ (\ \underline{\text{if}}\ p_2\ \underline{\text{then}}\ q_2\ \underline{\text{else}}\ q\ )$

becomes:-

$(p_1 \rightarrow (p_2 \rightarrow q_2,\ T \rightarrow q),\ T \rightarrow \Lambda).$

The derivation of an equivalent S-expression is trivial and is not here developed.

We now consider the predicate $\in$, i.e. does $x \in A$ for any given x and A. - to make the discussion less formal we consider only values and ignore the operators e and k -

if $A = \{y\}$

then $x \in A \Leftrightarrow x = y$

i.e. if $x = y$ then True else False

Now the predicate '=' may be defined in several ways.

If x and y are †Boolean then we may use the predicate $x \equiv y$, i.e.

S (S T x y) T (S T y x)

- if x is not †Boolean then we may use

S (S x y T) x F .

The last definition also works when x is False, the trouble arising when x is True.

If we have a denotation, α , for an object not in the set of program-defined elements, then we may use this to represent the value True by

S a T α, i.e. if a = T then α else a

hence the required equality predicate (for all program-defined parameters) could be represented by:

$(a = b) \doteqdot$ S (S (S a T α) (S b T α) T) (S a T α) F

---

† ' x is Boolean' may be checked by using

B(x) = S T (S T (S x V(x)T) F) F

where

V(x) = S T (S T x F) F.

Returning to sets:-

if $A = \{y\}$

then $x \in A$ iff $x = y$

and $x \notin A$ iff $x \neq y$

where $(x \neq y) \equiv ST\ (x = y)\ F$

Trivially if $A = \emptyset$

then $x \notin A \quad \forall x$

Given any non-void set A i.e.

$$A = \{y_1, \ldots, y_n\}$$

then the list

$$A' = (y_1, \ldots, y_n)$$

is derived from A and has an explicit order relation. Since any representation† of a set has an implicit ordering, from hereon we regard sets as represented by a corresponding list.

So: $x \in (y_1, \ldots, y_n)$

$$\equiv \underline{\text{if}}\ x = y_1\ \underline{\text{then}}\ \text{True}\ \underline{\text{else}}\ x \in (y_2, \ldots, y_n)$$

$$\equiv (x = y_1) \vee (x = y_2) \vee \ldots \vee (x = y_n)$$

$$\equiv \bigvee_{1 \leq i \leq n} (x = y_i)$$

Although the recursive definition is most informative, in the sense that when the element is found the search stops; the iterative form is more realistic in terms of implementation. Problems of indexing have been ignored, but the extension of S to act on (finite but unbounded) lists would easily circumvent these, i.e.

$S\ x, y, z$

---

† as used in a computing system.

where x is {a, b, c} or (a, b, c) could be

$$S \begin{Bmatrix} a \\ b \\ c \end{Bmatrix}, y, z$$

$$\equiv \begin{Bmatrix} S\ a,\ y,\ z \\ S\ b,\ y,\ z \\ S\ c,\ y,\ z \end{Bmatrix}$$

Notation and procedures for manipulating such lists would then have to be developed.

Tacitly we assume that the length of a list is easily obtainable from the representing system and hence we could embed any equality predicate in a suitable loop e.g.

(<u>for</u> i := 1, 1, length <u>do</u> (<u>if</u> x = y_i <u>then</u> (result:= True, exit)))

Obviously: $A \cap B = \{ x: x \in A, x \in B \}$

$A \cup B = \{ x: x \in A \text{ or } x \in B \}$

$A \setminus B = \{ x: x \in A \text{ and } x \notin B \}$

- are all derivable in a similar way.

Now given a list $A = (a_1, \ldots, a_n)$ and $a_i, a_j \in A$, then

$$a_i \underset{A}{<} a_j \text{ iff } a_i \in (a_1, \ldots, a_{j-1})$$

so, within a length-bounded i-indexed loop we could have

(<u>if</u> $a_i = a_\ell$ <u>then</u> (result:= True, exit)

<u>if</u> $a_i = a_j$ <u>then</u> (result:= False, exit)

)

Hence list (and implicit set) order-related predicates are also definable in terms of S .

6.8.3 On S abbreviations and pointer/value associations.

To complete our notes on macros we define the operators S_1 and S_2. These are merely special cases of S used when the parameters of S take particularly simple forms.

Notn: for S x, x, y write S_1 x, y ((re-)initialization)

for S x, x, $\underline{e}\Lambda$ write S_2 x (deletion).

- here x, y may be general elements of CPS, elements of STORE or STRUCTures derived from STORE elements

$$S_1\ \Lambda, x \equiv \mathrm{AUGR}(\mathrm{STORE}, x)$$

At this point we specify the pointer/value associations of all operations previously defined. In what follows these associations will not be checked but may be used by the reader to clarify the meaning (effect) of complex constructions. Here p and v denote objects of type pointer and value respectively.

AUGL/R(p, v)

INSERTL/R(p, p, v)

COPYL/R(p, v) $\rightarrow$ v

DELL/R(p, v)

TRIML/R(p, v)

v OF p $\rightarrow$ p

$\underline{k}$p $\rightarrow$ p

Next(p, p) $\rightarrow$ p

Prev(p, p) $\rightarrow$ p

$K^+p \rightarrow v$

$K^-p \rightarrow v$

LINK(p, p)

BREAK(p, p)

STRUCT(p) $\rightarrow$ v

ADJ(p, p, p, p)

APP(p, p)

ALP(p, p) .

CHAPTER 7

TRANSLATION AND A FORMAL DEFINITION OF X

7.1 Parsing Strategy

Compilation

We briefly describe the three phases of compilation, namely; parsing, validation and translation. These may be thought of as disjoint activities although in practice they are more efficiently performed concurrently. The descriptions given in this section do not specifically relate to the language X. Of particular note is the subject of type-checking validations. These can be checked dynamically at execution time but in general are more efficiently performed (if the language allows) statically during translation. (see §9.4).

The syntax analysis may be performed by any suitable method; however, the analysis is greatly simplified if sentences in the languages may be parsed directly from left to right (see [17,46,56,57,59]). The construction of a suitable grammar (which may generate a 'larger' language than is required) was discussed at length in Chapter 2. During parsing we develop and, later, contract numerous inter-related trees. The nodes of these trees are created by the parser and used in the validation and translation phases (see below). In what follows, only nodes of trees of incomplete syntactic classes (non-terminals) can be referred to by their syntactic index (e.g. $< \; >_{9.3}$), other quantities (if required) must be stored explicitly by the compiler.

Validation

In the main, this involves the checking of attributes to ensure that any context-sensitive restrictions which are permitted by the pure syntax [16], but which do not make sense are detected and/or removed before the current syntactic phrase is translated or executed. This may have to be done at compilation time or in execution; in the latter case the appropriate check must be embedded within the resultant Carabiner object program. We may specify these validity checks by using S with logical parameters.

This would make the overall approach more uniform but less simple to comprehend and hence we use set theoretic notation to facilitate readability.

E.g. given $\underline{k}\ x \in \underline{k}\ \underline{real}$ OF ATTRIB

and $\underline{k}\ y \in \underline{k}\ \underline{int}$ OF ATTRIB

then $y := x$

is syntactically correct in X but invalid semantically, this check being performed by the schema:-

$((\underline{real}$ OF ATTRIB$) \cup (\underline{int}$ OF ATTRIB$)) \cap \underline{k}^{-1}\underline{k}\ y = \emptyset$?

yes

no

$\underline{real} \in \underline{k}^{-1}\underline{k}\ y$?

no

yes

$\underline{k}\ x \in \underline{k}\ \underline{int}$?

no

yes

$\underline{k}\ x \in \underline{k}\ \underline{real}$?

yes

no

$\underline{k}\ x \in \underline{k}\ \underline{int}$?

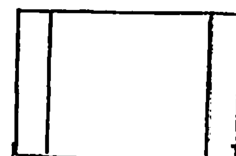
yes

no

Convert by a_{IR}

assign

exit

In the above diagram the boxes drawn thus  represent translation steps.

We could at this stage give elaborate 'ad hoc' methods for the validation of an X-program: however, a more rational and uniform technique is desirable, in which 'modal substructures'† of CPS may be used statically or dynamically to validate polymorphic objects [68]. The specification of such structures and their related operations lies beyond the scope of the current chapter but is dealt with in §9.4.

Translation

The translation phase is merely a syntax-driven compiler [17,56,57] which operates on 'standard' BNF and is defined by sequences of injections (actions) implanted within the right-hand sides of the syntax productions. These injections are delimited by meta-brackets thus:-

$$\rightarrowtail \quad .... \quad \leftarrowtail$$

Any string delimited in this way is ignored by the parser but is, subject to the satisfaction of validation predicates, added to the translator's output stream.

This output stream, which may temporarily hold sets of partially translated elements, ultimately consists of a sequence of Carabiner operations and is placed at the node a of CPS. It is activated by the distributive e operation which constitutes the execution phase of the program.

---

† Informally this is a connected subgraph of CPS used to define a complex mode type.

7.2 The Semantic Injections of Language X.

Initial Structure

The routines used in the validation of an X program and the execution of the corresponding resultant Carabiner program need to be incorporated (as part of the initial structure) in CPS. However, to keep this example as simple as possible, we here discuss only the composition of CPS as determined by the standard prelude for X.

We describe the construction of the initial CPS step by step and give a diagramatic representation of the full initial state at the end of this section.

Notation:

$$\mathrm{LOC(a)} \stackrel{\mathrm{def}}{=} \mathrm{AUGR(STORE,\ a)}$$

$$\mathrm{HEAP(a)} \stackrel{\mathrm{def}}{=} \mathrm{AUGL(STORE,\ a)}$$

$$\mathrm{AT(a)} \stackrel{\mathrm{def}}{=} \mathrm{AUGR(ATTRIB,\ a)}$$

The Construction:

The fundamental objects in CPS are the lists ATTRIB and STORE

ATTRIB IS ( )

STORE IS ( )

The explicit modes of language X are <u>real</u> and <u>integer</u>, so:

AT((<u>real</u>, <u>integer</u>))

The nameless procedure, pa, needs to be set in STORE. In this language it takes no parameters and has a void[†] result, hence:

---

† For a quantity of (implicit) type void we have two possibilities :-

(i) <u>void</u> $\in$ ATTRIB
and
LINK(<u>void</u> OF ATTRIB, quantity)

(ii) <u>void</u> $\notin$ ATTRIB
and
ATTRIB $\cap$ $\underline{k}^{-1}$ (quantity) = $\emptyset$

Here we use the first characterization of the <u>void</u> mode but make no general pronouncement on the problem.

HEAP($p\alpha$)

$p\alpha$ IS ( )

$S_1\underline{k}p\alpha$ ($\alpha_1,\alpha_2,\alpha_3,\alpha_4$)

where †† α_1 = INSERTL(STORE, $p\alpha$, OUTPUT FROM $p\alpha$)

†† α_2 = OUTPUT FROM $p\alpha$ IS ( )

†† α_3 = LINK(void OF ATTRIB, OUTPUT FROM $p\alpha$)

α_4 = BLOCK(τ)

AT((proc, void))

LINK(proc OF ATTRIB, a)

The most complex enhancement to the structure is the embedding of what McCarthy calls the 'base functions' [75]. In this example these are the X-operations of the diadic plus (for integers and reals) and the conversion routine from integer to real, and the Carabiner operation PTL† (for details of these see §6.5.2). The required extension to the CPS is brought about by the following sequence of Carabiner statements:-

HEAP({pa_{IR}, pa_{R+}, pa_{I+}})

pa_{IR} IS ($\S_1$)

pa_{R+} IS ($\S_1$, $\S_2$)

pa_{I+} IS ($\S_1$, $\S_2$)

LINK(proc OF ATTRIB, {a_{IR},
a_{R+},
a_{I+} })

[By virtue of the operator p, we implicity have

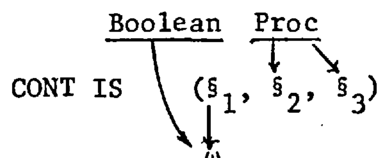
$S_1\underline{k}pa_?$, $a_?$

LINK($pa_?$, $a_?$) for $\forall a_?$]

---

†† Since there is no I/O for an X-program it may sensibly be regarded as a subroutine instead of a function and hence these steps could be disregarded. In general, of course, this is not true.

† This implies also that the 'dummy' operation CONTrol needs to be set up with a template:-



To simplify discussion this template is ignored in the examples.

LINK(int OF ATTRIB, {$\S_1$ OF pa_{IR},
$\S_1$ OF pa_{I+},
$\S_2$ OF pa_{I+} })

LINK(real OF ATTRIB, {$\S_1$ OF pa_{R+},
$\S_2$ OF pa_{R+}, })

HEAP(PTL)

PTL IS $(\S_1, \S_2)$

AT(Bool)

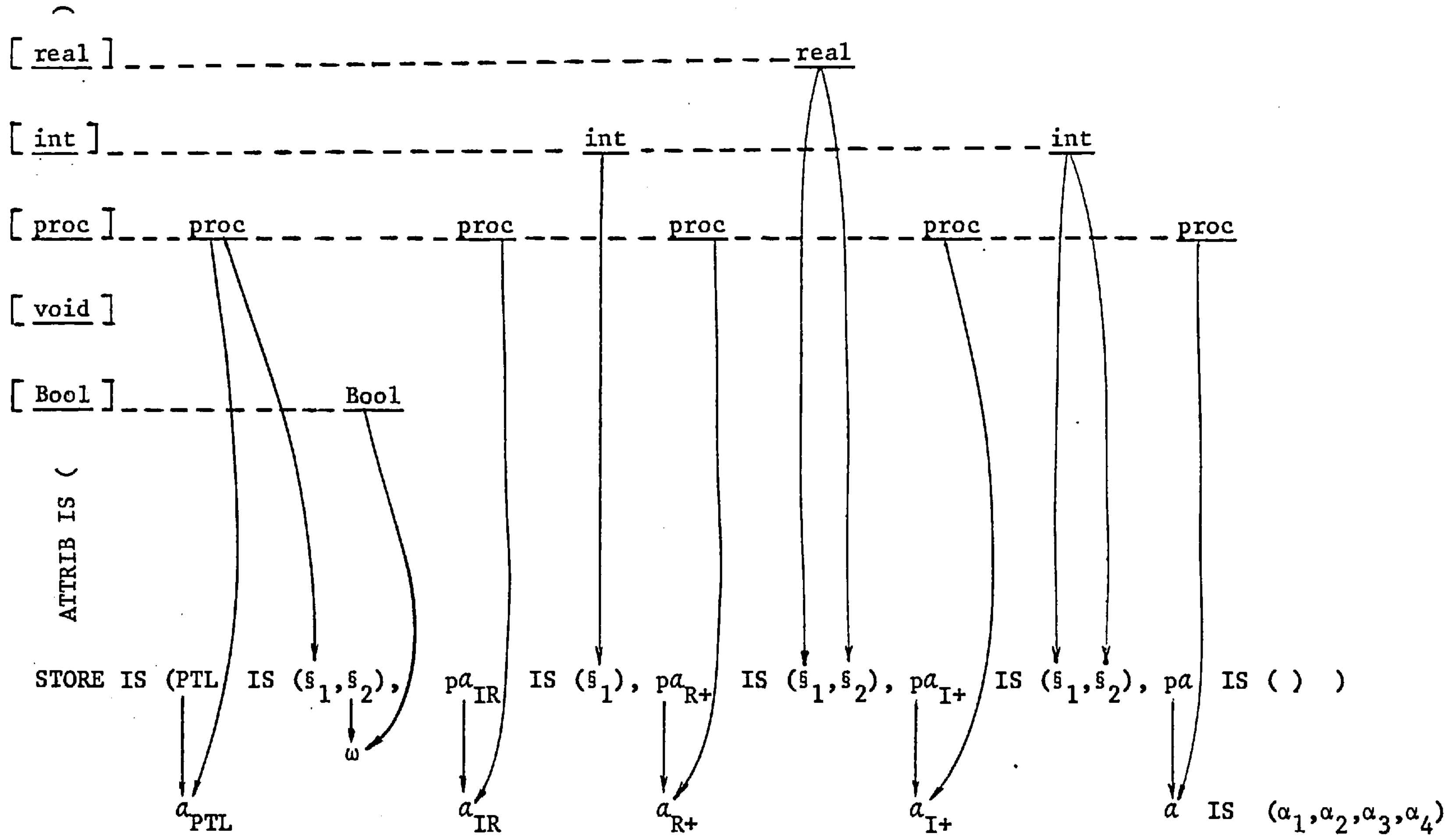
$S_1\underline{k}$ PTL, a_{PTL}.

LINK(proc OF ATTRIB, {PTL, $\S_1$ OF PTL })

$S_1\underline{k}(\S_2$ OF PTL), ω

LINK(Bool OF ATTRIB, $\underline{k}(\S_2$ OF PTL))

Diagramatically, the CPS now looks like:-



[real]
[int]
[proc]
[void]
[Bool]
real
int
int
proc
proc
proc
proc
proc
Bool
ATTRIB IS (
STORE IS (PTL IS (§1,§2), pa IR IS (§1), pa R+ IS (§1,§2), pa I+ IS (§1,§2), pa IS ( ))
ω
a PTL
a IR
a R+
a I+
a IS (α1,α2,α3,α4)

Here solid lines denote the relation k and the broken lines represent the equivalence relation generated by ATTRIButes over CPS.

SEBNF Definition of language X

Here we give the definition of X in semantically extended BNF. In the definition the predicates are described in an informal Algol/set-theoretic form and all type-checking and validation is done dynamically. SEBNF is BNF together with DTC's (see Chapter 2) and the meta symbols:-

�under ... ... ⋔

Occurence of a string enclosed by such brackets denotes immediate activation of the string by a distributive e. Further definition is not given but illustrative examples will be given later to clarify its meaning.

In what follows particular note should be taken of the use of e, k, and the distinctions between:-

f, kf, f(x), ef(x)

1) <program> ::= $\langle \underline{\text{block}} \rangle_{1.1}$

S_1 (a, STRUCT(BLOCK)),

S_1 (BLOCK, Λ)

eek$a$

2) <block> ::= begin

⋚ AT($\updownarrow\phi$),

LOC($\updownarrow\phi$)

AT(label),

LOC(LOC($\updownarrow\phi$)) ⋛

$\langle \text{decn} \rangle_{2.1}$

;

$\langle \text{stmts} \rangle_{2.2}$

end

≯ block exit ≮[†]

3) <decn> ::= let ≯ LOC(AT(declist)) ≮
<proper id list>$_{3.1}$
be
<type>$_{3.2}$
≯ LOC(LOC(ek(declist OF ATTRIB))),
LOC(LINK(< >$_{3.2}$ OF ATTRIB, ek(declist OF ATTRIB))),
S_2(ek(declist OF ATTRIB)),
S_2(declist OF ATTRIB) ≮.

4) <proper id list> ::= <id>$_{4.1}$
≯ LOC(< >$_{4.1}$),
LINK(declist OF ATTRIB, < >$_{4.1}$) ≮
< id list >$_{4.2}$

5) <id list> ::=,
<proper id list>$_{5.1}$ |
Λ

6) <type> ::= real |
int

7) <stmts> ::= <stmt >$_{7.1}$ <stmts 1>$_{7.2}$

8) <stmts 1 > ::= ;
<stmts>$_{8.1}$ |
Λ

---

† In contrast to most other semantic injections, this is not realised by only a few statements but by a rather large program which restructures the whole of the preceding block by a subalgorithm of that given in §7.2.1.

```
9)  <stmt>   ::=  <block>9.1 |
                  goto
                  <label>9.2
                  } if (< >9.2 ∉ ek(label OF ATTRIB))
                       then (AT(< >9.2),
                             LINK(label OF ATTRIB, < >9.2 OF ATTRIB) )
                    AUGR(BLOCK, e CONT (True, §2, Λ)),
                    LINK(Bool OF ATTRIB, §1 OF (CONT OF BLOCK)),
                    LINK(§2 OF (CONT OF BLOCK),< >9.2 OF ATTRIB) { |
                  <id 1>9.3
                  :=
                  } LOC(LOC(ework)) {
                  <exp>9.4
                 †} LOC(if (TYPE(< >9.3) = int)
                         then (if (TYPE(work) = int )
                              then (S1 (k < >9.3, ekwork) )
                              else EXIT (T)
                                        )
                         else (if (TYPE(< >9.3) = real)
                              then (if (TYPE(work) = int )
                                   then ADJ(int OF ATTRIB, kwork, real
                                                       OF ATTRIB, kwork)
                                        S1 (kwork, S1paIR(kwork))
                                                                  )
```

---

† TYPE(x) ≡ $(\underline{k}^{-1}\underline{k}x) \cap$ ATTRIB

The required dynamic type checks and transfers can be deduced from the following table for x := y

TYPE (y)

TYPE(x)	:=	int	real	other
	int	assign	error	error
	real	convert assign	assign	error
	other	error	error	error.

```
                                     else (if (TYPE(work) = real)
                                              then Λ
                                              else EXIT (T)
                                                          )
                                S1(k < >9.3, ekwork)
                                                   )
                                                   )
                       S2 work) ⊀ |
                 L
                 <special>9.5

10)  <id 1 >    ::=   A |
                      B |
                      C

11)  <id>  ::=  A |
                B |
                C |
                L

12)  <label>  ::=  L
                   <digit>12.1
                   <rest of int>12.2

13)  <special>  ::=  <digit>13.1
                     <rest of int>13.2
                     <unlab. stmt>13.3 |
                     := ⊁ LOC(LOC(ework) ) ⊀
                     <exp>13.4
                    †⊁LOC(if(TYPE(L) = int)
                            then(if(TYPE(work) = int)
                                   then( S1(kL,ekwork) )
                                   else EXIT (T)
                            else(if(TYPE(L) = real)
```

† see production 9.

```
                                        then(if(TYPE(work) = int)
                                               then ADJ(int OF ATTRIB, kwork,
                                                      real OF ATTRIB, kwork)
                                                   S1(kwork,S1paIR(kwork) )
                                                                          )
                                               else (if(TYPE(work) = real )
                                                      then Λ
                                                      else EXIT (T)
                                                                   )
                                             S1(kL,ekwork)
                                                          )
                                                          )

                               S2 work)⊀

14)  <digit>  ::=  Ø | 1 | 2 | ... | 9

15)  <rest of int.>  ::= <digit>15.1
                         <rest of int.>15.2|
                         Λ

16)  <unlab. stmt>  ::= <block>16.1|
                        goto
                        <label>16.2
                        ⊁if(< >16.2⊄ek(label OF ATTRIB) )
                           then (AT(< >16.2),
                                 LINK(label OF ATTRIB, < >16.2 OF ATTRIB) ),
                         AUGR(BLOCK, e CONT(True,§2,Λ) ),
                         LINK(Bool OF ATTRIB, §1 OF(CONT OF BLOCK) ),
                         LINK(§2 OF(CONT OF BLOCK, < >16.2 OF ATTRIB) ⊀ |
                       <id>16.3
                       :=
                       <exp>16.4
                      †⊁LOC(if(TYPE(< >16.3) = int)
                              then (if(TYPE(work) = int)
```

---

† see production 9.

```
                              then(S1(k < >16.3,ekwork) )
                              else EXIT (T)
                                        )
                    else (if(TYPE(< >16.3) = real)
                              then(if(TYPE(work) = int)
                                   then ADJ(int OF ATTRIB, kwork,
                                            real OF ATTRIB, kwork)
                                         S1(kwork,S1paIR(kwork) )
                                                                )
                                   else (if(TYPE(work) = real)
                                            then Λ
                                            else EXIT (T)
                                                         )
                         S1(k< >16.3,ekwork)
                                        )
                                        )
               S2 work) {

17)  <exp> ::= <token>17.1
               <exp follower>17.2

18)  <exp follower> ::= +
                  }LOC(LOC(work)) {
                  <exp>18.1
                 †}LOC(if(TYPE(Prev(STORE,work) = int)
                       then(if(TYPE(work) = int)
```

† This insertion may be more easily comprehended after consultation of the following table:-

result := x + y

TYPE(y)

TYPE(x)	+	int	real	other
	int	a_{I+}	Convert x→R a_{R+}	✗
	real	Convert y→R a_{R+}	a_{R+}	✗
	other	✗	✗	✗

```
                              then S1(kPrev(STORE,work),
                                eS1pa_I+(ekPrev(STORE,work),ekwork))
                              else (if(TYPE(work) = real)
                                   then(ADJ(int OF ATTRIB,
                                      kPrev(STORE,work),
                                      real OF ATTRIB,
                                      kPrev(STORE,WORK)),
                                        S1kPrev(STORE,work),
                                     e(S1pa_IR(ekPrev(STORE,work))),
                                        S1kPrev(STORE,work),
                           e(S1pa_R+,(ekPrev(STORE,work),ekwork))
                                   else EXIT (T)
                      else (if (TYPE(Prev(STORE,work) = real)
                              then(if(TYPE(work) = real)
                                     then S1kPrev(STORE,work),
                         e(S1pa_R+, (ekPrev(STORE,work), ekwork))
                                     else if (TYPE(work) = int)
                                            then(ADJ(int OF ATTRIB,
                                     kwork, real OF ATTRIB, kwork),
                                                S1kwork,e(S1pa_R+,
                                     (ekPrev(STORE,work), ekwork))
                                            else EXIT (T)
                              else EXIT (T)
                    S2work≮ | Λ

19)  <token>  ::= <id>19.1
                  ≯S1 kwork, ek < >19.1,
                   LINK(TYPE(< >19.1), kwork ≮ |
                   .<digit>19.2<rest of int>19.3
                   ≯S1kwork, e(.< >19.1< >19.3),
                   LINK(real OF ATTRIB, kwork) ≮ |
                  <digit>19.4
                  <rest of int.>19.5
                  <rest of number>19.6
```

20) <rest of number> ::= .

<rest of int.>$_{20.1}$

≯S_1kwork, e(< >$_{19.5}$.< >$_{20.1}$),

LINK(real OF ATTRIB, kwork) ≮ |

Λ

≯S_1kwork, e(< >$_{19.4}$< >$_{19.5}$),

LINK(int OF ATTRIB, kwork) ≮

7.2.1 Control Translations in X.

To aid control translation we use a (purely notational) 'function'.

CONT(α, β, γ).

This function may accur as an item in STORE and corresponds loosely to

RAM(α, β, γ).

α is a predicate, so

$\underline{k}\alpha \in \underline{k}$ Bool

β is a pointer such that

$\underline{k}\beta \in$ STORE

or

$\underline{k}\beta \in$ ATTRIB

and

$\underline{k}^2\beta \in$ STORE

(The latter case models a label, ℓ, held in ATTRIB such that $\underline{k}\ell$ is the first statement in the list 'named' by ℓ.)

Similarly for γ .

CONT differs from RAM in that β, γ are not arguments of the procedure but merely pointers; moreover, if $\leq_s$ is the list order relation of STORE and

$\underline{k}\beta \leq_s$ CONTrol(α, β, γ)

then the resultant control routine may not be RAM but PTL (see appendix).

CONTrol is primarily used to emulate the underlying flow schema of a program and its modification by a suitable algorithm into an equivalent semi-structured program.

The simplicity of control commands in language X leads to simple equivalent Carabiner control operations and the translation is easy.

Before giving the algorithm, first informally and then in Carabiner, we make some remarks and give some illustrations. These are not exhaustive and may not be applicable to other source languages.

(i) There are no conditional control transfers in X so

$$\mathrm{CONT}(\alpha, \beta, \gamma) \Rightarrow \alpha = \underline{\mathrm{True}} \in \underline{k}\ \underline{\mathrm{Bool}}$$
$$\gamma = \Lambda$$

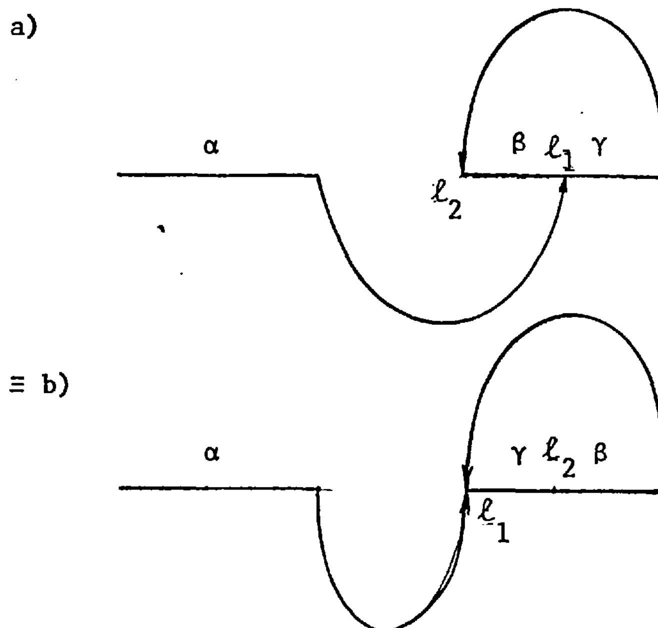
hence:

(ii) <u>goto</u> L1 $\Rightarrow$ CONT(True, $\downarrow$, Λ)
L1 $\in$ ATTRIB

(iii) L1: statement $\Rightarrow$ L1 $\in$ ATTRIB
$\downarrow$
statement' $\in$ STORE

- where statement' is the translation of statement.

(iv) Since control flow changes can only be dictated by 'obeying' labels, we must insert additional CONTrol indicators prior to each label. To illustrate the need for this, consider the following flow path.



The flow from β to ℓ_1 in (a) needs to be catered for in the same way as in (b); the difficulty being that, although in (a) the flow from β to γ seems to be *forward*, it does in fact imply a jump *back* to a previously translated segment - namely γ .

A solution to this problem is to replace any label, ℓ_n, by:-

$$\ldots,\ \mathrm{CONT(True},\ \ ,\ \Lambda),\ \omega,\ \ldots$$

(with ℓ_n pointing to ω, and an arrow from the empty slot in CONT to ω)

- then *any* manipulation *through* a label must involve explicit manipulation of that label.

(v) Given the restriction on control transfers across block boundaries (as in X), then temporary markers, $\overset{\uparrow}{\phi}$, may be used to delimit sublists of STORE and ATTRIB such that if 'goto'<label> is the original text then

(a) <label> $\in$ the required sublist of ATTRIB and is !(unique) in that sublist, and

(b) *k*<label> $\in$ the corresponding sublist of STORE and is ! in that sublist.

N.B. Of course $\underline{k}^{-1}$<label> need not be unique - or even exist.

From the previous observations we may formulate the following translation procedure:

(i) On recognising the *beginning* of a block, set temporary marks, $\overset{\uparrow}{\phi}$, in ATTRIB and STORE, and add *label* to ATTRIB.

(ii) While in the block process *goto* statements thus:-

given *goto* <label>, check that <label> $\notin$ *k*(*label* OF ATTRIB)

if so then create label i.e. add <label> to ATTRIB

and set $\underline{k}^{-1}$ <label> to *label*

replace goto <label> by CONT(True, $\S_2$, Λ)

set k($\S_2$ OF CONT) = <label> OF ATTRIB

(iii) While in the block process <label>s thus

given <label>, check that <label> $\in$ k(label OF ATTRIB)

if it is then stop (fail),

otherwise add <label> to ATTRIB

and set k^{-1}<label> = label OF ATTRIB

add CONTrol to STORE,

set $\S_3$ OF CONT to Λ,

$\S_1$ OF CONT to True

link Boolean OF ATTRIB to $\S_1$ OF CONT

add ω† to STORE

set k<label> = ω

link $\S_2$ OF CONT to ω

(vi) On recognising the end of a block; insert Block( ) to left of $\overset{\uparrow}{\phi}$ in STORE, add 'next' to ATTRIB and process the section e COPYR(STORE, $\overset{\uparrow}{\phi}$) from left to right as follows.

(a) if the section is null then goto (e)
else goto (b)

(b) set knext to first element of section and goto (e)

(c) if knext is a CONTrol statement
then goto (d)
else add statement to BLOCK
if section from knext is null
then goto (e)
else set knext to following statement
goto (c)

---

† ω may here be regarded as the unity operation i.e.

Sx, x, x for any x .

(d) (CONTrol Statement)

if $\underline{k}^2$(§$_2$ OF $\underline{k}$(next)) ∈ BLOCK

then we have a loop so generate PTL

goto (f)

else *if* $\underline{k}^2$(§$_2$ OF $\underline{k}$next) ∈ section

then set $\underline{k}$next = that element of section

else error. stop.

goto (c)

(e) Add EXIT(1) to BLOCK

goto (f)

(f) Trim from $\updownarrow$

Trim from label OF ATTRIB

From the above we may define the following Carabiner translation sequences for X:

⟩ BLOCK ENTRY ⟨ ≡ ⟩ AT(next)
LOC($\updownarrow$)
AT(label)
LOC(LOC($\updownarrow$)) ⟨

⟩ GENERATE 'GOTO' ⟨ ≡ ⟩ *if* ($< >_{9.2}$ $\not\subset$[††] $\underline{ek}$(label OF ATTRIB))
then (AT($< >_{9.2}$),
LINK(label OF ATTRIB,$< >_{9.2}$ OF ATTRIB))
else Λ,
AUGR(BLOCK, CONT (*True*,§$_2$, Λ)),
LINK(*Bool* OF ATTRIB, §$_1$ OF (CONT OF BLOCK)),
LINK(§$_2$ OF (CONT OF BLOCK), $< >_{9.2}$OF ATTRIB) ⟨

⟩ LOG LABEL ⟨ ≡ ⟩ *if* L $< >_{12.1}< >_{12.2}$ ∈ $\underline{ek}$ (label OF ATTRIB)
then $\underline{e}$EXIT (T)[†]

[†] Error; label occurs twice in same block.

[††] Set theoretic notation is used to aid comprehension; however, this does not go against the philosophy of using S to specify operations; since in all finite (practical) cases we may express set operations as S operations on lists - see Chapter 6.

```
                else Λ,
              AT(L < >12.1 < >12.2),
              LINK(label OF ATTRIB, L < >12.1 < >12.2 OF ATTRIB),
              LOC(CONT (True, §2, Λ)),
              LINK(Bool OF ATTRIB, §1 OF (CONT OF BLOCK)),
              LINK(§2 OF (CONT OF BLOCK), L < >12.1 < >12.2 OF ATTRIB),
              LOC(ω),
              LINK(L < >12.1 < >12.2 OF ATTRIB, ω) ⊰

⊱BLOCK EXIT⊰ ≡ ⊱INSERTL(STORE,ϕ↕ , BLOCK( )),
          AT(next),
          if(COPYR(STORE,ϕ↕ ) = ∅)
            then(AUGR(§1 OF BLOCK, EXIT(1)))
            else(LINK(next OF ATTRIB, §1 OF COPYR(STORE,ϕ↕))),
                BLOCK(PTL(True,(if(ek(next OF ATTRIB) = CONT( ))
                                    then(if(ekek(§2 OF ek(next
                                                   OF ATTRIB)) ∈ BLOCK)
                                          then(INSERTL(BLOCK,
                             k²(§2 OF k(next OF ATTRIB)), PTL(True, §2)),
                                              LINK(Bool OF ATTRIB,
                                               §1 OF(PTL OF BLOCK)),
                                              S1(§2 OF(PTL OF BLOCK),
                         eCOPYL(BLOCK, k²(§2 OF k(next OF ATTRIB)))),
                                              DELR(BLOCK,
                                       k²(§2 OF k(next OF ATTRIB))),
                                              EXIT(1) )
                                          else(if(ekek(§2 OF ek(next
                                    OF ATTRIB)) ∈ eCOPYR(STORE,ϕ↕ ))
                                                then LINK(next OF
                               ATTRIB, k²(§2 OF ek(next OF ATTRIB)))
                                                else EXIT(T))
```

```
                    else(AUGR(BLOCK, ek(next OF ATTRIB)),
                        if(eCOPYR(STORE,
                                k(next OF ATTRIB)) = Ø )
                           then(AUGR(BLOCK, EXIT(1)),
                                EXIT(1)
                           else(ADJ(next OF ATTRIB,
                      knext OF ATTRIB, next OF ATTRIB,
                      eNext(STORE, knext OF ATTRIB)),
                                S1(Prev(STORE, next OF
                                             ATTRIB, Λ)))

TRIMR(STORE,ϕ̂ ),
TRIMR(ATTRIB, label OF ATTRIB)) ⤙
```

7.3 A translated X program.

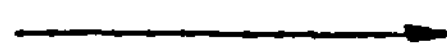
We give an X program, its Carabiner equivalent and descriptions of the CPS state at various points during the execution of the object program.

The source program is:-

```
begin let A, B be real;
    begin let B, C be integer;
          C := 1;
          begin let D, B be integer;
                A := 3.1;
                B := 2;
          L1    D := C + 1;
                A := B + D;
                goto L2;
                A := B;
          L2    C := B + C;
                goto L1
       end
     end
end
```

With the knowledge that the above is a valid X program we may parse and translate it to give

τOF (α_4 OF a) IS (LOC.(ϕ),
LOC (A),
(S_1<u>k</u>A, ω),
LOC (B),
(S_1<u>k</u>B, ω),
LINK(<u>real</u> OF ATTRIB, (<u>k</u>A, <u>k</u>B)),

(A) 

```
LOC(φ↑),
LOC(B),
(S1kB, ω ),
LOC(C),
(S1kC, ω ),
LINK(int OF ATTRIB, {kB,kC} ),
(S1kC, 1 ),
LOC(φ↑),
LOC(D),
(S1kD, ω ),
LOC(B),
(S1kB, ω),
LINK (int OF ATTRIB, {kD,kB})
(S1kA, 3.1),
(S1kB,2),
(S1PTL, ( {/}, TRUE ),
.          /
.          /
.         (S1kD,(S1paI+,(ekC, e1))),
.         (S1kA,(S1paIR,(S1paI+,(ekB, ekD)))),
.          .
.          .
```

passes 1 and 2 (B)

```
.          .
.          .
.          .
.          .
.         (S1kC,(S1paI+,(ekB, ekC )))
.
TRIMR(STORE,φ↑ ),
TRIMR(STORE,φ↑ ),
TRIMR(STORE,φ↑ )
            )
```

During translation the program a (i.e. $\underline{k}pa$) is set equal to the preceding list of Carabiner statements, i.e.

S $\underline{k}pa$, ω, a .

Execution of the object program is then caused by the statement

$\underline{e}a$.

a is represented by a list and the sequential execution of the constituent statements in the list is caused by $\underline{e}$ which distributes over a since:-

$\underline{proc} \in (\underline{k}^{-1}(a) \cap \text{ATTRIB.})$

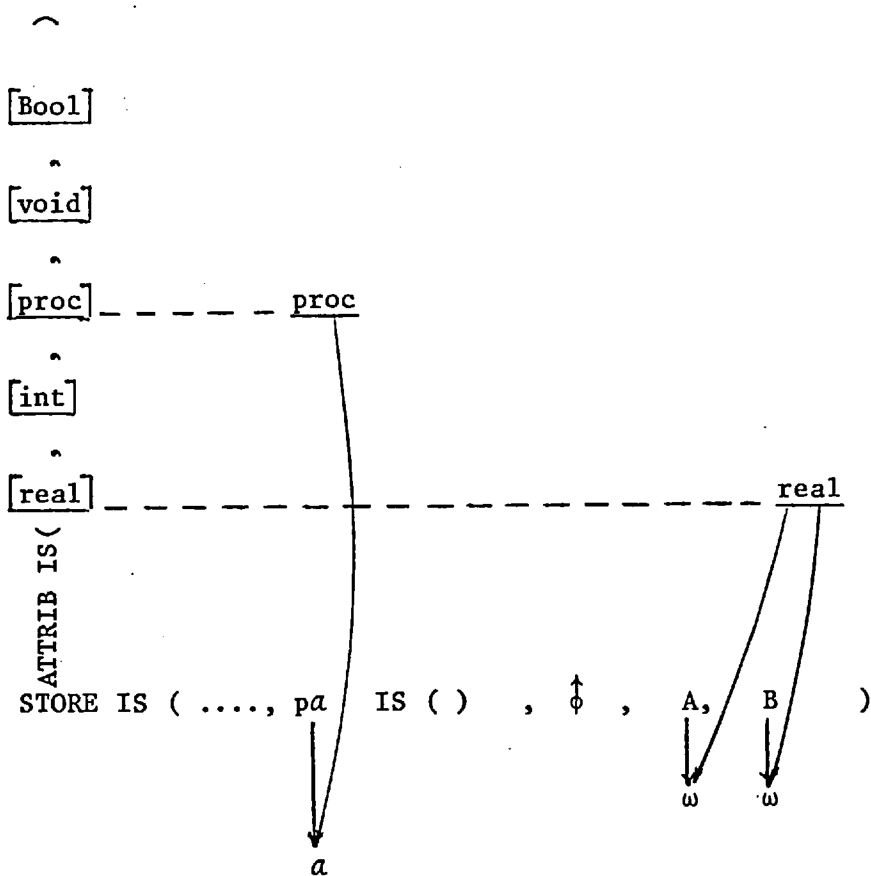
Because of the obvious diagramatic complexities we henceforth denote a by its name and not by the afore-mentioned list.

Three stages in the execution of the program are depicted; these correspond to the positions A and B (at 1st and 2nd pass) in the Carabiner object list. For the present we have ignored the mechanics of calling procedures (i.e. of copying parameters and returning values) and hence in the 'snapshots'† given below we only give details of the list STORE to the right of pa .

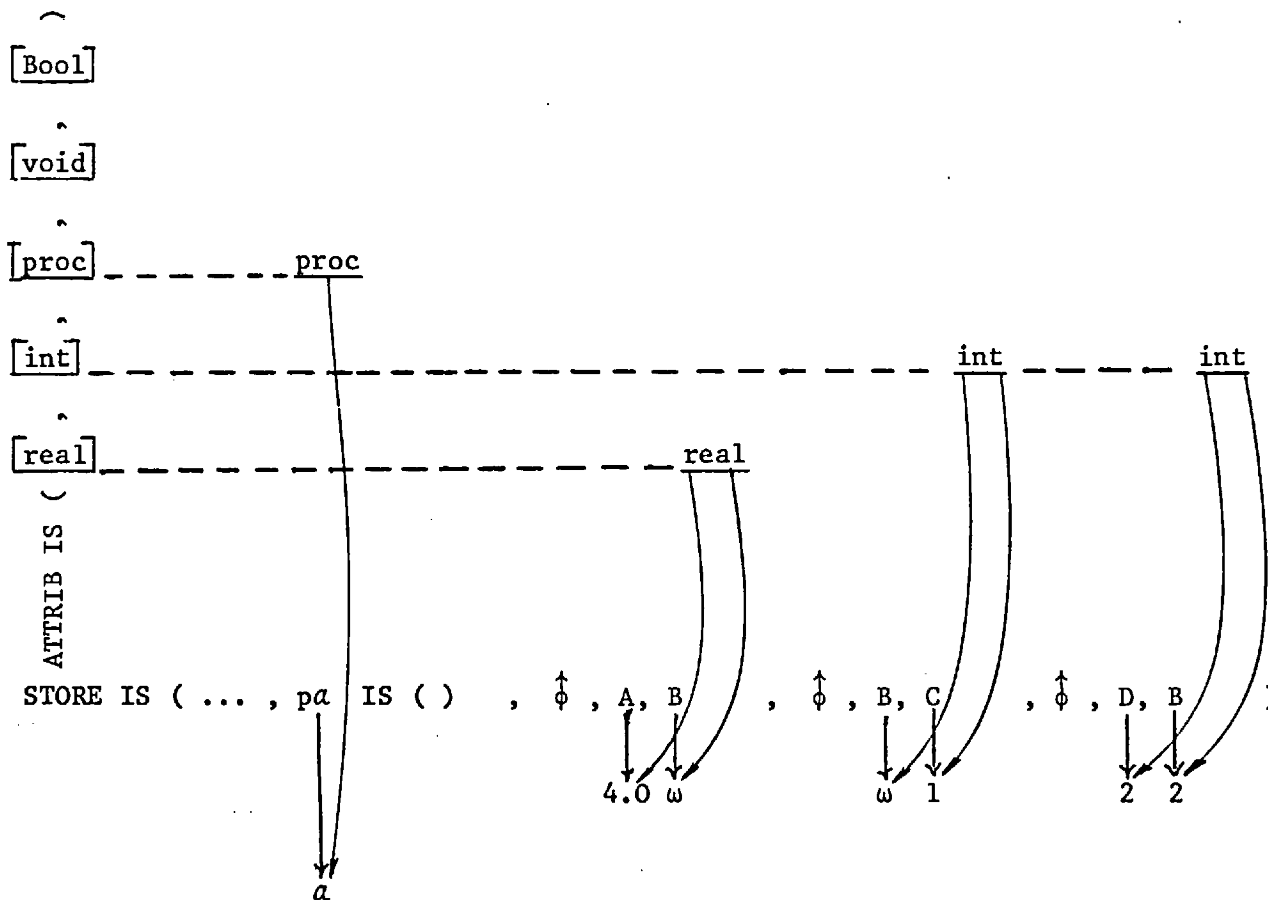
---

† In concept these are not unlike the snapshots of Naur [83].

CPS at Stage A



CPS at Stage B on the first Pass



At stage B on the second pass the structure is as on the first pass but three node values have changed; namely,

$$\underline{k}A = 6 \cdot 0,$$

$$\underline{k}C = 3,$$

$$\underline{k}D = 4$$

CHAPTER 8

PROPERTIES OF THE PROGRAM SPACE (CPS)

8.1 On Derivatives, Neighbourhoods, Relatives and STRUCTs

From §6.8 we recall the following operations on CPS:-

(1) $\text{deriv}(x) = \underline{k}^{-1}(x) \cup \{z : z \in \underline{k}(x) : x \notin \text{ATTRIB}\}$

$\cup \{z : z \in \underline{k}(y_i) \cup \underline{k}^{-1}(y_i),\ (1 \leq i \leq m) \text{ if } x \text{ IS } (y_1, \ldots, y_m) : x \notin \text{ATTRIB}\}$

(2) given $x \in \text{STORE}$

$D_o(x) = \text{deriv}(x)$

and $D_n(x) = \{z : z \in \text{deriv}(y) : y \in D^*_{n-1}(x)\}$

†where

$D^*_n(x) = D_n(x) \setminus \text{STORE}^* \text{ for all } n \in \mathbb{N}$

(3) $\text{Rel}(x) = \bigcup_{n=o}^{\infty} D^*_n(x) \cup \{x\}$

and

$\text{Rel}^*(x) = \text{Rel}(x) \cup \{ (y,z) : y,z \in \text{Rel}(x) \text{ and } z = \underline{k}y \}$

(4) $\text{Nhd}(x) = \text{Rel}(x) \setminus \bigcup_{y \in \text{STORE} \setminus \{x\}} \text{Rel}(y)$

(5) $\text{STRUCT}(x) \doteq \text{Nhd}(x) \cup \{ (a,b) : (a,b) \in \text{Rel}^*(x) \text{ and } (a \in \text{Nhd}(x) \text{ or } b \in \text{Nhd}(x)) \}$

---

†$\text{STORE}^* = \{y : y \in y_1 \in \ldots \in y_n \in \text{STORE}\}$ for all $n \in \mathbb{N}$

These may be interpreted as follows:

(1) deriv(x) is the set of all points one ($\underline{k}$) step away from x; the derivative of x.

(2) $D_n(x)$ is the nth derivative of x.

(3) Rel(x) is the set of all points related to x via $\underline{k}$, $\underline{k}^{-1}$ and OF. Rel*(x) is Rel(x) together with inter-connecting links.

(4) Nhd(x) is the 'neighbourhood' of all points related to only x.

(5) STRUCT(x) is the neighbourhood of x, together with all internal and external links.

STRUCT forms the basis of the topological discussion which follows.

8.2 A dynamic Topology for CPS

The operator STRUCT allows us to define a dynamic topology[†] upon STORE.

Define a topology $\mathcal{T}_o$ over CPS via the basis $\mathcal{B}$ where

$$B \in \mathcal{B} \Leftrightarrow B = \text{STRUCT}(x) \text{ for some } x$$

i.e. $T \in \mathcal{T}_o \Leftrightarrow T$ is the (possibly void) union of some elements of $\mathcal{B}$.

Given such a topology, $\mathcal{T}_o$, then for any $x \in$ CPS the smallest neighbourhood N, of x is

(a) if $x \in$ STORE, then N = STRUCT(x)

(b) if $x \in$ STRUCT(y) for some $y \in$ STORE, then N = STRUCT(y)

(c) otherwise $N = \emptyset$.

[†]The basis formed by this operator changes (dynamically) as the space states change.

Trivially $\mathcal{T}_0$ is not even T_0[†] as may be seen from the structure:-

STORE IS (..., x, ...)
$\downarrow$
3

whence

$$3 \in \text{STRUCT}(x)$$

so

$$\nexists U \text{ open: } 3 \notin U \text{ and } x \in U.$$

Now if we restrict our considerations to STORE, or equivalently redefine (b) in $\mathcal{T}_0$ to yield $n = \emptyset$, then we get $\mathcal{T}_1$ in which all sets are <u>clopen</u>[††], since if $T \in \mathcal{T}_1$ then

$$T = \bigcup_{i \in \mathfrak{J}} \text{STRUCT}(i) \quad \text{with } \mathfrak{J} \subset \text{STORE}$$

so T is open

but $$\bar{T} = \bigcup_{i \in \text{STORE} \setminus \mathfrak{J}} \text{STRUCT}(i)$$

so $\bar{T}$ is open and hence T is closed.

Trivially, if all sets are open then $\mathcal{T}_1$ is T_2 since

$$x \neq y, \; x, \; y \in \text{STORE}$$

$$\text{STRUCT}(x) \cap \text{STRUCT}(y) = \emptyset$$

8.3 <u>On Stability of Programs</u>

Whilst we have no more than a vague idea of the mappings (as generated by complete programs) which act on the set, Γ, of all configurations, γ, of CPS we can still make some general comments about the topology involved and (via separation considerations) limits of sequences of such mappings.

[†] See §8.3

[††] i.e. closed and open.

For a given program (homomorphic via translation to f) and a given initial state γ_j we have:

$$f : \Gamma \to \Gamma$$

and

$$f : \gamma_j \mapsto \gamma_1 \quad \text{(where f is assumed not to 'hang-up' on } \gamma_j\text{)}.$$

Now, if f terminates 'normally' (i.e. STORE is TRIMRed to p^{α} on termination)[†] then γ_1 is very similar to γ_j; the only essential difference being in the contents of the Input/Output vectors (buffers). Defining a suitable metric (and hence a topology) on these buffers allows us to discuss the stability of a program about a given input vector:

if: $f : \gamma_j \to \gamma_1$

then: given any $\varepsilon > 0$, if we can find $\delta > 0$

such that: $f(N(\gamma_j, \delta)) \subset N(\gamma_1, \varepsilon)$

- using analytical notation[††] - then we say f is <u>stable</u> about γ_j.

If $\exists \gamma_j : \nexists \gamma_1$ and $f : \gamma_j \mapsto \gamma_1$

then f diverges (hangs-up) at γ_j.

[†] It may be justifiably argued that since modification of the contents of the Input buffers is the only way of effecting the outcome of the program, and that examining the contents of the output buffers is the only way of establishing the outcome of the action caused by the program; it makes sense to consider the following even when the program terminates in another fashion (and also maybe when it does not terminate).

Note that introducing a trace creates a new output buffer which must then be treated in the same manner as other buffers.

[††] i.e. $N(x, y) = \{z : d(x, z) < y\}$
for the metric d.

If $\exists \gamma_j : f : \gamma_j \mapsto \gamma_1$ and $\exists \varepsilon > 0$ for which there is no $\delta > 0$:

$$f(N(\gamma_j, \delta)) \subset N(\gamma_1, \varepsilon)$$

then γ_j is a <u>critical point</u> of f; equivalently f is <u>unstable</u> about γ_j.

Given that f is divergent at γ_j and is defined by a list:

$$f \text{ IS } (f_1, \ldots, f_n)$$

i.e.

$$f = f_n \circ f_{n-1} \circ \ldots \circ f_1$$

- then it may be fruitful to examine the (internal) effect of the individual f_i's; in doing this we could consider the topology of a general state of the CPS. [Here we consider changes in buffer values to be <u>external effects</u>; changes in the STORE being termed <u>internal effects</u>.] This was done in §8.2.

Given any convergent $f_i : \gamma_{j_i} \mapsto \gamma_{j_{i+1}}$ then topologies on γ_{j_i} and $\gamma_{j_{i+1}}$ induce the product topology on $\gamma_{j_i} \times \gamma_{j_{i+1}}$ and hence (via subset topology) on f_i. A full definition of this topology and discussion of its separation is given below, but first we recall the separation axioms:-

<u>T_0</u> Given a space $X : x, y \in X$ and $x \neq y$.

If $\exists U \subset X : U$ open, $x \in U$, $y \notin U$

then X is T_0.

<u>T_1</u> Given a space $X : x, y \in X$ and $x \neq y$.

If $\exists U, V \subset X : U, V$ open

$x \in U, y \notin U$
$x \notin V, y \in V$

Then X is T_1.

equivalently: given $x \in X$ if $\{x\}$ is closed then X is T_1.

$\underline{T_2}$ (Hausdorf) Given a space $X : x, y \in X$ and $x \neq y$.

If $\exists\, U, V \subset X : U, V$ open, $x \in U$, $y \in V$ and $U \cap V = \emptyset$

then X is T_2.

Regular: Given a space $X : x \in X$ and $A \subset X$, A closed, $x \notin A$.

If $\exists\, U, V \subset X$: U, V open: $x \in U$, $A \subset V$ and $U \cap V = \emptyset$

then X is Regular.

$\underline{T_3}$ X is T_3 if it is T_1 and Regular.

Normal: Given a space $X : A, B \subset X$, X, B closed.

$A \cap B = \emptyset$; if $\exists\, U, V \subset X$: U, V open

$A \subset U$, $B \subset V$ and $U \cap V = \emptyset$

then X is Normal.

$\underline{T_4}$ X is T_4 if it is T_1 and Normal.

With reference to the metric on the I/O buffers, we may formulate the notion of functional approximation:

given $f : \Gamma \rightarrow \Gamma$

and any $g : \Gamma \rightarrow \Gamma$

then

(i) g approximates f with accuracy ε if for any $\gamma : f(\gamma)$ converges

then $g(\gamma) \in N(f(\gamma), \varepsilon)$ $(\varepsilon > 0)$

Trivially, if g approximates f with accuracy ε then f approximates g with the same degree of accuracy.

(ii) g partially approximates f with accuracy ε if for any $\gamma : f(\gamma)$ and $g(\gamma)$ both converge

then $g(\gamma) \in N(f(\gamma), \varepsilon)$ $(\varepsilon > 0)$

(iii) Given $\Gamma_1 \subset \Gamma$ such that g (partially) approximates f for all $\gamma \in \Gamma_1$ then g (partially) approximates f over Γ_1.

The definition of a suitable metric needs detailed consideration but it would seem sensible that it should be either linear i.e.

$$d(\underline{x}, \underline{y}) = \Sigma \, |\underline{x}_i - \underline{y}_i|$$

and $$N(\underline{x}, \mathcal{E}) = \{y : |\underline{x}_i - \underline{y}_i| < \mathcal{E}\}$$

or weighted to give bias to initial data, i.e.

$$d(\underline{x}, \underline{y}) = \Sigma \, \sigma_i \, |\underline{x}_i - \underline{y}_i|$$

and $$N(\underline{x}, \mathcal{E}) = \{\underline{y} : |\underline{x}_i - \underline{y}_i| < \sigma_i \, \mathcal{E}\}$$

where σ_i is a decreasing positive-valued function of i.

However, regardless of the metric, since metric spaces are T_4 and $T_4 \Rightarrow T_3 \Rightarrow T_2 \Rightarrow T_1 \Rightarrow T_0$, we will be able to use the strongest possible theorems on convergence.

The relationships between critical points, canonical forms [96, 97] and properties of programs [39-41] demand further examination.

8.4 On Modal Substructures and Coercions

Via modal chains, we define the modal substructure of $x \in$ CPS. This is then demonstrated by an example.

Defn: A modal chain of $x \in$ CPS is a digraph $D(\underline{y}, z)$ where

$$\underline{y} = y_1, \ldots, y_n \quad \text{and} \quad z = \{(y_i, y_{i+1}) : (1 \leq i < n)\}$$

such that:- (a) $x = y_1$

(b) $y_i \in$ ATTRIB iff $i = n$

(c) $y_i = y_j$ iff $i = j$

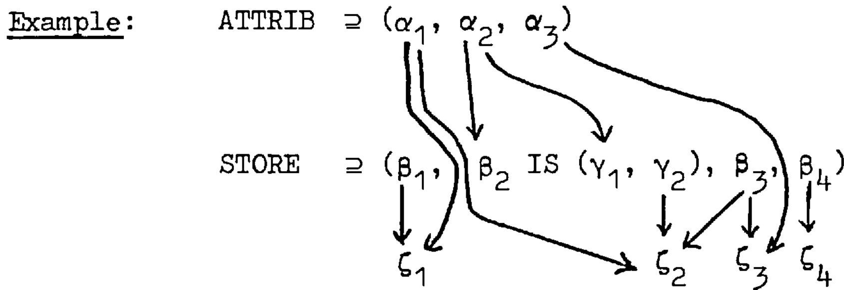
(d) $y_{i+1} \in \underline{k}\, y_i \cup \underline{k}^{-1}\, y_i \quad (1 \leq i < n)$

Defn: The Modal Substructure, MS, of $x \in$ CPS is

$MS(x) = \bigcup_i y_i$ where y_i is a modal chain of x and the natural equivalences induced by the CPS carry over.

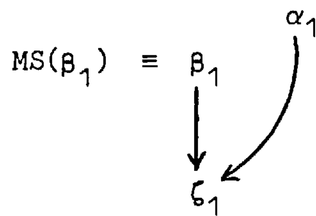
Example: ATTRIB $\supseteq (\alpha_1, \alpha_2, \alpha_3)$

STORE $\supseteq (\beta_1, \beta_2$ IS $(\gamma_1, \gamma_2), \beta_3, \beta_4)$

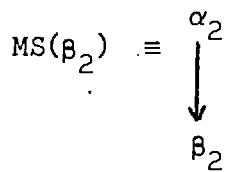


Then: $MS(\alpha_i) \equiv \dot{\alpha}_i \quad (i = 1,2,3)$

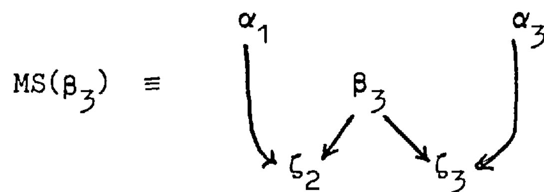
$MS(\beta_1) \equiv$



$MS(\beta_2) \equiv$



$MS(\beta_3) \equiv$



$MS(\beta_4) \equiv \emptyset$

$MS(\gamma_1) \equiv \alpha_2 \rightarrow \gamma_1$

$MS(\gamma_2) \equiv$ $\alpha_1 \rightarrow \zeta_2$, $\gamma_2 \rightarrow \zeta_2$, $\beta_2 \rightarrow \zeta_2$, $\beta_2 \rightarrow \zeta_3$, $\alpha_3 \rightarrow \zeta_3$

$MS(\zeta_1) \equiv \alpha_1 \rightarrow \zeta_1$

$MS(\zeta_2) \equiv MS(\zeta_3) \equiv$ $\alpha_1 \rightarrow \zeta_2$, $\beta_2 \rightarrow \zeta_2$, $\beta_2 \rightarrow \zeta_3$, $\alpha_3 \rightarrow \zeta_3$

$MS(\zeta_4) \equiv \emptyset$

Now, S_1 <u>ky</u>, <u>ek</u>x is only (directly) executable if

$$MS(\underline{k}y) \equiv MS(\underline{k}x)$$

Coercions of the dereferencing kind may be modelled by allowing the removal of nodes from within an MS in such a way that the resulting digraph is still a valid MS for the original node considered. This is expounded further in §9.4

Coercions which explicitly change node values are, of course, language dependent and must be dealt with individually (see §9.5).

CHAPTER 9

ON DESCRIBING OTHER PROGRAMMING LANGUAGE FEATURES

Here we give a brief discussion of how the language and the space of the Carabiner system may be used to specify characteristics of high-level programming languages that do not appear within our example language, X. The list of features examined is by no means exhaustive but is quite extensive, having been extracted from Ledgards Mini-Languages [68]. A more detailed treatment of a set of PLECS (Programming Languages to Exhibit Carabiner), based on the mini-languages is given in two technical reports [33, 34]. The contents of this chapter constitute a précis of these reports.

9.1 Assignment and I/O

By direct application of the three fundamental operators, S, k and e, we can model any kind of generalized assignment by a statement of the form:-

$$S_1\ \alpha,\ \beta$$

Here, α is a left-hand value and β a right-hand value [6, 15]. Details of this construction have already been given in chapter 6 and will not be repeated here.

Now consider the program element of CPS:-

ATTRIB IS (..., proc, ...)

STORE IS (..., OUTPUT FROM αIS(§), pα IS (§)

Restricting our examination to the modelling of a high-level source language and ignoring matters relating to job organization, I/O spooling etc., then trivially the action of the program (i.e. $\underline{e}\,\underline{e}\,\underline{k}\,p\alpha = \underline{e}\alpha$), in general, will cause the items in the list α to be processed by $\underline{e}$ and information resulting from this computation will be placed in the list OUTPUT FROM α. The actual form of these lists depends on conventions for I/O control (i.e. line feed, space, page throw etc.). An immediate consequence of our representation of procedures, and hence of any complete program, is that I/O is essentially reduced to assignments to and from buffers. Input may be specified thus:-

input to X : $S_1\,\underline{k}\,X$, $\underline{e}$ COPY ($§_i$ of α)

or $S_1\,\underline{k}\,X$, $\underline{e}$ TRIML($§_i$ OF α, ∇)

- where ∇ denotes space (say).

output from X : AUGR($§_j$ OF OUTPUT FROM α, $\underline{e}\,\underline{k}\,X$)

Trivially these I/O commands may be embedded within conditional constructs.

9.2 Transfer of Control and Block Structure

As is shown in the appendix, any program may be re-structured[†] so that the only control constructs required are:

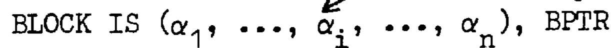
RAM (a, b, c) i.e. if...then...else

PTL (a, b) i.e. do...while...

BLOCK (a)

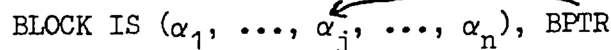
EXIT (a)

If, however, we wish to leave the program structure as dictated by the source language, we can use the procedure template and adjust its 'next instruction' pointer kBPTR (see chapter 6) to change:



BLOCK IS (α_1, ..., α_i, ..., α_n), BPTR

to



BLOCK IS (α_1, ..., α_j, ..., α_n), BPTR

The related problem of scope sensitive data (e.g. locally (re-) declared identifiers in Algol-60) is easily solved by using a marker, Φ say, which is placed before any local quantities and can later be used to TRIM off these quantities.

e.g. BEGIN $\Rightarrow$ LOC(Φ)

END $\Rightarrow$ TRIMR (STORE, Φ)

This is exactly the same method as used in language X as described in chapters 3 and 7.

[†] This is always possible but may not be desirable;
e.g.

goto (read)
 - where read, inputs an integer (label)

9.3 Functions and Parameters

Recall that a procedure (function) template is of the form:

ATTRIB IS (..., proc, ...)

STORE IS (..., OUTPUT FROM f IS ($\underline{\S}$), f IS ($\underline{\S}$), ...)

ω

- where $\underline{k}^{-1}(\S_i$ OF OUTPUT FROM f$) \subseteq$ ATTRIB
  and
  $\underline{k}^{-1}\ \underline{k}\ (\S_i$ OF f$) \subseteq$ ATTRIB

In defining a procedure we must define all the necessary ($\underline{k}$) links explicitly; then, when the procedure is 'called' we may use the topological operator STRUCT (see chapters 6 and 8) to copy the whole structure, related as in the definition, into local work space, by e.g.

LOC (STRUCT(f)) (†)

The process of loading, executing and then deleting a function is elaborated in §6.4 and yields a Carabiner sequence of the form:-

S Λ, Λ, STRUCT(f)

S $\underline{\S}$ OF f, $\underline{\S}$ OF f, $\underline{x}$

$\underline{e}\,\underline{k}$ f

S STRUCT (f), STRUCT (f), Λ

Loosely this represents: LOAD f

LOAD parameters ($\underline{x}$)

EXEC f

DELETE f

---

† Properly, we need two LOC's, the other being of the form LOC(STRUCT(OUTPUT FROM f)), however here this serves only to cloud the main issues.

Implicitly associated with each template is a pointer PTR which well defines the order in which parameters are loaded and also the order of evaluation (e̲) of the sequence, k̲f̲, of statements which constitute the body of f.

The only problem that occurs in relation to parameters is that of emulating differing modes of calling the parameters - or equivalently (re)initializing identifiers within the procedure body. (A fuller discussion of such modes is given in CPL and related documentation [6, 33, 68, 86]).

As an extreme example consider the 3-ary function f, with the three parameters called by value, reference, and name (expression) respectively.

```
e.g.        f(a, b, c), value a,
                        ref   b,
                              c
```

Now let modes of execution (modex) be as follows:

	compilation	⇒ 1
(function)	definition	⇒ 2
(program)	execution	⇒ 3
(function)	execution	⇒ 4

The stages involved in processing a program of the form:

```
define f
   :
call f
   :
```

are thus:- modex ← 1

(compile)

$\underline{e}_1$ LOC ($\underline{e}_2$ ...)

⋮

modex ← 2

(process definitions)

$\underline{e}_2$ LOC(f($\underline{e}_3\underline{k}\S_1$, $\underline{k}\S_2$, $\underline{e}_4\underline{k}\S_3$)

(also $\underline{e}_2$ LOC($\underline{e}_4$ set modex = 4

⋮

$\underline{e}_4$ set modex = 3))

modex ← 3

⋮

<u>call</u> f ($\underline{e}_3\underline{k}\S_1$, $\underline{k}\S_2$, $\underline{e}_4\underline{k}\S_3$)

modex ← 4

⋮

including e.g. $\underline{e}_4 S_1\underline{k}\ \S_2$, $\underline{e}_4\underline{k}\S_3$

(originally b := c)

⋮

modex ← 3

exit from f

⋮

This would cause trouble if f were recursive or occurred in an embedded construct, i.e.

y := f(a, f(b, c, d), e, g)

Other methods utilise a 'saving' function which, like all functions, creates a new copy of itself and hence has to re-evaluate its parameter(s). This re-evaluation is just what is needed to model

call by name (or expression). This approach has been adopted to define PLEC5 [33].

9.4 Type Checking

Using strictly disjoint types (or modes), pre-run type checking is not possible in all languages (cf. [68]). However, if we define suitable mode-hierarchies, we are always able to perform some degree of checking by an extension of traditional dictionary techniques.

Essentially our method is a development of the de-referencing of Algol-68 [108] and the projections used in Algol N [55, 104, 124]. Here is not the place to enter into a detailed description of the method since the manipulations involved must be defined in terms of the elements and constructs of the language under consideration - the types involved in the language X are far too simple to justify the use of such a general technique and the describing of a suitable language would be too time and space-consuming. We enumerate the main features of the system:-

I We need to create a (scope sensitive) dictionary of all variables, function templates (as in §6.4) and attributes; moreover these entries should be linked to all static (i.e. permanent) attributes. This is achieved by execution of all declarative statements within the program; and in languages such as FORTRAN where subprograms may physically follow CALL's to them, this may necessitate two passes of the checker.

II In order to be able to link items to attributes of varying degree we need to put some hierarchical structure on sets of

related attributes. For entities of fixed type there is no problem; we link the entity to that type. However, if, for example, we had a value which was always numeric but could alternate between being real and int(eger), then the best we could do at compile time would be to regard the value as linked to both real and int, and hence (semi-) type check by examining a subgraph.

i.e. (..., real, ..., int, ...)

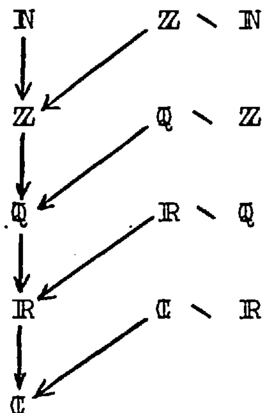
or

(..., real, ..., rint ..., int, ...)

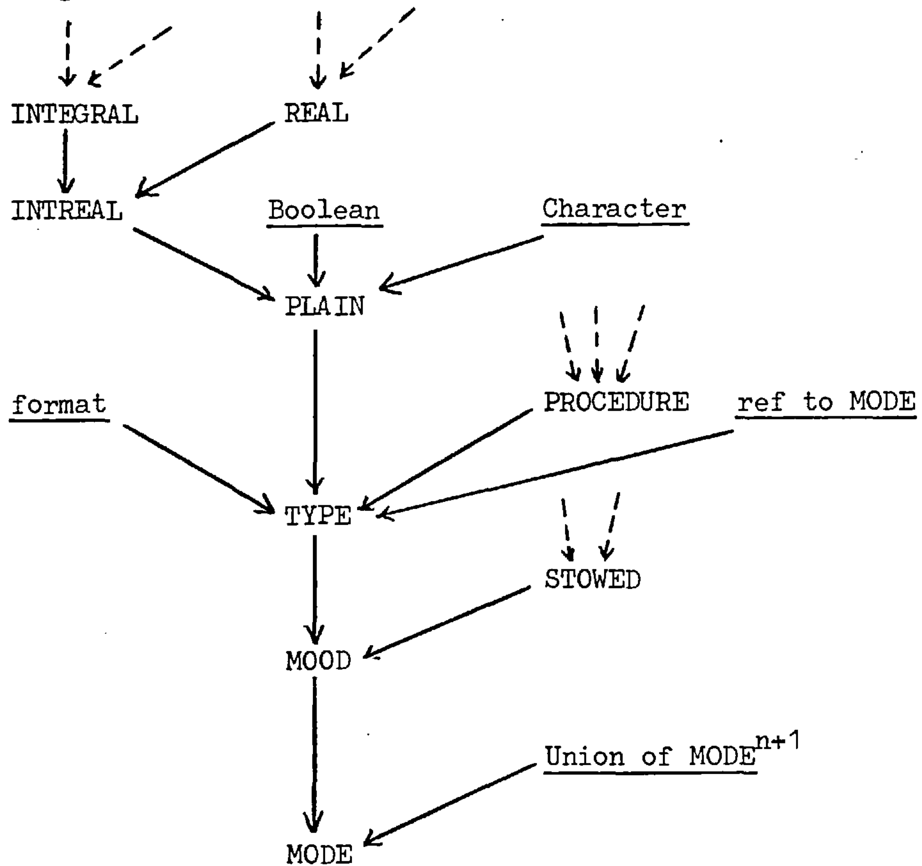
- the relationship between these constructs is given in III below.

Further examples can be drawn from mathematics:-

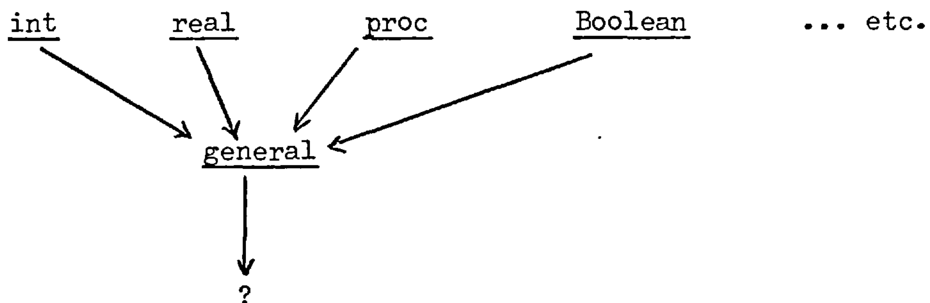
Using the common set denotations:-



- from Algol 68 [108]:-



- from a general 'type-free' language [68]



Note: Given an element, x, of CPS such that $\underline{k}^{-1}x \subset$ ATTRIB and say -

$$\underline{k}^{-1}x = \{p, q\}$$

- then the attributes of x are p and q.

Clearly this is violated in the present construction since we may have elements which (seem to) have properties that are inverses of each other, e.g.

$\mathbb{R}$ and $\mathbb{C} \setminus \mathbb{R}$.

However, since we are only considering declarations and no values are present we can explain this apparent contradiction by saying that the properties of a location are that it can hold a value of (say) type $\mathbb{R}$ or $\mathbb{C} \setminus \mathbb{R}$. These properties are not necessarily disjoint and so no conflict arises in the model; of course at run-time a value can only have non-conflicting attributes although these may be changed. This is exactly what is required when examining a 'Union of' mode in Algol 68 [108] (see §9.5).

III In order to test for the possibility of compatibility of attributes at run-time we define two operations; the first of these is a contraction.

Consider a connected structure, A_0, within CPS (e.g. Rel(x) for some x in STORE), then if $\exists\, y \in A_0$ such that y is not an atom or an anti-atom then we may contract A_0 about y. To do this we remove y and link all elements of $\underline{k}^{-1}y$ to all elements of $\underline{k}y$ (these exist by the assumptions made about y). If the resultant structure is A_1 then we denote the relation between A_0 and A_1 by

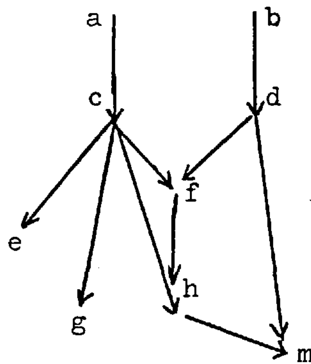
$$A_1 \triangleleft A_0$$

A_1 may then be contracted, and so on. Extending notation and terminology we may define the operation of contraction to include the removal of more than one node, i.e.

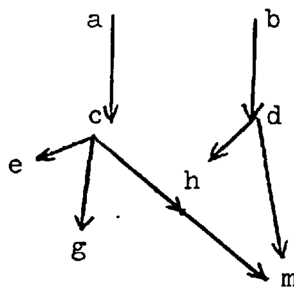
$$A_i \lhd A_j \text{ and } A_j \lhd A_l$$
$$\Rightarrow \quad A_i \lhd A_l .$$

Since any state of CPS is finite and hence so is any sub-structure of a state, then for any A_0 we must reach a contraction which contains only atoms and anti-atoms. Although the contraction chain (i.e. the chain of contractions) may not be unique, the resultant (non-contractible) structure is unique and called the Unique Ultimate Contraction (UUC).

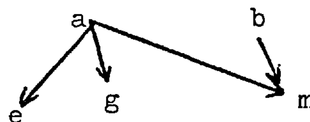
e.g. if A is



A_1 is



and A_2 is



Then $A_2 \lhd A_1 \lhd A$

and UUC (A) is A_2.

Clearly, if $\bar{A}$ is the set of nodes in A and

$$\bar{A} \supset \bar{\bar{A}} \quad = \{a \in \bar{A} : a \text{ is not an atom or an anti-atom}\}$$

then the set of contractions of A is isomorphic to the power set of $\bar{\bar{A}}$ and has the same related (finite) lattice. Using the natural isomorphism between A and $\bar{\bar{A}}$ then

$$A \sim \bar{\bar{A}}$$

and

$$\text{UUC }(A) \sim \emptyset \; .$$

IV If we now allow the possibility of throwing away part of a structure (in order to test for possible compatibility of mode options which interact but are not strictly contained one within another) by removing any subset of nodes (and associated k-links) such that the resultant structure is connected - i.e. reduce a structure to a connected subgraph - and still has all the original atoms and at least one anti-atom (denote this by $\overset{*}{\subset}$), then we define a subcontraction B of A, written B ◀ A,

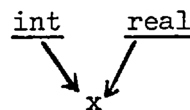
if $\exists$C:

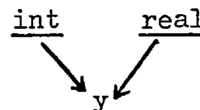
$$B \overset{*}{\subseteq} C \lhd A$$

In some cases compatability may only be required to the extent that A, B are suitably compatible iff $\exists$C:

$$C \blacktriangleleft A \quad \text{and} \quad C \blacktriangleleft B$$

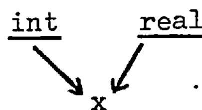
e.g.





In other instances it may be required that $A \lhd B$

e.g.



N.B. The relation ◁ is elsewhere termed 'modal substructuring'[†]. This extended dictionary technique demands that all procedures be defined with templates that have explicit output vectors and that the mode void be implicitly modelled.

In general, manipulation of the modal substructures only allows us to test for possible compatibility between constructs; any actual run time structures must match exactly even if this requires coercion to the structures in order to achieve a match. Of course, allowable coercions must be specified by the semantics of the language involved.

9.5 Structured Data

To illustrate the way in which (user defined) data structures can be represented we examine two commonly occurring constructs, namely a tree (of integers) and a list (again, of integers).

Before giving these constructions, we note that the interpretation of (Carabiner) lists within ATTRIB need not necessarily be the same as that of lists in STORE. Here we use such a list to represent unions of attributes, i.e.

$$x \text{ IS}(x_1,\ldots,x_n) \subseteq \text{ATTRIB}$$
$$\downarrow$$
$$y$$

$$\Rightarrow \; y \text{ is of type } \bigcup_i x_i$$

[†] Formally defined in §8.4.

If, however, elements of such a list are (<u>k</u>)-linked to other ATTRIButes, then the elements explicitly name components of the composite objects of type x. e.g. given the situation above, then y has n components named x_1, ..., x_n. The following examples help clarify.

<u>define</u> <u>type</u> LIST IS UNIT OR PAIR,
UNIT IS (ATOM IS INT)
PAIR IS (HEAD IS INT, TAIL IS LIST).

In the CPS this is:-

ATTRIB IS(.., <u>INT</u>, LIST IS (UNIT IS (ATOM),PAIR IS (HEAD, TAIL)),...

As a direct result of the construction it is obvious that no object manipulated by a program is a LIST - it may be either a UNIT or a PAIR; moreover where a type, such as LIST, is a union of other types, there must be provision for deciding which (sub)type is applicable. In most cases these types will have different structure and this can be used.

e.g. <u>let</u> A,B <u>be</u> LIST

A := (37)

B := (17,(6))

⇒ A is a UNIT

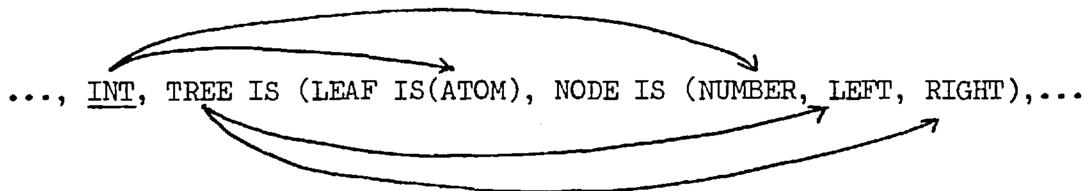
B is a PAIR

and TAIL OF B is a UNIT

Similarly,

```
define type  TREE IS LEAF OR NODE
             LEAF IS (ATOM IS INT)
             NODE IS (NUMBER IS INT,
                      LEFT IS TREE,
                      RIGHT IS TREE)
```

is modelled thus:-



then

```
    let A be TREE
    A := (1, (2,(3),(4)), (5, (6,(7),(8)), (9)))

⇒   RIGHT OF A is a NODE
RIGHT OF (RIGHT OF A) is a LEAF, value 9
RIGHT OF (LEFT OF A) is a LEAF, value 4
etc.
```

9.6 String Manipulation

Because Carabiner uses the (character) representations of the high-level source languages, and all the basic operations are defined (by Markov algorithms) in terms of their effects upon the string representations of the given parameters; string manipulation is implicitly inherent within the system.

Moreover, since string manipulation transformations are easily translated into MAs, any such transformation can be identified with

an equivalent base function. Extension of the set of base functions may seem a very naive way of dealing with string manipulations, however since (a) the system is designed primarily as a means of defining high-level languages and (b) any non-trivial MA can be decomposed into explicit loops, matching predicates and, substitutions, we feel justified in adopting this approach.

CHAPTER 10

CLOSING REMARKS

Over the last ten years, particularly in the latter half of that period, there have been numerous attempts to devise practical means by which semantics could be defined. Below, we note the work of prominent researchers and groups of researchers in this field. This is followed by a summary of comparisons and contrasts between these systems and Carabiner. We conclude with a resumé of work directly related to Carabiner and a discussion of how it may be developed further.

Probably the best known experiment in defining high-level languages is the Vienna Definition Language, (VDL) [54, 66, 69, 70, 112] developed by IBM's Vienna Laboratory to specify PL/I and later used to provide a formal definition of Algol-60. This system incorporates a very general abstract syntax, developed from BNF, and uses tree structures to specify data, programs and the environments in which programs are executed. The Common Base Language [36] designed by the Computational Structures Group of Project MAC at MIT under the direction of Jack Dennis utilises VDL as the basis for an UNCOL [79, 98, 99].

Several methodologies have used Church's λ-calculus [18] as a starting point. Of these the work of Landin [61-63], Böhm [13, 14] and early work by Strachey [101] demand mention. The formal equivalence of the substitution properties of λ and S was also investigated by Nixon and Wesselkamper in the initial stages of the Crampon project [86].

Galler and Perlis [48] and deBakker [5] base their systems on Markov Algorithms [74] and hence are in some sense definitive; moreover deBakker uses a meta-language (after van Wijngaarden [106, 107]) and a similar notation, '⌈ ... ⌉', to that of Carabiner.

The most recent, and probably the most mathematically formidable, system has been evolved by the Programming Research Group at Oxford (Strachey, Scott etc. [81, 90-92, 102, 103]). This system depends on a considerable amount of mathematical idealisation.

Other, less easily classifiable, work includes McCarthy's use of state vectors [76] to define semantic changes; Feldman's semantic meta-language FSL [44]; and Wirth's use of semantic phrases to define the language EULER [117].

In offering Carabiner as an alternative to the above mentioned systems, we put forward the following considerations:-

that although Carabiner demands a specific syntax† (VDL does not), an explicit order of evaluation (VDL allows some choice) and full definition of all functions and operations, this is no loss since syntax rarely, if ever, needs the generality given by VDL and any choice in orders of evaluation are more properly admitted to the model explicitly (via axioms) instead of by default;

that in permitting the interlinkage of any components in CPS, Carabiner is essentially more flexible than the M.I.T. System;

---

† The current restriction on the extensibility of syntax is not intrinsic. It is included only to simplify the translator which may later be replaced by a more general recogniser (cf the work of Vettes [109]).

that following the discussion of §6.3 any λ-calculus system is incapable of giving information about the internal dynamics of a calculation; moreover we need to reflect the distinction between identifier/value and place-holder (as in mathematics) - see §6.2 ;

that while the Markov-based systems, and McCarthy's, can be definitive about string-to-string transformations the model of the program space used in Carabiner allows more meaningful interpretations of intermediate states of a computation. Also the use of S to effect changes in the model makes the overall system more uniform;

that despite its mathematical inelegance Carabiner is no less precise than the Scott-Strachey semantics and is easier to comprehend (c.f [81]);

that Carabiner's (semantically ideal) intermediate language - without implementation specifications - facilitates easy cross-translation between computing systems using common character sets. This is in contrast to FSL in which it is believed the semantics and pragmatics are intermixed to such an extent as to make such translations almost impossible, and;

that Wirth's system lacks the well-defined pragmatic level which is present in Carabiner.

Summarising: in order to use Carabiner to define a high-level language programming system, one must first know the representation to be used and have a suitable grammar which is based on the given representation. Into this grammar we inject semantic phrases which

act on our abstract program space. At this stage, elementary operations of the language are given in terms of the source level representation and are specified by Extended Markov Algorithms. This gives a machine (i.e. implementation) independent description of the high-level language in terms of an intermediate language having only a few basic functions and acting on a realistic model that is capable of representing intermediate states of a computation. From this description, the implementor can determine the language designers intended semantics which may then be interpreted as closely as is desired. Replacing the original EMAs by ones representing the operations as implemented then gives a definitive specification of the system implementation.

Also, the system may be 'watered down' to a form suitable for use as a reference manual related to a specific implementation, or provide for the direct execution interpretation of prototype languages [113].

We have attempted no formal verification of the sufficiency of Carabiner but as evidence of the versatility of the system we cite its use to define a set of ten high-level languages. These languages, which we call PLECs (Programming Languages to Exhibit the use of Carabiner) were specially designed to act as test cases for methods of language description and are closely modelled on Ledgard's mini-languages [68]. Documentation of these definitions is given elsewhere [33, 34] and was abstracted in chapter 9.

Carabiner builds closely on the concepts used in the Crampon project [86, 87, 114-116]. Although the current model is less complex than that of Crampon, the basic elements are similar; however, Carabiner is more pragmatic, insisting that all actions within the execution of a program are explicitly defined.

Building on the work presented here and that currently being undertaken by Snidvongs [93-97] on "S-algebras" it would seem timely to investigate the use of S-operations to express optimization formulae, and hence to attempt to develop a uniform theory of optimization to extend and replace the presently used set of ad-hoc tricks and graph-theoretic transformations.

The links between the three above mentioned projects are, at present, notional rather than concrete. The problems of unifying them could be investigated in an attempt to define a structured (layered) translation scheme downwards from high-level source languages. Such implementation investigations could possibly also cast light on finding an alternative representation for function templates such as discussed in chapters 6 and 9.

References

1. Aho, A.V. and Johnson, S.C. "LR Parsing", Computing Surveys 6(2), pp. 99-124 (1974).

2. Allen, F.E. and Cocke, J. "A Catalogue of Optimising Transformations", in Design and Optimization of Compilers, Ed. Randell Rustin, pp. 1-30, Prentice-Hall 1972).

3. Ashcroft, E. and Manna, Z. "The Translation of 'goto' programs into 'while' programs". Computer Science Department Report CS 188, Stanford University (1971).

4. Ashcroft, E. and Manna, Z. "The Translation of 'goto' programs into 'while' programs". Proc. IFIP Congress 71, Ljubljana (1971).

5. de Bakker, J.W. Formal Definition of Programming Languages, Math. Cent. Tracts 16, Mathematisch Centrum, Amsterdam (1967).

6. Barron, D.W. et al. 'The Main Features of CPL', Computer Journal 6(2), pp. 134-143 (1963).

7. Berge, C. Theory of Graphs (English translation), Methuen, London (1962).

8. Berge, C. Graphs and Hypergraphs (English translation), North-Holland Publishing, London (1973).

9. Bochmann, G.V. "Multiple Exits from a Loop without the GOTO", Comm. Assoc. Comp. Mach. 17(7), pp. 443-444 (1973).

10. Bochmann, G.V. Semantics Evaluated from Left to Right, Publication No. 135, Dept. Informatique, Universitē de Montreal, Canada (1973).

11. Bochmann, G.V. Semantic Equivalence of Syntactically Related Attribute Grammars, Publication No. 148, Dept. Informatique, Universitē de Montreal, Canada (1973).

12. Böhm, C. and Jacopini, G. "Flow Diagrams, Turing Machines and Languages with only two formulation rules", Comm.Assoc. Comp.Mach. 9(5), pp. 366-371 (1966).

13. Böhm, C. The CUCH as a Formal and Description Language, ref. [100], pp. 179-197 (1966).

14. Böhm, C. "Introduction to CUCH", in Automata Theory (Ed. E.R. Caianiello), Academic Press, New York, (1966).

15. Buxton, J.N. et al. CPL Working Papers, Technical Report by the University of London Institute of Computer Science and the Mathematical Laboratory, University of Cambridge (1966).

16. Carnap, R. Introduction to Symbolic Logic and its Applications, Dover Publications, New York (1958), p. 79.

17. Cheatham, T.E. and Satley, K. "Syntax Directed Compiling", Proc. AFIPS (SJCC), 25, pp. 31-57, Washington (1964).

18. Church, A. The Calculi of Lambda-conversion, Ann.Math. Stud. No. 6, Princeton University (1941).

19. Clint, M. and Hoare, C.A.R. "Program Proving: jumps and functions", Acta Informatica, 1, pp. 214-224 (1972).

20. Cocke, J. and Schwartz, J.T. Programming Languages and their Compilers, Courant Institute of Mathematical Sciences, New York (1970).

21. Cocke, J. "Global Common Subexpression Elimination", SIGPLAN Notices, 5(7), pp. 20-24 (1970).

22. Cooke, D.J. CARABINER Paper I, "On Substitution Operators", University of London Institute of Computer Science, 1st Edn. Internal document ICSI 437 (1972), 2nd Edn. Internal document ICSI 526 (1974).

23. Cooke, D.J. CARABINER Paper II, "On Extended Markov Algorithms", University of London Institute of Computer Science, 1st Edn. Internal document, ICSI 445 (1972), 2nd Edn. Internal document ICSI 528 (1974).

24. Cooke, D.J. CARABINER Paper III, "On Control Routines", University of London Institute of Computer Science, Internal document ICSI 456 (1972).

25. Cooke, D.J. CARABINER Paper IV, "On Syntax Modifications", University of London Institute of Computer Science, 1st Edn. Internal document ICSI 482 (1973), 2nd Edn. Internal document ICSI 527 (1974).

26. Cooke, D.J. CARABINER Paper V, "The Carabiner Program Space", University of London Institute of Computer Science, Internal document ICSI 525 (1974).

27. Cooke, D.J. CARABINER Paper VI, "On the Construction of BNF Syntax", University of London Institute of Computer Science, Internal document ICSI 499 (1973).

28. Cooke, D.J. CARABINER Paper VII, "On Carabiner Control Translations - I", University of London Institute of Computer Science, Internal document ICSI 512 (1973).

29. Cooke, D.J. CARABINER Paper VIII, "On Carabiner Control Translations - II", University of London Institute of Computer Science, Internal document ICSI 515 (1973).

30. Cooke, D.J. CARABINER Paper IX, "On the Formalization of Procedure Activation", University of London Institute of Computer Science, Internal document ICSI 521 (1973).

31. Cooke, D.J. CARABINER Paper X, "On Parameters, Pointers and Values", University of London Institute of Computer Science, Internal document ICSI 523 (1974).

32. Cooke, D.J. The Carabiner Project. An Overview, University of London Institute of Computer Science, Internal document ICSI 529 (1974).

33. Cooke, D.J. PLECS. A set of Programming Languages to Exhibit the use of Carabiner (part 1), University of London Institute of Computer Science, Internal document ICSI 530 (1974).

34. Cooke, D.J. PLECS (part 2), Loughborough University of Technology Dept. of Computer Studies, technical report (in preparation 1974).

35. Currie, I.F. et al. "ALGOL 68-R", in ALGOL-68 Implementation Ed. J.E.L. Peck, pp. 21-34, North-Holland (1971).

36. Dennis, J.B. On the Design and Specification of a Common Base Language, Project MAC, TR-101, M.I.T. (1972).

37. Dijkstra, E.W. "The Structure of 'THE'-multiprogramming System", Comm.Assoc.Comp.Mach. 11(3) pp. 341-346 (1968).

38. Dijkstra, E.W. Notes on Structured Programming, Tech. Rept. EWD249, Dept. of Mathematics, Technical University, Eindhoven (1969).

39. Dijkstra, E.W. A Short Introduction to the Art of Programming, Tech.Rept. EWD316, Dept. of Mathematics, Technical University Eindhoven (1971).

40. Dijkstra, E.W. A Simple Axiomatic Basis for Programming Language Constructs, Tech.Rept. EWD372, Dept. of Mathematics, Technical University Eindhoven (1973).

41. Dijkstra, E.W. "The Composition of Programs Guided by their Correctness Proofs", invited lecture given at the Second IUCC Computer Science Colloquium, Kent (1973).

42. Engeler, E. "Structure and Meaning of Elementary Programs", [43], pp. 89-101 (1971).

43. Engeler, E. Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics No. 188, Springer, Berlin (1971).

44. Feldman, J.A. "A Formal Semantics for Computer Languages and its application to a Compiler-Compiler", Comm.Assoc.Comp. Mach. 9(1), pp. 3-9 (1966).

45. Foster, J.M. List Processing, MacDonald (Computer Monographs Series No. 1), London (1967).

46. Foster, J.M. "A Syntax Improving Program", Computer Journal, 11, pp. 31-34 (1968).

47. Fraleigh, J.B. A First Course in Abstract Algebra, Addison-Wesley (1967).

48. Galler, B.A. and Perlis, A.J. A View of Programming Languages, Addison-Wesley (1970).

49. Greibach, S.A. Inverses of Phrase-Structure Generators, Doct. Thesis, Div. Eng. and Appl. Physics, Harvard U., Cambridge Mass. (1963).

50. Greibach, S.A. "Formal Parsing Systems", Comm. Assoc. Comp. Mach. 7, pp. 499-504 (1964).

51. Greibach, S.A. "A new Normal-Form for Context-Free Phrase Structure Grammars", J.Assoc. Comp. Mach. 12(1), pp. 42-52 (1965).

52. Harary, H. et al. Structural Models, J. Wiley, New York (1965).

53. Harary, H. Graph Theory, Addison-Wesley 1969).

54. Henhapl, W. and Jones, C.B. The Block Structure Concept and some possible Implementations with Proofs of Equivalence, TR 25.104, IBM Vienna (1970).

55. Igarashi, S. et al. ALGOL N, Research Institute for Mathematical Sciences, University of Kyoto, Japan, Document 66 (1969).

56. Ingerman, P.Z. A Syntax Oriented Translator, Academic Press (1966).

57. Irons, E.T. "The Structure and Use of the Syntax-Directed Compiler", Annual Review of Automatic Programming, 3, pp. 207-227, Pergamon Press, (1963).

58. Knuth, D.E. and Floyd, R.W. "Notes on avoiding 'goto' statements", Computer Science Dept. Report CS 148, Stanford U. (1970), and Information Processing Letters 1, pp. 23-31 (1971).

59. Knuth, D.E. "Top-down Syntax Analysis", Acta Informatica 1(2), pp. 79-110 (1971).

60. Kripke, B. Introduction to Analysis, W.H. Freeman, San Francisco (1968).

61. Landin, P.J. "The Mechanical Evaluation of Expressions", Computer Journal, 6, pp. 308-320 (1964).

62. Landin, P.J A Formal Description of ALGOL 60, [100], pp. 266-294 (1964).

63. Landin, P.J. "A Correspondence between ALGOL 60 and Church's lambda notation", Comm. Assoc. Comp. Mach. 8, pp. 89-101 and pp. 158-165 (1965).

64. Lang, S. Algebra I. §8, Addison-Wesley, Reading (Mass.) (1965).

65. Lang, S. Analysis I, Addison-Wesley, Reading (Mass.) (1968).

66. Lauer, P. Formal Definition of ALGOL 60, TR 25.088, IBM Vienna (1968).

67. Leavenworth, B.M. "Programming with(out) the GOTO", Sigplan Notices, 7(11), pp. 54-58 (1972).

68. Ledgard, H.F. "Ten Mini-languages", Computing Surveys 3(3), pp. 115-146 (1971).

69. Lucas, P. Two Constructive Realizations of the Block Concept and their Equivalence, TR 25.085, IBM Vienna (1968).

70. Lucas, P. et al. Method and Notation for the Formal Definition of Programming Languages, TR 25.087, IBM Vienna, (1968).

71. MacLane, S. Homology, p. 122, Lemma 7.2 with $\varepsilon_j = +1$, Academic Press, New York (1963).

72. MacLane, S. and Birkhoff, G. Algebra, p. 42, Macmillan, New York (1967).

73. ibid. p. 138.

74. Markov, A.A. Theory of Algorithms, U.S.S.R. Academy of Sciences (1954); English translation by the Israeli Program for Scientific Translations (1961).

75. McCarthy, J. "A Basis for a Mathematical Theory of Computation", in Computer Programming and Formal Systems, (ed. Braffort and Hirschberg), North-Holland Publishing Co., Amsterdam (1963).

76. McCarthy, J. A Formal Description of a subset of ALGOL, [100] pp. 1-12 (1964).

77. McKeeman, W.M. et al. A Compiler Generator, Prentice-Hall, Englewood Cliffs, N.J. (1970).

78. Mendelson, E. Introduction to Mathematical Logic, Van Nostrand, New York (1964).

79. Mock, O. et al. "The Problem of Programming Communications with changing Machines: a proposed solution", Comm. Assoc. Comp. Mach. 1(8), pp. 12-18, 1(9), pp. 9-15 (1958).

80. Moss, R.M.F. and Roberts, G.T. A Preliminary Course in Analysis, Chapman and Hall, London (1968).

81. Mosses, P. The Mathematical Semantics of ALGOL 60, Tech. Mon. PRG-12, Programming Research Group, Oxford U. (1974).

82. Naur, P. (Editor) et al. "Revised Report on the Algorithmic Language ALGOL-60", Comm.Assoc. Comp. Mach. 6(1), pp. 1-17 (and elsewhere) (1963).

83. Naur, P. "Proof of Algorithms by General Snapshots", BIT, 6, pp. 310-316 (1966).

84. Nivat, M. and Nolin, L. Contribution to the Definition of ALGOL Semantics, [100] pp. 148-159 (1964).

85. Nivat, M. and Nolin, L. "Sur un Procédé de Définition de la Syntaxe d'ALGOL", Proc. 3rd AFCALT Congress (Toulouse), (1963).

86. Nixon, E. and Wesselkamper, T.C. CRAMPON Paper I, University of London Institute of Computer Science, internal document ICSI 294 (1970).

87. Nixon, E. and Wesselkamper, T.C. CRAMPON Paper II, University of London Institute of Computer Science, internal document ICSI 309 (1971).

88. Peterson, W.W. et al. "On the Capabilities of While, Repeat and Exit statements," Comm. Assoc. Comp. Mach. 16(8), pp. 503-512 (1973).

89. Schumann, S.A. and Jorrand, P. "Definition Mechanisms in Extensible Programming Languages", proc. F.J.C.C., AFIPS (37), pp. 9-20 (1970).

90. Scott, D. Outline of a Mathematical Theory of Computation, Tech.Mon. PRG-2, Programming Research Group, Oxford U. (1970).

91. Scott, D. The Lattice of Flow Diagrams, Tech. Mon. PRG-3, Programming Research Group, Oxford U. (1970).

92. Scott, D. and Strachey, C. Towards a Mathematical Semantics for Computer Languages, Tech. Mon. PRG-6, Programming Research Group, Oxford U. (1971).

93. Snidvongs, K. Towards a Calculus of S-Expressions Paper I, "On Finite Spaces of Disjoint Elements", University of London, Institute of Computer Science, internal document ICSI 496 (1973).

94. Snidvongs, K. Towards a Calculus of S-Expressions Paper II, "Over Generalized Finite Spaces", University of London Institute of Computer Science, internal document ICSI 497 (1973).

95. Snidvongs, K. Towards a Calculus of S-Expressions Paper III, "The Transformation of Simple Programs", University of London Institute of Computer Science, internal document ICSI 514 (1973).

96. Snidvongs, K. Towards a Calculus of S-Expressions Paper IV, "On Conditional Expressions", University of London, Institute of Computer Science, internal document ICSI 524 (1974).

97. Snidvongs, K. Towards a Calculus of S-Expressions Paper V, "On Control Structures", University of London, Institute of Computer Science, internal document ICSI 533 (1974).

98. Steel, T.B. "UNCOL", Datamation, 6(1), pp. 18-20 (1960).

99. Steel, T.B. "UNCOL: The Myth and the Fact", Annual Revue of Automatic Programming 2, pp. 325-344 (1961).

100. Steel, T.B. (ed.) Formal Language Description Languages for Computer Programming, Proc. IFIP Working Conference, Vienna (1964), Pub. North-Holland (1966).

101. Strachey, C. Towards a Formal Semantics, [100], pp. 198-220 (1964).

102. Strachey, C. Varieties in Programming Languages, Tech. Mon. PRG-10, Programming Research Group, Oxford U. (1972).

103. Strachey, C. and Wadsworth, C.P. Continuations: A Mathematical Semantics with Full Jumps, Tech. Mon. PRG-11, Programming Research Group, Oxford U. (1974).

104. Suzuki, N. et al. "The Implementation of ALGOL N", Proceedings of an International Symposium on Extensible Languages, Grenoble, France (Sept. 1971); reproduced as SIGPLAN Notices, 6(12), pp. 15-21 (1971).

105. Taylor, A.E. Introduction to Functional Analysis, J. Wiley, New York (1958).

106. Van Wijngaarden, A. "Generalized ALGOL", Annual Revue of Automatic Programming, 3, pp. 17-26 (1963).

107. Van Wijngaarden, A. Recursive Definition of Syntax and Semantics [100], pp. 13-24 (1964).

108. Van Wijngaarden, A. (editor) et al. Report on the Algorithmic Language ALGOL 68, Mathematisch Centrum Amsterdam, MR 101 (1969).

109. Vettes, F. A General Method of Automatic Syntax Analysis, University of London, Institute of Computer Science, internal document ICSI 531 (1974).

110. Wegner, E. "A Hierarchy of Control Structures", MOL Bulletin, 1, pp. 5-11 (1972).

111. Wegner, E. "Tree-Structured Programs", MOL Bulletin, 2, (1973).

112. Wegner, P. "The Vienna Definition Language", Computing Surveys, 4(1), pp. 5-63 (1972).

113. Wells, M. and Denison, A. "Direct Execution of Programming Languages", Computer Journal, 17(2), pp. 130-134 (1974).

114. Wesselkamper, T.C. CRAMPON Paper III, University of London, Institute of Computer Science, internal document ICSI 333 (1971).

115. Wesselkamper, T.C. CRAMPON Paper IV, University of London, Institute of Computer Science, internal document ICSI 367 (1972).

116. Wesselkamper, T.C. A Mathematical Model of the Computing Process in a high Level Language, Doct. Thesis, University of London (1972).

117. Wirth, N. and Weber, H. "EULER, a Generalization of ALGOL, and its Formal Definition", Comm. Assoc. Comp. Mach. 9, pp. 13-23 and pp. 89-99 (1966).

118. Wirth, N. Systematic Programming: An Introduction, Prentice-Hall (1973).

119. Wood, D. "The Theory of Left-Factored Languages", Computer Journal, 12(4), pp. 349-356 and 13(1), pp. 55-62 (1969-70)

120. Woodger, M. "On Semantic Levels in Programming", Proc. IFIP Congress, pp. 402-407, Ljubljana (1971).

121. Woodward, P.M. A Note on Foster's Syntax Improving Device, RRE Memo. 2352, Royal Radar Establishment, Malvern (1966).

122. Wulf, W.A. "Programming without the 'goto' ", Proc. IFIP Congress '71, Ljubljana (1971).

123. Wulf, W.A. "A Case Against the 'goto'", SIGPLAN Notices, 7(11) pp. 63-69 (1972).

124. Yoneda, N. "The Description and the Structure of ALGOL N", Proc. International Symposium on Extensible Languages, Grenoble, France (Sept. 1971); reproduced as SIGPLAN Notices, 6(12), pp. 10-14 (1971).

125. Zurcher, F.W. and Randell, B. "Iterative Multi-Level Modelling - A Methodology for Computer System Design", Proc. IFIP Congress '68, pp. 867-871, Edinburgh (1968).

126. Golde, H. et al. "Report on Sublanguages", ALGOL Bulletin, AB33.3.5 (1972).

APPENDIX: THE REMOVAL OF 'GOTO'S

A1. Introduction

Herein we give a language-independent algorithm which acts on a program, characterized by a flow-chart, and creates an equivalent[†] program which is modular or 'structured'.

Whilst fully recognising that (well-)'structured' programming [38] (otherwise called step-wise refinement [118], iterative multi-level modelling [125], system hierarchy [37], step-wise program composition [39], top-down program development, etc.) is fundamentally concerned with program construction, we note that the properties of structured programs are far easier to derive and manipulate than those of more general programs [39, 40, 41], so much so, that one could justifiably spend time in obtaining a program that is equivalent to the original one and yet is composed in such a way that optimizing techniques can more readily be applied [2, 19-21]. A more complete discussion of the 'goto' problem [3, 4, 9, 12, 38, 58, 67, 88, 110, 111, 122, 123] is given in Carabiner working paper no. 7 [28]. That paper also contains an extended version of the algorithm set out below. In that version, several optimization stages are incorporated with the effect of producing results which compare favourably with a similar algorithm published recently [88].

[†] i.e. the new program defines the same function (represented by input-output pairs) as the original program.

The algorithm presented here acts on a single entry/single (logical) exit flow diagram, and includes no optimizations. It preserves the original topology (up to identification) and introduces no extra variables† or predicates†. The transformation from multi-entry/multi-exit program segments into ones with only one entry and one exit is usually dependent on language semantics and is considered elsewhere [29].

Throughout the appendix we shall regard Carabiner as having no explicit 'goto' statement and consequently PTL is included as a fundamental operation (as discussed in §6.5). To signify that we are using this restricted version of Carabiner we shall refer to it as Carabiner*.

A2. Carabiner* Structure

Carabiner* is a block-structured language with five control operations, two of which deal with the block entry and exit, whilst the others deal with control flow within blocks. We consider the in-block operations first.

In-Block Structure

A Carabiner* block is a linear list of statements and has no labels. In the absence of any explicit control directive the evaluation sequence is, by default, strictly linear, i.e. upon completion of one statement the next statement in the list is executed. This is the first (implicit) control operation; given a list of (non-control) statements, S_i, we will denote these by:

---

† such as a state vector or (implicit or explicit) Boolean variables [3, 4] or flags [9].

$S_1, S_2, \ldots, S_n$

or

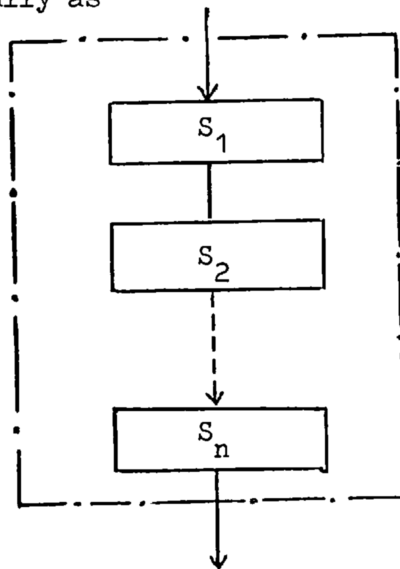
$[S_1$

$[S_2$

$\vdots$

$[S_n$

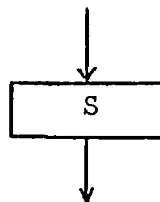
and diagramatically as



In what follows we shall regard any strict sequence of statements as above (i.e. void of control operations) as a single execution sequence and we may write it as:

$[S$

and represent it as the trivial diagram:



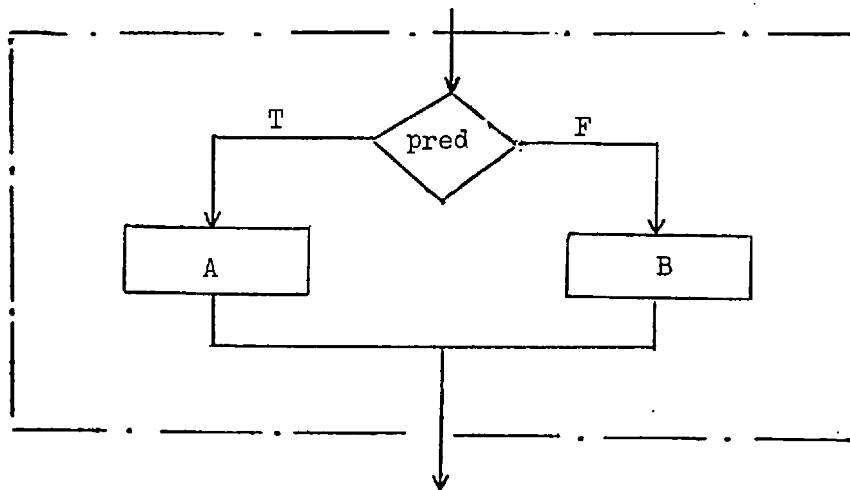
The other two in-block control operations correspond to forward and backward jumps in the flow, i.e. they emulate what in FORTRAN might be:

```
        A = B
    7   B = C + B
        IF (A.GT.B) GO TO 7
          .
          .
          .
or
        GO TO 3
          .
          .
          .
    3   CONTINUE
```

The Carabiner* operation for a forward jump is RAM([A,[B,pred) - i.e. the ramification of the processes [A and [B governed by the predicate 'pred' (see §4.5) and is equivalent to the Algol-60 construct:-

<u>if</u> pred <u>then</u> [A <u>else</u> [B

Diagramatically:

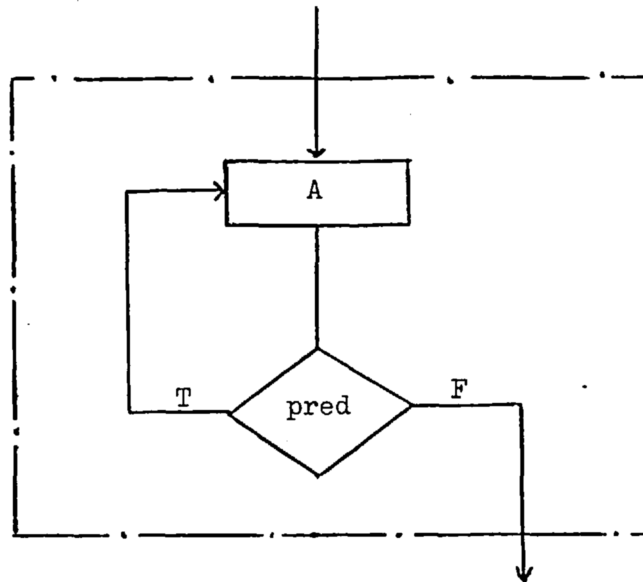


For a backward jump (i.e. a potential loop) we postulate PTL([A, pred), i.e. Process [A, Test pred and if true† then Loop and repeat.

Diagramatically this is:

---

† i.e. if ek pred = True



Block Control

Block control is governed by the two operations BLOCK(x) and EXIT(y), where x is the body of the required block and y is either a positive integer (by default 1) which specifies the number of nested blocks to be exited from, or 'T' which signifies termination of the program, this effectively means exit to operating system level. A FORTRAN subroutine might have the following exit identifications:

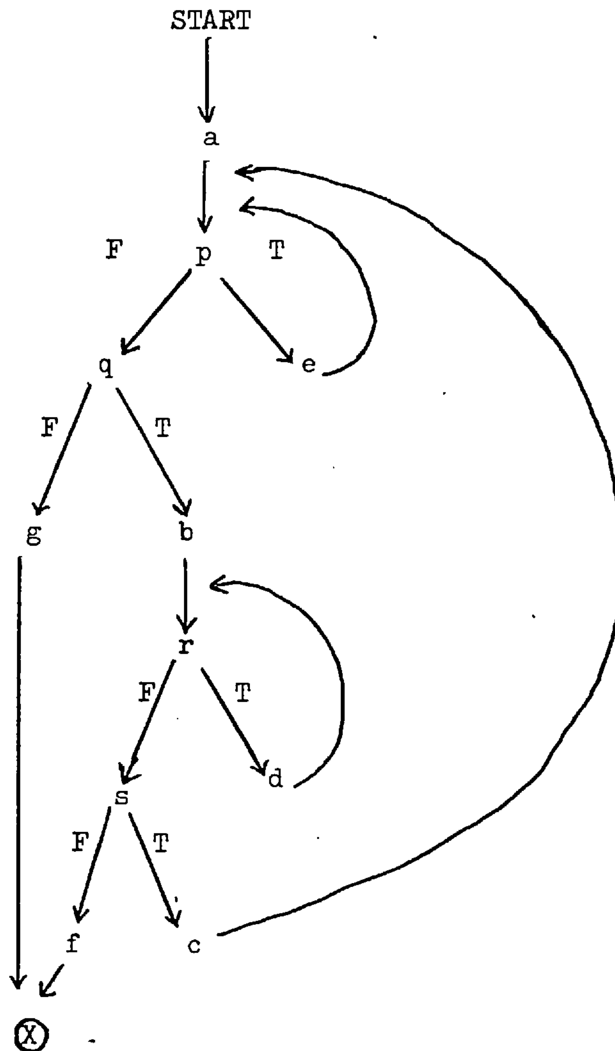
```
SUBROUTINE EXAMPLE(X)
:
IF (Y) STOP                    ⇒ EXIT(T)
:
RETURN                         ⇒ EXIT(1)
:
RETURN                         ⇒ EXIT(1)
END
```

The above example is an over-simplification since it treats a subroutine as merely a block and ignores all questions related to the passing of parameters or results. A more useful example is the following Algol-60 program:

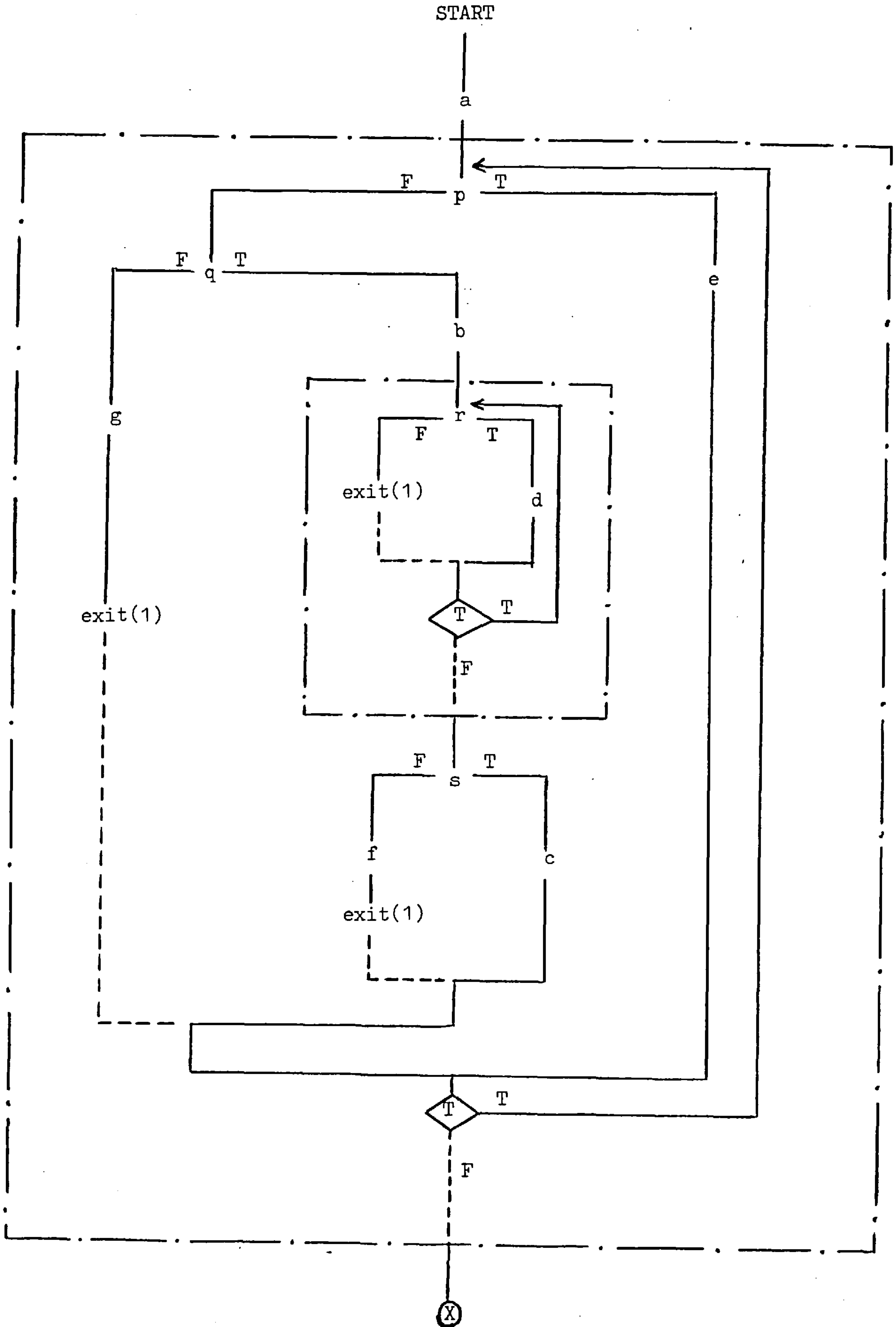
<u>begin</u>	⇒ BLOCK()
⋮	
<u>begin</u>	⇒ BLOCK()
⋮	
<u>end</u>;	⇒ EXIT()
⋮	
<u>end</u>	⇒ EXIT(T)

For simplicity of notation we shall use Ⓧ for a stop or EXIT(T) statement, or return in a subprogram.

<u>Note</u>: The 'in-block' operations of Carabiner* are sufficient to model any structured program [38], and by the addition of the EXIT(n) instruction we may also model non-structured programs, e.g.



This is 'unstructurable' without destroying the topology or adding extra Boolean or state variables [3, 4] but yields, by the algorithm of section 2 the following 'semi-structured' schema:



Here we have used chain lines (—·—·—) to denote blocks and broken lines (- - - - -) to show completion flows: these are never followed in the execution but serve to retain the structured skeleton on which the incomplete segments hang.

To simplify the representations of flow diagrams we shall use digraph representation (as in the above example) with the natural correspondence between process boxes and 2-nodes, bi-decisions (and flow joins) and 3 nodes etc. This realization will not be formalized.

A3. The Translation Algorithm

The translation process and the intermediate graph constructs used are based on Engeler's normal form [42] and consist of four stages, each of which is described individually. The sufficiency of the algorithm is discussed in section A5.

Stage 1

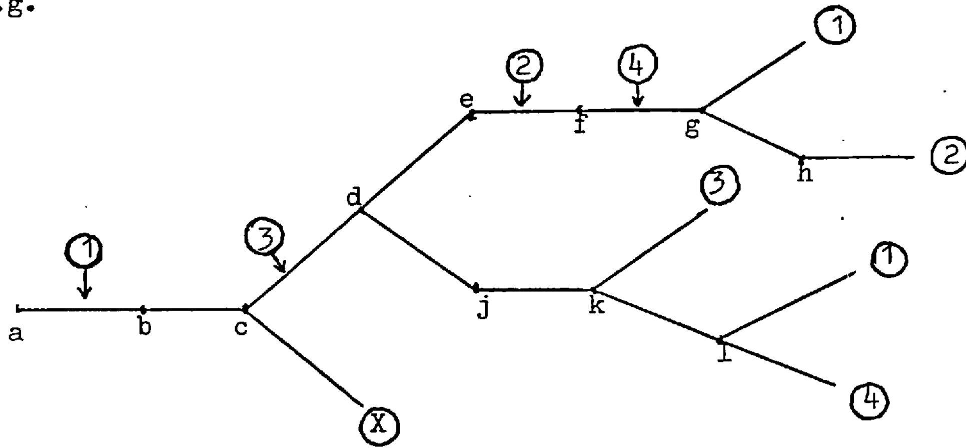
Transformation of a flow-chart into a 'linked tree'.

Defn: A linked tree is a tree† with leaves labelled by integers or a special symbol Ⓧ. (This denotes a source language block exit, e.g. stop). To each such integer there is at least one corresponding pointer placed between two adjacent nodes (neither of which is a numbered leaf) elsewhere in the tree. The links being implied by the association between equal integers.

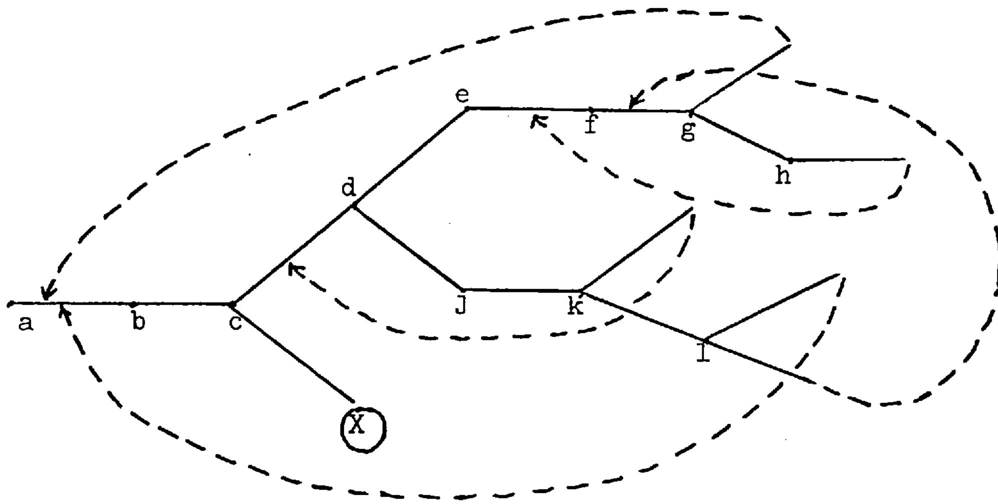
---

† with all nodes of outdegree ≤ 2 (see §5.2.2).

e.g.



Here the implicit links are:



Stage 1 Translation

(i) Label all nodes of the flow-chart in a suitable manner (e.g. by lower case letters) and all flow joins by distinct integers. Replace all multiple (> 2) decisions by sequences of binary choices.

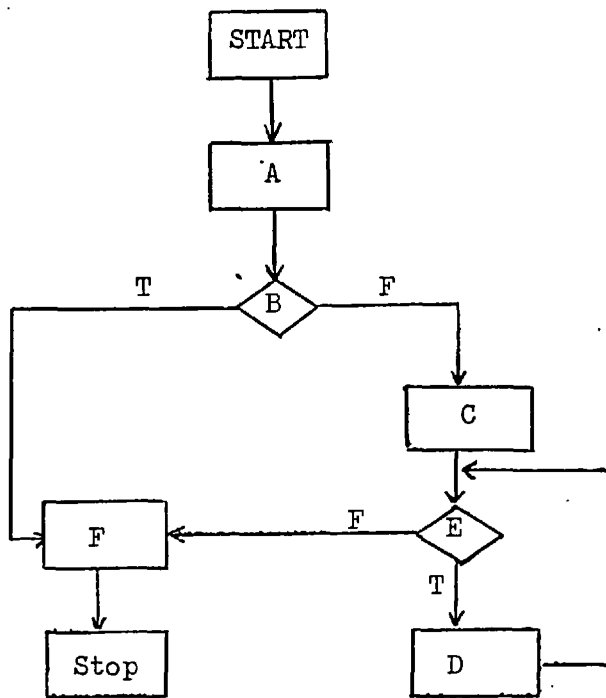
(ii) Starting at the entry point, construct the first branch of the tree, following the True (T) branches at predicates until either, (a) a stop, Ⓧ, is reached or, (b) a node is encountered for the second time; in this case do not duplicate the node but terminate the branch by a leaf named by the respective integer. (i.e. the one associated with the join in the flow.)

(iii) Taking the highest incomplete branch (i.e. the one furthest from the root) repeat the method of (ii) to complete all branches without duplicating any node of the original flow-chart.

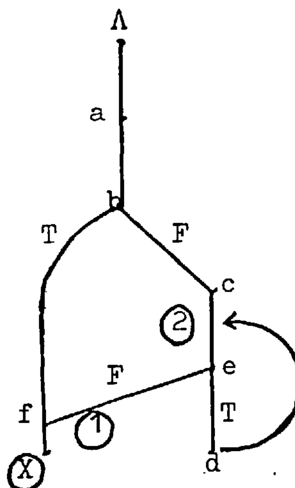
e.g. 1: Program:-

```
      [A
      if B then goto lab
      [C
      do [D while E
  lab [F
      Stop
```

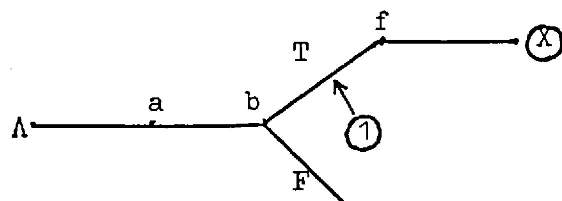
Flow-chart:-

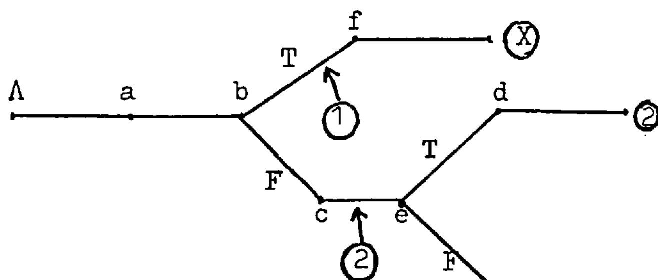


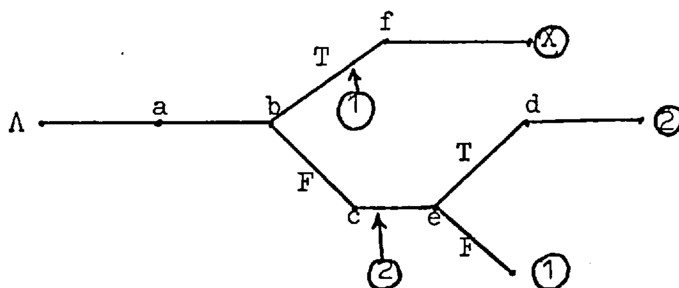
labelled flow-chart:-



Linked tree (by branches):







e.g. 2: program: if A then [B else [C

lab [D

if E then goto lab else goto lab 1

[F

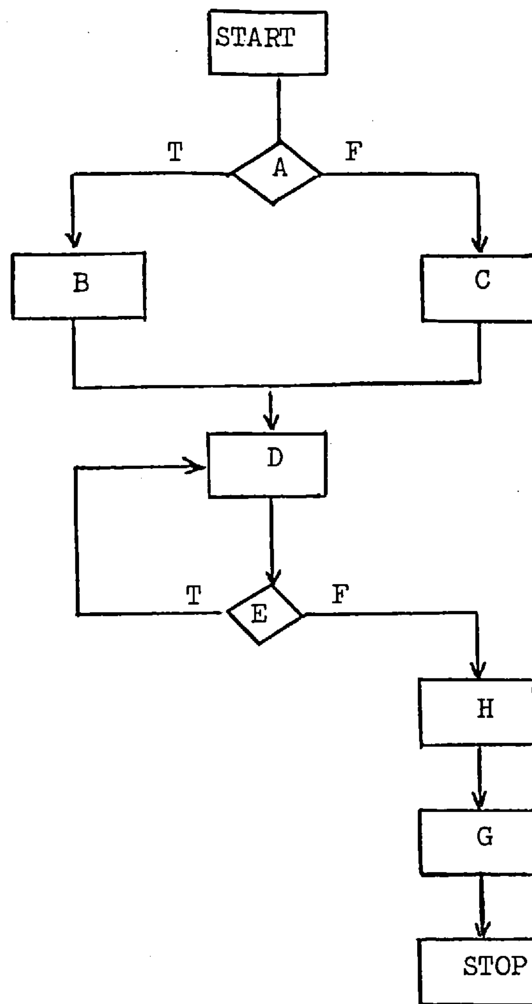
lab 2 [G

Stop

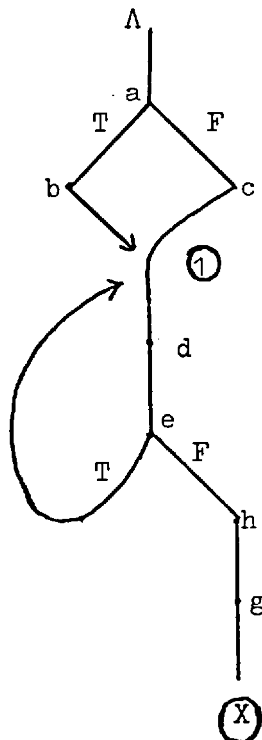
lab 1 [H

goto lab 2

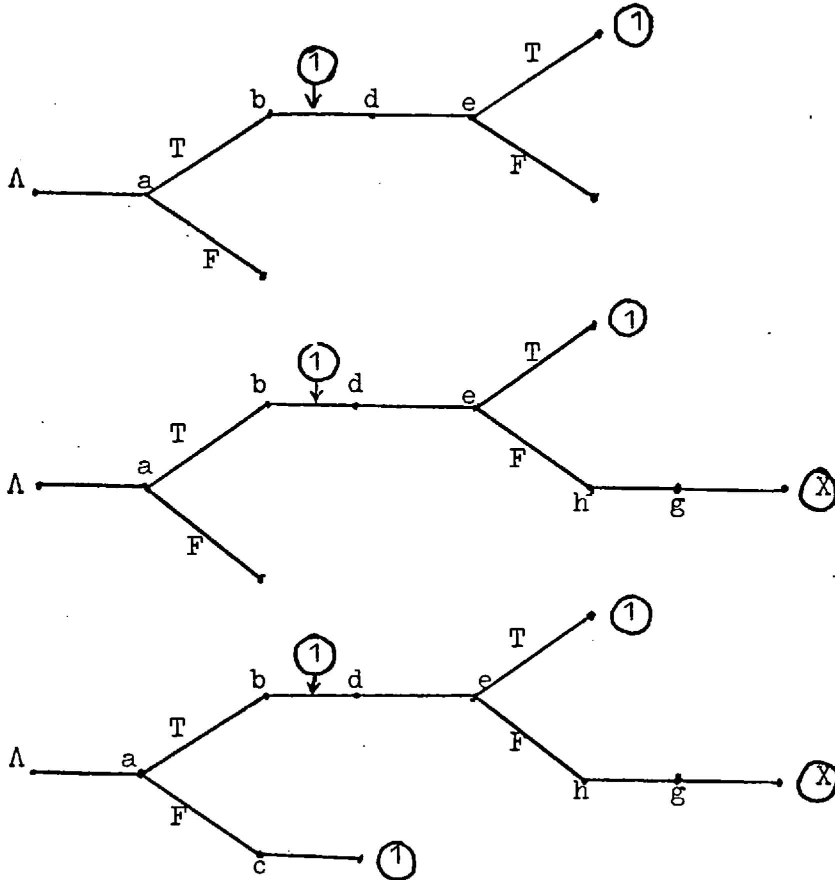
Flow-chart:



labelled flow-chart:



linked-tree (branch by branch):



Stage 2

The linked tree produced by stage 1 needs to have its links untangled. If we regard $<$ as the order relation implicit within the tree so that root $<$ node for all (non-root) nodes etc. and we denote label n by l_n such that if

$$\frac{l_n}{x \qquad y}$$

then

$$p < l_n \qquad \text{for all} \quad p < x,$$

$$x < l_n ,$$

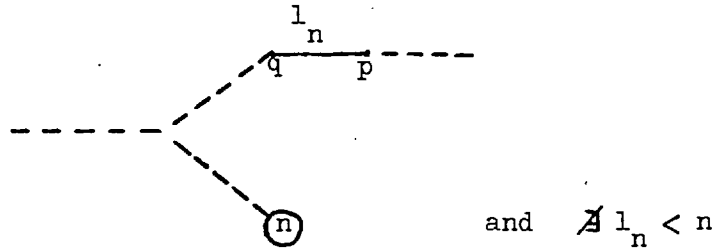
$$l_n < y$$

and

$$l_n < q \qquad \text{for all} \quad q > y.$$

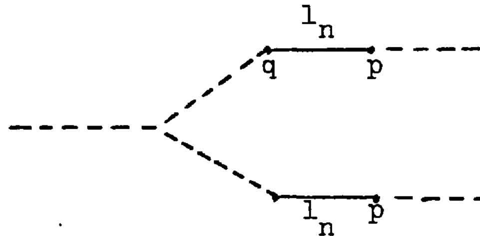
(i.e. regard l_n as a null node). Then we modify the tree so that all leaves n are such that $l_n < n$. This is trivially possible by duplicating part of the tree whenever $l_n \nless n$.

i.e. if



and $\nexists\, l_n < n$

then, delete ⓝ and replace by the subtree whose root is l_n; so we have:



We now formalise the translation phase described above:

<u>Defn</u>: Using the 'l_n' notation and the natural ordering derived from the tree then any tree with the property that $l_n < n$ for all leaves n is called a <u>simply-linked tree</u> (SLT).

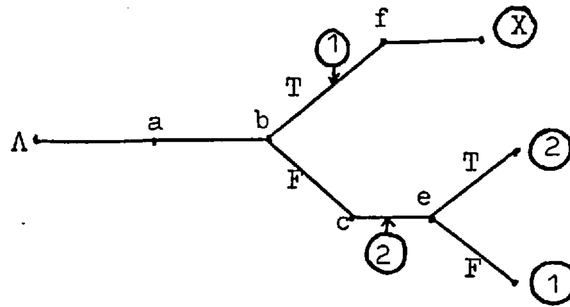
Stage 2 may therefore be regarded as the manipulation of a linked tree into an equivalent simply-linked tree.

<u>Stage 2 Translation</u>:

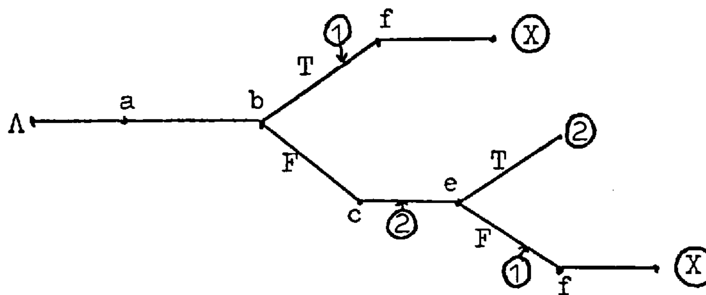
Consider the leaves in the order dictated by the 'True first' rule as in Stage 1(ii).

(i) Find the first leaf 'n' such that $l_n \nless n$. (If there are none then we are done), remove this leaf and replace it by the subtree whose root is l_n. (This in general creates new copies of numbered leaves and removes one leaf.) Goto (i).

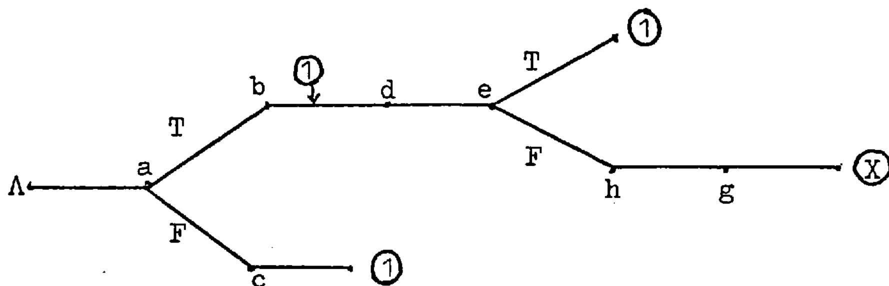
e.g. 1: linked tree:



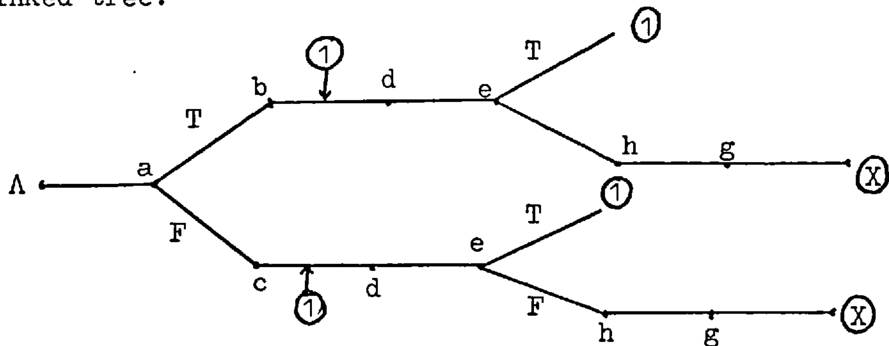
simply-linked tree:



e.g. 2: linked tree:



simply-linked tree:



Notice that Stage 2 may create duplicate labels. In this case if there is more than one on the <u>same</u> branch, erase the one closest to the root.

This technique is called node-splitting [2, 20, 21] and preserves topology (by identifying split nodes, any path in the LT is transformed into an equal path, described by a sequence of node names, in the resulting SLT).

Stage 3:

We now impose some nested blocks onto the directed graph.

Stage 3 Translation:

(i) Erase all unused labels.

(ii) The remaining labels now have a partial order relation imposed by the (tree-like) directed graph. Draw a system of blocks ordered in the manner determined by the remaining labels†.

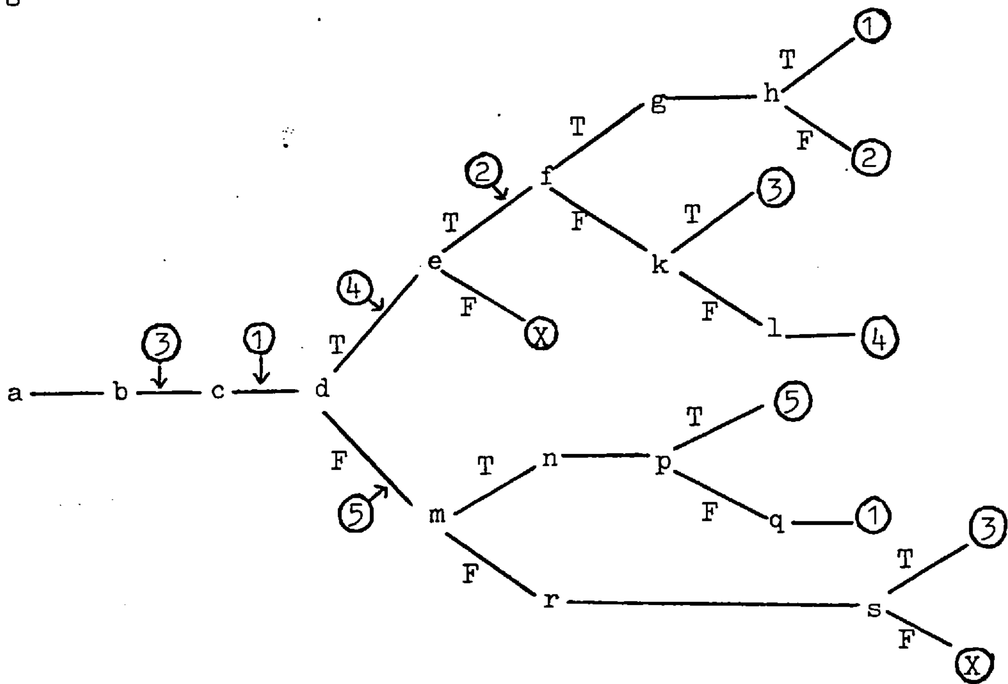
(iii) Upon this system of blocks superimpose the flow graph in such a way

- (a) that the entry point and all termination leaves are outside the largest block;
- (b) that entry to a block coincides with the positioning of the corresponding label;
- (c) that numbered leaves are drawn on a block boundary;
- (d) that the flow only crosses a block boundary in order to reach an outer boundary or X as in (a), and
- (e) that all statements whose position is not dictated by (a) - (d) is located in the most deeply nested block allowable.

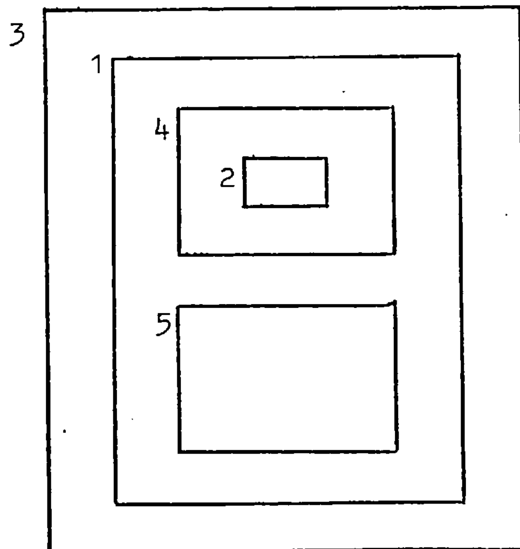
---

†Note that two blocks may relate to the same label but in this case the occurrences are on different branches.

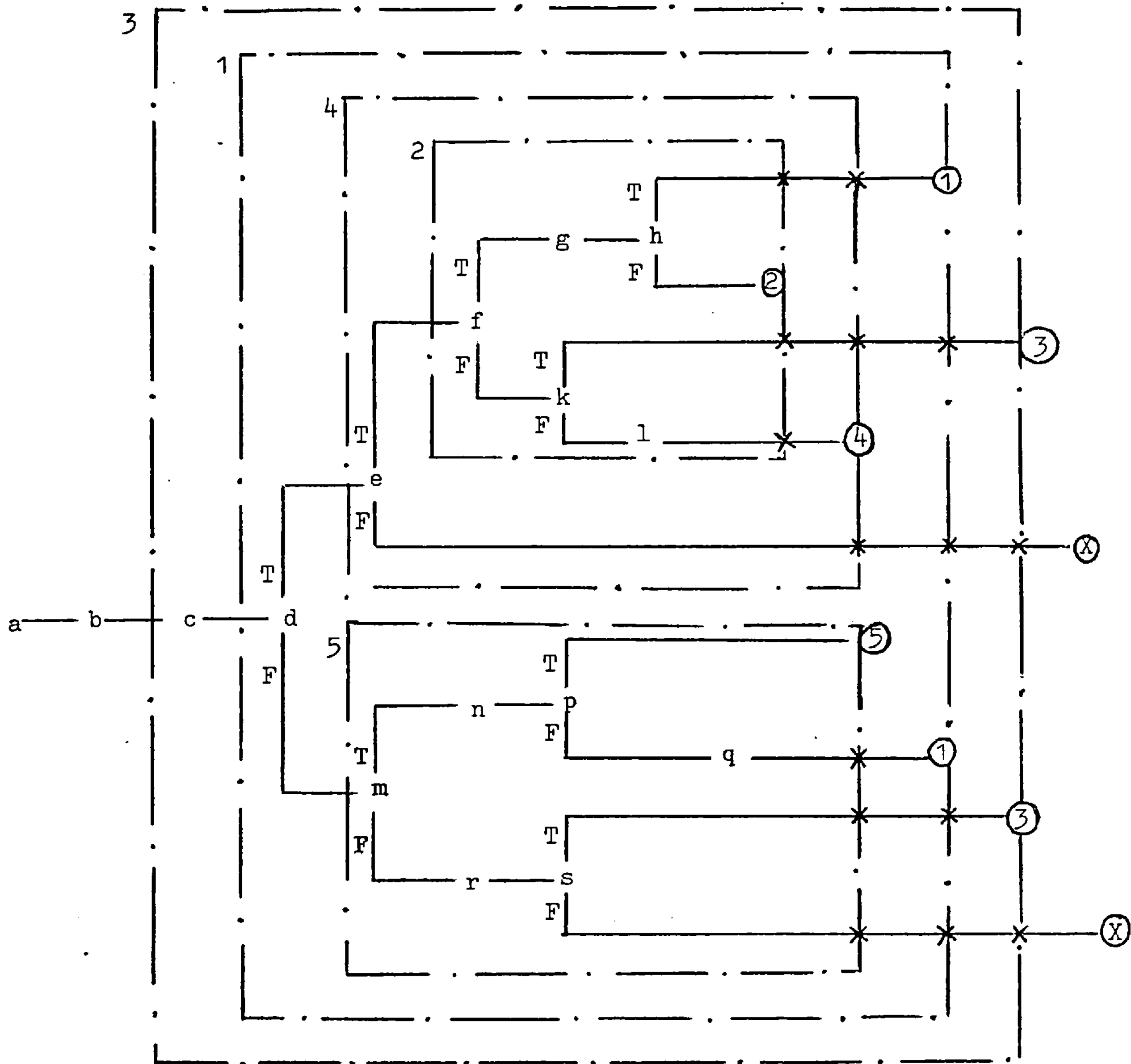
e.g.



underlying block structure:



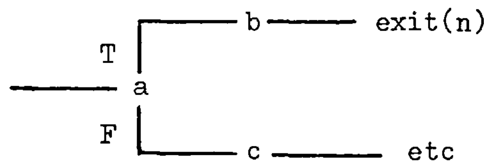
Flow mapped onto this block structure:



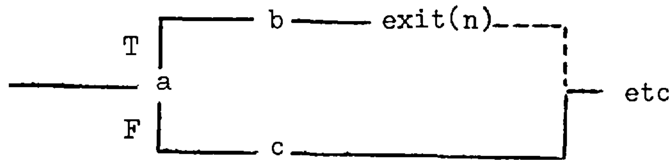
Stage 4:

By virtue of stage 3 part (iii)(e) we arrange that each block has only one immediate exit, hence when we have a sequence of n ($n \geq 1$) consecutive block exits we may replace this by EXIT (n) and either terminate the flow within the block or draw it <u>anywhere</u>. This leads naturally to the completion of the flow in a structured fashion by completing 'open' if-then-else constructs.

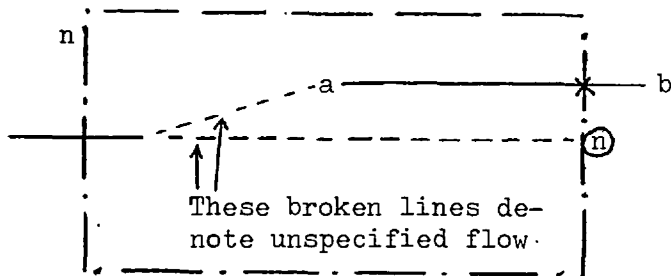
e.g.

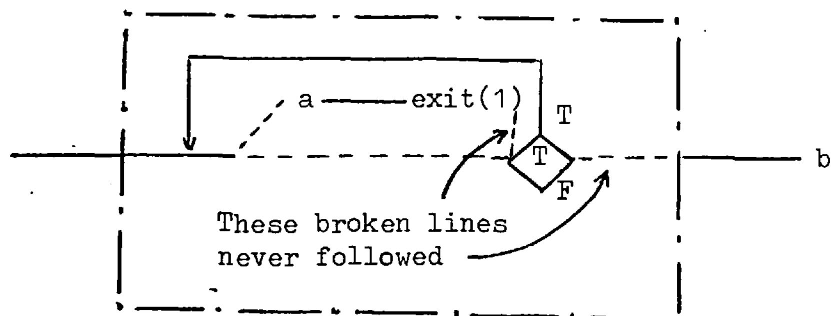


may be drawn:



Similarly, we may explicitly draw the loop control mechanism:





Stage 4 translation:

(i) Insert explicit exit statements.

(ii) Insert explicit loop control.

(iii) Coalesce block exits.

(iv) Complete flow after exits so as to give fully structured (lattice) flow within each implicit or explicit block.

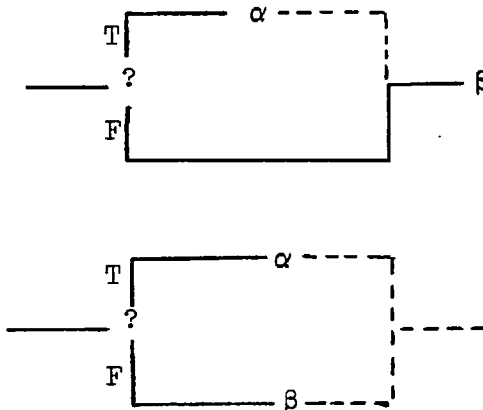
Note: (iv) simply implies the closure of if-then-else phrases should be in reverse order to their creation.

Where 'exit's occur this could lead to ambiguity, hence if sequences terminate in an exit statement there are two courses of action open:

either (a) we stipulate

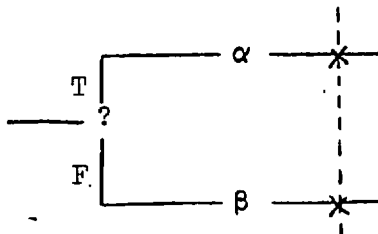
if exit $\in \alpha$ then

if exit $\notin \beta$ then

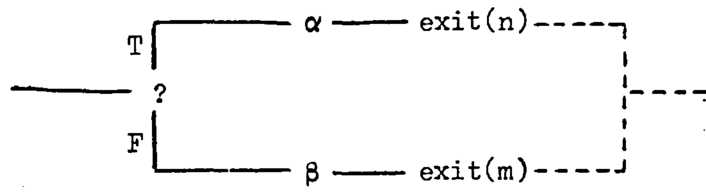


or (b) we let the existing diagram dictate the flow and only merge after exits.

i.e. if



<u>then</u>:

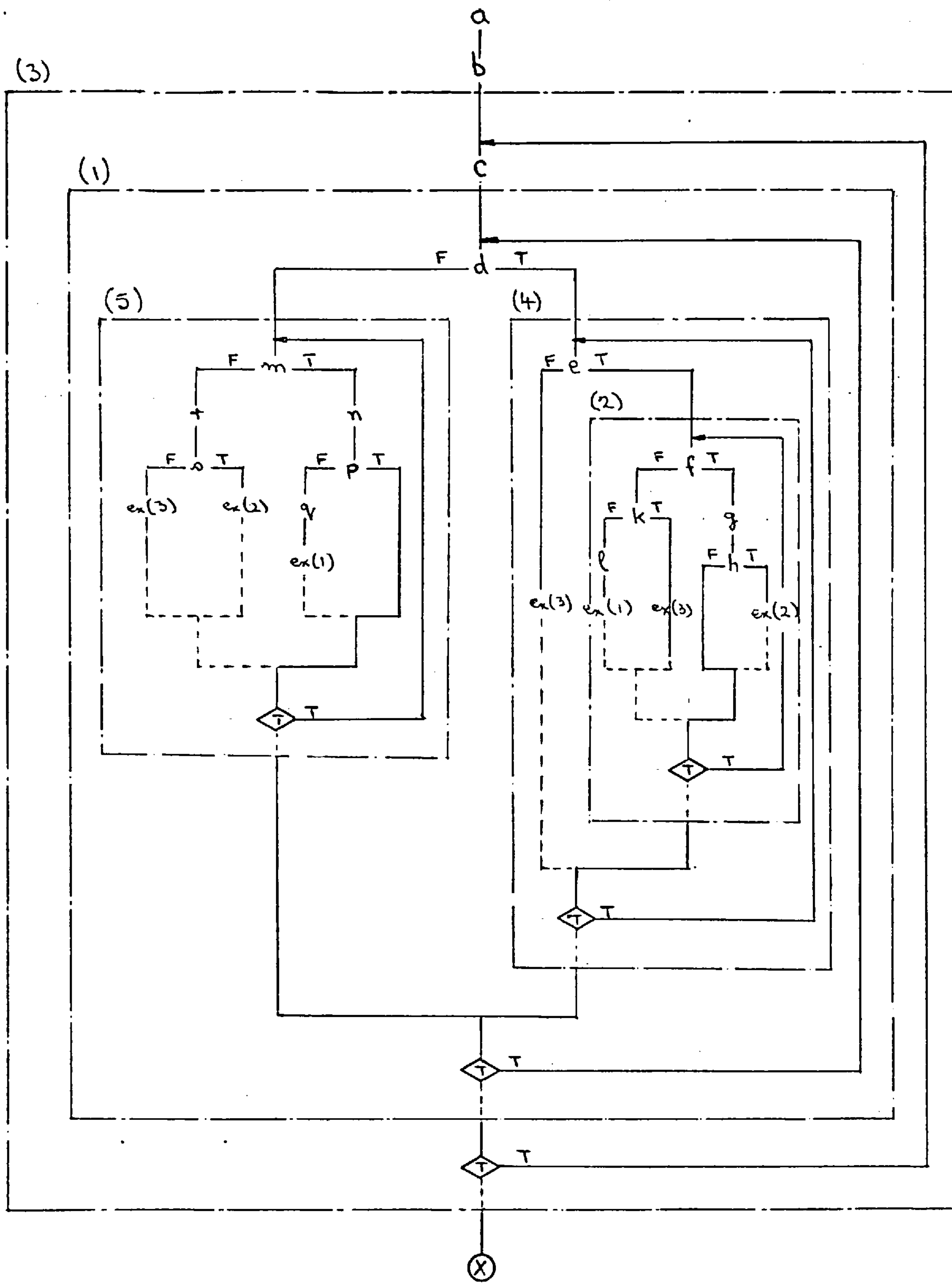


etc.

i.e. let the arms of the predicate be as before.

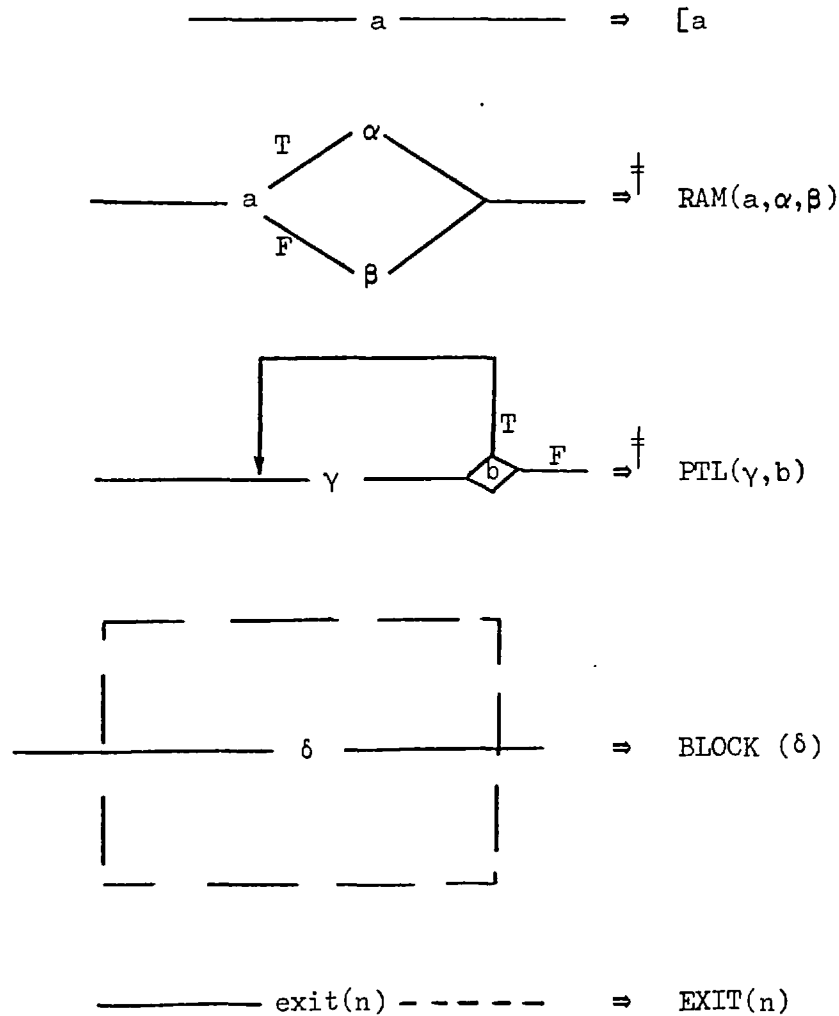
We choose the latter.

e.g. (as used to illustrate stage 3).



A4. Realization of the Carabiner* Program

Having obtained a resulting single entry/single exit diagram, we now set about extracting a Carabiner* program[†]. Using Greek letters to denote arbitrary segments of flow chart and equivalent sequences of Carabiner* statements, we use the following equivalences to dictate the construction of the resultant program from the flow chart after its manipulation by the four stages above.



[†] Of course, equivalent extractions exist for most other languages.

[‡] The same rules are to be applied even when broken lines are present within the flow diagram.

The order of translation of the flow chart is <u>irrelevant</u>; however, to be definitive, we stipulate that the flow lattice shall define the order by beginning translation at the entry node and moving via adjacent nodes (constructs) and expanding each greek denotation as it arises (in RAM expand α before β).

A5. <u>Discussion</u>

<u>Sufficiency of the Algorithm</u>

As stated in the introduction to A3, the notation of an SLT is based on Engeler's normal form [42]. An inductive proof of the existence of a schema of this form equivalent† to a given schema is to be found in section 1 of [42]; however the constructions given above constitute the basis for a proof of what is essentially the same theorem. This theorem states that, given any finite flow diagram with one entry and one exist we can generate an equivalent† flow diagram which uses only the execution statements of the original plus control constructs analogous to our operations of BLOCK, EXIT, RAM and PTL; from this the generation of a program in a sufficiently structured language is trivial, moreover validation of the theorem is easily demonstrated.

Here we do not give a rigourous mathematical proof (although one could easily be formulated) but argue the finite applicability of each stage; the characterization of the output from the final stage, being as in the statement of the theorem, will then infer validation of the theorem.

---

† in the sense that flow-paths (described by a sequence of node denotations) through the schema are preserved.

Stage 1 Translation

Transformation of a (finite) flow chart into a linked tree.

Given a finite directed graph we start from a specific node (the entry point) and select a (necessarily acyclic) chain the length of which is finite.

At this stage either we finish or we have another specific node (the highest incomplete False fork) from which to generate another chain.

This process stops before repetition of any nodes already used and utilizes at least one further node and/or completes another False fork. This last paragraph is then repeated until all branches are complete.

By virtue of the total number of nodes used (and hence predicate nodes and False forks) being finite, this process must terminate.

Stage 2 Translation

Transformation of a linked tree into a simply-linked tree.

In a linked tree there are a finite number of used labels and corresponding leaves. The substitutions caused by stage 2 involve extensions to this tree; the extensions duplicate labels on distinct branches and once a branch has all the used labels (hung) upon it there can be no more substitutions in that branch. From the finite original tree we derive a finite extended tree in a finite number of steps.

The flow modifications are now complete and we have to lift our semi-structured program from the derived tree. The blocking

structure of stage 3 follows directly from the tree-structured relation between the remaining (used) labels of the SLT. Similarly the sub-tree structure of the blocked SLT well defines the 'lattice' completions to be made in stage 4. These final stages are purely manipulative; they cause no extensions of any kind but merely give a structured interpretation of the SLT already derived.

Comparison with other works

We give else where [28], stage by stage modifications of various program schemas taken from other papers. These programs are ones which have been considered 'awkward' by other workers.

The example (given in §A2) is taken from Ashcroft and Manna [3, 4] and demonstrates that the inclusion of block structure renders the use of extra (Boolean or 'state') variables unnecessary.

Other examples, taken from a well-known paper by Knuth and Floyd [58] and from other sources have been processed by (a slightly extended version of) the algorithm, with encouraging results.

We have shown that our construction is always applicable and by incorporating a few fairly trivial optimizations we believe a practical restructuring procedure could easily be developed.