



University
of Glasgow

<https://theses.gla.ac.uk/>

Theses Digitisation:

<https://www.gla.ac.uk/myglasgow/research/enlighten/theses/digitisation/>

This is a digitised version of the original print thesis.

Copyright and moral rights for this work are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This work cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Enlighten: Theses

<https://theses.gla.ac.uk/>
research-enlighten@glasgow.ac.uk

SECURE EXTENSIBLE LANGUAGES,
DESIGN OF

by

R. PETER McE. MORTON

A thesis presented to the
University of Glasgow,
in conformance with the
requirements for the degree,
Doctor of Philosophy

Computing Science Department,

October, 1975.

ProQuest Number: 10647447

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10647447

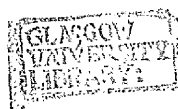
Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Thesis
4340
Copy 2.



ACKNOWLEDGEMENTS

It is a pleasure to acknowledge my debts to Professor D.C. Gilles for his support, guidance and advice throughout the course of this research, and to Mr. J.W. Patterson for his constant fund of enthusiasm, insight and encouragement without which this thesis would never have reached fruition.

I am grateful to Mr. W. Findlay and Dr. D.A. Watt for their patient reading and invaluable criticism.

I acknowledge my gratitude to the Science Research Council, and latterly, to the University of Glasgow for financing this research.

Finally, I thank all members of the Computing Science Department who were of assistance to me in so many ways, both tangible and intangible.

TABLE OF CONTENTS

CHAPTER 0	INTRODUCTION AND HISTORICAL BACKGROUND	
0.0	Introduction	0-1
0.1	Historical Background	0-3
0.2	Specific Problems and Goals of this Thesis	0-9
0.3	Outline of this Thesis	0-10
CHAPTER 1	EXPOSITION ON EXTENSIBLE LANGUAGES	
1.0	Introduction	1-1
1.1	Extensible Languages	1-1
1.2	Terminology	1-6
1.3	Classification Scheme and Diagrammatic Representation	1-8
1.4	Brief Survey of Existing Extensible Languages	1-14
CHAPTER 2	DESIGN CONSIDERATIONS FOR EXTENSIBLE LANGUAGES	
2.0	Introduction and Design Criteria	2-1
2.1	The Design of Secure Programming Languages	2-1
2.1.0	Introduction and Overview	2-1
2.1.1	Notion of Security	2-3
2.1.2	Security, the State of the Art	2-4
2.1.3	Influence of the Features of a Language on Security	2-6
2.1.3.1	Human-Oriented View	2-7
2.1.3.1.1	Undesirable Actions in Programming Languages	2-10
2.1.3.1.2	Notion of Transparency and Overtransparency	2-13
2.1.3.1.3	Ramifications of Overtransparency	2-18
2.1.3.1.4	Stability	2-24
2.1.3.2	Machine-Oriented View of Overtransparency	2-27

2.1.3.3	Language Features for Security, Summary and Conclusions	2-38
2.1.4	Influence of Syntax and Pragmatics on Security	2-40
2.1.5	Summary and Conclusions	2-46
2.2	Security of Extensible Languages	2-47
2.2.0	Introduction	2-47
2.2.1	Security of the Base Language and Extended Language	2-49
2.2.2	Security of the Metalanguage	2-51
2.3	Security of Existing Extensible Languages	2-65
2.4	Proposals for a Secure Extensible Language System	2-72
2.5	String Processing Languages	2-80
2.6	Conclusions	2-84
CHAPTER 3	DESIGN OF A SECURE STRING PROCESSING LANGUAGE	
3.0	Introduction	3-1
3.1	General Principles of Language Design	3-1
3.1.1	Design of Features	3-1
3.1.1.1	General Programming Languages	3-1
3.1.1.2	Base Languages	3-9
3.1.2	Design of Syntax and Pragmatics	3-15
3.2	Design of the Snip Base Language	3-17
3.2.1	Summary of Snip	3-17
3.2.2	Snip Features	3-21
3.2.2.1	Control Structures	3-22
3.2.2.2	Data Structures and Operations	3-26
3.2.2.3	Patterns	3-33
3.2.2.4	Procedures, Functions and Parameters	3-38
3.2.2.5	Program Structuring	3-40
3.2.2.6	Language Extensions	3-42
3.2.3	Syntax and Pragmatics of Snip	3-46

3.2.3.1 Translatability	3-46
3.2.3.2 Security	3-47

CHAPTER 4 IMPLEMENTATION

4.0 Introduction	4-1
4.1 Implementation Strategy	4-1
4.2 Abstract Machine Modelling	4-2
4.2.1 Design of Abstract Machines	4-3
4.3 Design of the Snip Abstract Machine (SAM)	4-5
4.3.1 The Snip Abstract Machine	4-5
4.3.1.1 Conceptual Structure	4-5
4.3.1.2 Architecture	4-11
4.3.2 Design of SAM	4-19
4.3.2.1 Representation of Snip Data Structures	4-20
4.3.2.2 SAM Registers and Operations	4-25
4.4 Implementation of SAM	4-29
4.4.1 Run-Time Store and Interpreter Organisation	4-30
4.4.2 Representation of Snip Data Structures and Statements	4-30
4.4.3 Instruction-Interpreting Routines	4-35
4.4.4 Conclusions from SAM Implementation	4-37
4.5 The Translator	4-37
4.5.1 Processing of Metalanguage Text	4-37
4.5.2 Processing of Augment Text	4-40

CHAPTER 5 CONCLUSIONS AND FURTHER RESEARCH

5.0 Introduction	5-1
5.1 Review and Critique	5-1
5.2 Future Research	5-11

APPENDIX A INFORMAL DESCRIPTION OF BASE SNIP

A.0 Introduction	A-1
A.1 Summary	A-1

A.2	Notation, Terminology and Vocabulary	A-1
A.3	Identifiers, Numbers and String Literals	A-2
A.4	Constant Definitions	A-3
A.5	Data Types	A-3
A.5.1	Scalar Types	A-3
A.5.2	Structured Types	A-4
A.5.2.1	Vector Types	A-4
A.5.2.2	String Types	A-4
A.5.2.3	File Types	A-5
A.6	Declarations and Denotations of Variables	A-6
A.6.1	Entire Variables	A-7
A.6.2	Component Variables	A-7
A.6.2.1	Indexed Variables	A-7
A.6.2.2	Substrings	A-8
A.6.2.3	String Cursors	A-8
A.6.2.4	Current File Components	A-9
A.7	Expressions	A-9
A.7.1	Operators	A-10
A.7.1.1	Operators \neg and <u>SIZE</u>	A-10
A.7.1.2	Multiplying Operators	A-10
A.7.1.3	Adding Operators	A-11
A.7.1.4	Relational Operators	A-11
A.7.2	Function Designators	A-13
A.8	Statements	A-14
A.8.1	Simple Statements	A-14
A.8.1.1	Assignment Statements	A-14
A.8.1.2	Append Statements	A-16
A.8.1.3	Insertion Statements	A-17
A.8.1.4	Escape Statements	A-17
A.8.1.5	Procedure Statements	A-18
A.8.1.6	Empty Statements	A-19
A.8.2	Structured Statements	A-19
A.8.2.1	Compound Statements	A-20
A.8.2.2	Conditional Statements	A-20

A.8.2.2.1 If-Statements	A-20
A.8.2.2.2 Case-Statements	A-21
A.8.2.3 Loop Statements	A-21
A.9 Procedure Declarations	A-22
A.9.1 Standard Procedures	A-25
A.10 Function Declarations	A-25
A.10.1 Standard Functions	A-27
A.11 Pattern Declarations	A-27
A.12 Incremental Sections	A-30
A.12.1 Define-Statements	A-31
A.12.2 Take-Statements	A-32
A.13 Programs	A-34
A.14 Examples	A-35

APPENDIX B SNIP ABSTRACT MACHINE (SAM)

B.1 SAM Order Codes	B-1
B.1.1 Monadic Load Group (ML)	B-2
B.1.2 Subscripting Group (SC)	B-3
B.1.3 Load and Operate Group (LO)	B-4
B.1.4 Store Group (ST)	B-9
B.1.5 String Operation Initialisation Group (SOI)	B-12
B.1.6 Load-and-Store String Group (LSS)	B-13
B.1.7 Jump Group (J)	B-18
B.1.8 Block and Pattern Control Group (BPC)	B-21
B.1.9 Procedure Control Group (PC)	B-23
B.1.10 Parameter Passing Group (PP)	B-26
B.1.11 Library	B-29
B.2 Summary of Instruction Mnemonics and Parameters	B-30

APPENDIX C LL(1) GRAMMARS C-1

APPENDIX D SNIP TRANSITION DIAGRAMS D-1

APPENDIX E	SIMPLE TRANSLATION EXAMPLE	E-1
APPENDIX F	EXAMPLES OF SNIP SELF-EXTENSION	F-1
APPENDIX G	CHARACTERISTIC ERRORS IN PROGRAMMING LANGUAGES	G-1
APPENDIX H	CONFLICTS OF BASE LANGUAGE DESIGN CRITERIA	H-1

ABSTRACT

The basic premise of this thesis is that extensible languages afford the user considerable power and flexibility. We argue that this flexibility can, and should, be provided in a secure and error-resistant manner, but that this objective is not realised in existing extensible languages.

This thesis first investigates the nature of security in programming languages, building up a simple and informal theory of the design of secure languages, and relating this theory to the notions of structured programming and transparency.

We use this theory to build a conceptual model for a secure extensible language and its physical realisation. We show that existing extensible languages fail to meet the ideals of this model in total, and proceed to design an alternative and secure system which builds upon, but attempts to avoid the pitfalls of existing systems. We base this system on a string processing language (Snip) which is itself extensible. The remainder of this thesis discusses the design and implementation (based on an abstract machine, SAM) of this language.

INTRODUCTION AND HISTORICAL BACKGROUND

0.0 Introduction

In the last two decades, there have emerged several hundred so-called high-level programming languages. The term "high-level" (as opposed to low-level or machine-oriented) is intended to imply that the expressive powers of the language make it a valuable communication medium from the standpoint of a computer user rather than from the standpoint of a computer itself. There is thus an implicit assumption that there is a certain degree of mismatch between user-oriented and machine-oriented (specification of) algorithms (Els 73).

In contrast to the early days of programming, the costs of building hardware have fallen relative to the cost of programming effort; at the same time there has emerged a greater appreciation of the complexity of the task of programming. Correspondingly, therefore, there has been a growing tendency to place the burden of man-machine communication on the machine rather than on the human programmer: hence the interest in high-level languages.

High level programming languages are thus constructed with the aim of protecting the user from unnecessary detail in the realisation of algorithms on a given machine, in the hope of allowing easier writing, understanding and modification of programs. A further aim (or more probably effect) which is less commonly remarked upon is that of reducing programming error. We

might therefore say that high level languages are more secure. Thus, with the development of truly user-oriented languages such as Algol 60, the user was protected from the details of hardware realisation by such innovations as block and control structure.

While most high level languages are formally universal in the sense that they are capable of expressing every function which may be "computed" by a Turing machine (Che 69), each language is, in practice, useful only for those functions which may be conveniently and efficiently expressed in terms of that language cf. (Per 67). It is towards fulfilment of these aims of notational convenience and efficiency that a proliferation of high-level languages has been built up, each oriented towards solution of different classes of problem. One of the drawbacks of this development has been the considerable task of implementing compilers for a multiplicity of programming languages on a wide variety of real machines, each with its own distinctive architecture. One possible means of tackling this problem - and the one towards which this research is directed - is through the design of extensible languages: the expectation is that the user can extend an existing language, thus protecting himself further from the details of hardware realisation, and that the number of distinct high level languages required will thereby be considerably diminished, thus reducing the size of the implementation problem. A further problem arises, however, that while the extensions defined to a language may provide more protection for the user, the extension mechanism itself may fail to provide

0-3

sufficient protection from realisation details i.e. this mechanism may be insecure. This idea which has received scant attention from designers of extensible languages forms the basis of this thesis.

In this initial chapter, we consider solutions to the problem of implementing compilers for the ever increasing number of high level programming languages; as noted, we will be particularly concerned with the solution offered by extensible languages. We point to the considerable flexibility afforded to the user of extensible languages, and discuss the advisability of such unrestricted licence. This leads us to the main issue of this thesis.

0.1 Historical Background

Having considered the origin of the high level programming language and its proliferation, we proceed to consider the various means devised for the economic implementation of large numbers of diverse languages on equally diverse real machines. Possible solutions are as follows:

- (a) The design of a single (or perhaps several) ideal and universal (or shell) language(s), capable of efficiently and conveniently expressing all algorithms.
- (b) The design of a compiler-compiler or translator writing system capable of generating a compiler from the language specification.
- (c) The design of portable translators which may be easily transported from one machine to another.

- (d) The design of extensible languages which may be tailored to the users' requirements.

We discuss these solutions further:-

(a) Universal or Shell Language

The idea of a shell language (in the sense that it contains constructs suitable for all users and all application areas) has long been considered (cf. (Hal 64; New 68)). It is generally accepted, however, that the search for such a language cannot succeed (Hal 68; New 68; Sch 70; Sol 74). A shell language must contain constructs oriented towards such diverse application areas as software writing, numerical analysis, artificial intelligence, commerce etc. The translator for a shell language is therefore almost inevitably large and slow, and consequently hard to implement and maintain. We must expect that programs will frequently incur overheads of constructs they do not require and also of constructs which are unnecessarily flexible for their particular problem area. Perhaps the most serious difficulty, from the point of view of the user, is, however, encountered when a single notation is used to convey distinct meanings in different application areas (Che 68; New 68; Sol 74).

PL/1 for example, as far as universality is concerned, has no facilities for pattern matching or defining co-routines. Several anomalies arise because all users are bound to a single interpretation of the meaning of constructs even if the interpretation contradicts well-established usage. We consider two examples:

Example 0-1

Both logical expressions " $5 < 6 < 7$ " and " $7 < 6 < 5$ " are true in PL/1. This situation arises as conditional expressions have a bitstring value. Thus " $(7 < 6) < 5$ " evaluates as " $1'B < 5$ " and hence " $1 < 5$ ".

Example 0-2

The interpretation of the matrix product " $A \times B$ " in PL/1 is the matrix whose ij^{th} element is the product of the ij^{th} elements of A and B. (Che 68)

That it is impossible to foresee all the applications of a language and the demands upon it is shown clearly by the development of major programming languages; indeed, even were this possible, it is probably unfeasible in practice to cater for all possible applications.

(b) The Compiler-Compiler or Translator Writing System

The aim of translator writing systems is to automate the generation of compilers, given the formal definition of syntax and semantics of a language. A fairly comprehensive survey of such systems is to be found in a paper by Feldman and Gries (Fel 68): development of translator writing systems is traced from the classic Brooker-Morris compiler-compiler (Broo 63). While formal description of the syntax of programming languages has been achieved with reasonable success, there has been less agreement on how to describe semantics or to associate a meaning with constructs: the compiler writer is forced to hand-code semantic functions by specifying the

generation of appropriate target code.

(c) Portable Translators

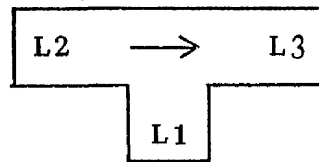
We say that a program is portable if it is relatively easy and straight-forward to move it from one machine to another (Poo 73); if the effort to move the program is considerably less than the effort required to write it initially, then we say that the program is highly portable. We consider briefly two techniques of transferring a compiler from one machine to another (Poo 74).

Portability through High Level Language Coding

Suppose we have a compiler for language A written in terms of high level language B which runs on machine X and produces machine code for that machine. Suppose also that we wish to transfer the compiler for A to machine Y. If the compiler for B is available on machine Y, then the transfer is quite straight forward, provided the compilers are compatible; the compiler still produces code for machine X and code generation routines therefore have to be re-written.

The more usual situation is one in which there is no compiler for language B on machine Y, and the implementor is faced with the task of first bootstrapping the compiler for B on to the new machine Y. In this case, languages A and B may be identical (i.e. the compiler for A may be written in terms of A itself) and A may be bootstrapped as follows: The compiler for A written in A is modified to produce assembly code for Y. This modified compiler is then compiled by the original

compiler running on machine X. The resulting compiler produces code for machine Y, but runs on machine X. The process is therefore repeated to produce a compiler which runs on Y. The process of bootstrapping is much more readily understood using the T-diagram notation (Ear 70). A translator written in language L1 to translate source language L2 to target language L3 is represented as



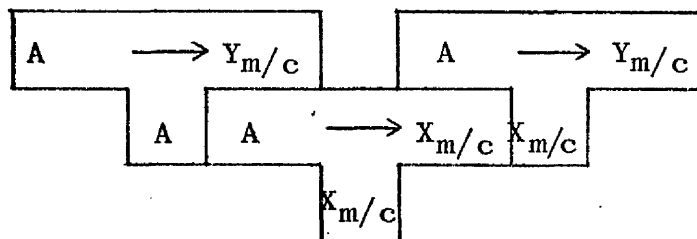
The bootstrapping process described is illustrated in figure 0-1.

An alternative method of bootstrapping is to code a compiler for a simple version of A, A_1 (say). A more advanced version of A, A_2 (say) is then coded in A_1 and compiled by the first compiler. This process can be repeated many times.

Portability through abstract machine modelling

An alternative means of achieving portability is through the design of an abstract machine well-suited to modelling the data structures and operations of the language to be transported.

A translator is written to compile programs in language A to equivalent programs for the abstract machine. Portability is achieved by first implementing the abstract machine on the receptor machine and subsequently



Key

A language A

$Y_{m/c}$ assembly code for machine Y

$X_{m/c}$ assembly code for machine X

FIGURE 0-1

bootstrapping the translator using the abstract machine.

The classical approach to this problem was the attempt to design a "universal" abstract machine capable of efficient realisation on all real machines, and a suitable target language for all high level languages (cf. UNCOL (Stee 61)). One of the main reasons that this has not been put into practice is the difficulty in specifying a suitable such machine (Poo 74).

We note the similarity between the idea of abstract machine modelling and the second form of bootstrapping discussed above.

We extend these ideas of abstract machine modelling in chapter 4.

(d) Extensible Languages

An alternative approach is that of the extensible language which may be extended and perhaps modified and adapted to suit the needs of individual users or groups of users. The expectation is not that a single extensible language is a panacea for all these problems, but that a group of extensible languages may reduce the number of distinct special purpose languages required. We do not therefore regard portability and extensibility as rival solutions, but rather as complementary techniques, both of which provide partial solutions to the problem of programming language proliferation.

While languages such as Algol 68 and Snobol 4 are not normally considered as extensible, they do however include some of the features of extensible languages, namely the introduction of new operators and data types.

0-9

We find it convenient to regard these languages as restricted forms of extensible language rather than introduce a new category to describe them.

0.2 Specific Problems and Goals of this Thesis

Having considered the historical background, we proceed now to consider, in particular, the area of investigation of this thesis, namely extensible languages. Extensible languages, by definition, allow the user considerable flexibility and power to manipulate the syntax and semantics of the language concerned. In some cases, considerable knowledge of the language syntax, translator architecture and machine code as well as great ingenuity and skill is involved in introducing extensions.

We feel intuitively, that the greater the freedom allowed in an extensible language system, the greater the complexity and the greater the opportunity for, and likelihood of creating meaningless programs (cf. Solntseff Sol 74). Feldman and Gries (Fel 68), for example, point to the possible disastrous effects of misuse of Cheatham's macros; they suspect also that the sensitivity of Galler and Perlis' scheme to programming error will seriously restrict the applicability of that system. Irons (Iro 70) and Cheatham (Che 66) point to the danger of introducing ambiguity in extensible languages.

If such flexibility is indeed necessary, then there is perhaps little we can do about this alleged sensitivity to programming error; however, if, as we suspect, this flexibility can be circumscribed without compromising the flexibility genuinely required by the

("average") user, then we can perhaps improve the situation greatly.

It is principally towards some solution of this problem, which has been largely neglected by designers of extensible languages, that this research is directed. Essentially, we are considering one aspect of software reliability, a property which assumes increasing importance with the ever-widening application of computers, while the consequences (both social and economic) of failure to provide it become more serious.

0.3 Outline of Thesis

In chapter 1 we review the current status of research into extensible languages.

In chapter 2 we consider first the design of secure programming languages i.e. languages which are resistant to error. We widen this notion to include extensible programming languages, and develop a (logical) model for a secure extensible language. We consider how existing systems measure up to this model and find that they fall far short of it. We proceed therefore to design a more secure system, building upon the more secure aspects of existing systems. We argue that a string processing language is a suitable means of realising this extensible language system; the remaining chapters of the thesis are concerned with this realisation.

In chapter 3 we design a string processing language suited to our purposes and capable of implementing a secure extensible language mechanism.

In chapter 4, we consider how implementation might

be achieved. We design an abstract machine for this purpose, but touch only briefly on portability.

Finally, in chapter 5, we review and criticise the first four chapters. We consider the viability of the system proposed and discuss how far our objective of designing a secure extensible language has been achieved.

We will frequently refer in the text to the programming languages Algol 60 (Nau 62), Algol W (Bau 71), Algol 68 (Wij 68) and Pascal (Wir 70). To avoid excessive repetition, we will not normally repeat the references to these languages.

CHAPTER 1

EXPOSITION ON EXTENSIBLE LANGUAGES

1.0 Introduction

Before proceeding to the body of this thesis, we find it expedient to review the current conception of extensible languages and associated terminology. We introduce also a classification scheme and diagrammatic representation used in later chapters of this thesis. Included also is a brief survey of existing extensible languages.

1.1 Extensible Languages

Informally, we regard a language as extensible when the user has the capability of extending and perhaps altering the meaning of existing constructs in that language. It has also been suggested (Sch 70; Sol 74) that the definition ought also to include contraction to allow exclusion of unnecessary constructs and thus avoid needless overheads.

According to this definition, every language is then extensible in the sense that the user can modify its compiler. Indeed, Scowen (Sco 71) has proposed and implemented a compiler for the language Babel with precisely this aim in mind. We propose, however, to exclude such systems from our definition and to insist that extensions be introduced in a manner which affords some degree of independence from the translator architecture.

According to the original conception, an extensible

language is composed of two essential components (Sch 70):

- (1) a base or core language comprising a set of indispensable primitive constructs,
- and (2) a set of extension mechanisms establishing a framework for defining new linguistic constructions in terms of already existing ones.

Typically, definitions are pyramided using a particular version of the extended language as the new base language.

Two advantages accrue from this organisation: firstly, the problems of portability and standardisation are reduced, since once the core and extension mechanism have been transported, it is a simple matter to implement any extended version of the language. Secondly, the difficulty of specifying the semantics of the language is eased since extensions are defined in terms of base constructs or existing extensions (Lea 66).

This idea of extensibility evolved, or, perhaps more accurately, there was an alternative school of thought: Extensible languages should permit new language abstractions to be introduced not only to allow convenient programming of a particular sequence of actions, but also to allow efficient specification of particular sets of action which could not previously be programmed efficiently. (This is achieved by allowing generation of sequences of machine code which were not possible before (Gal 74)). These two schools of thought have led to the loosely defined terminology of syntactic and semantic forms of extension. We make the following informal definitions:

DEFINITION 1-1

We say that an extension mechanism allows syntactic

extensibility if the user has the ability to introduce explicit modifications to the syntax of a language.

DEFINITION 1-2

We say that an extension mechanism allows semantic extensibility if the user has the ability to associate new meanings with the constructs of the language (with the implication that the extended language is capable of actions which were not efficiently expressible in terms of the original base language).

Galler (Gal 74) observes that both forms of extension should be available.

The system as originally conceived is capable only of syntactic extension, since extensions are defined in terms of the "current" base language. The simplest means of handling semantic extension is to introduce a mechanism similar to that in the compiler-compiler or translator writing system. Unfortunately, the user has to have a considerable knowledge of the target language and of the translator architecture in order to define extensions by this means (Sol 74). This is clearly none too satisfactory, and as a result, the development of extensible languages has taken on the character of an attempt to isolate and generalise various components (cf. below) of programming languages with the object of introducing systematic variability in a less machine-dependent and translator-dependent manner (Sch 70). A consequence of this effort has been the gradual emergence of a more abstract view of extensible languages in which the base language is construed as a set of essential primitives,

1-4

minimally organised by the syntax into a coherent language. Semantic extension is considered as a set of constructors serving to generate new, but completely compatible primitives; syntactic extension permits the definition of specific structural combinations of these primitives. Thus, extensible languages have progressively assumed the aspect of a language definition framework which has the unique property that an operational compiler exists at each point in the definitional process (Iro 70; Sch 70).

We consider the programming language components which have been, to some extent isolated:

syntax

operators

data structures

control structures

We choose to include syntax in this group, since syntactic extensions could be handled by compiler-compiler methods, but do not consider this further here.

Data Structures

Data structure extensions cannot easily be defined by simple syntactic extension or by compiler-compiler methods because of the difficulty of handling context-sensitive syntax (e.g. organising type checking through identifier table processing) and context-sensitive semantics (e.g. delay assignment until expression has been evaluated, storage allocation, addressing functions).

Several methods of defining data structures in terms of machine-independent and translator-independent abstractions have been devised. Perhaps the best known are those of Standish (Sta 69) and Garwick (Gar 68). The

1-5

type, addressing function and field designator are implicitly defined for each data structure introduced. Cheatham's scheme (Che 66) allows the user to define his own addressing function. Iron's scheme (Iro 70) provides no type checking.

Jorrand (Jor 71) has observed that this is essentially a syntactic form of extension, since extensions are defined in terms of a fixed set of primitives and constructors. He proposes a system to allow semantic extension.

More recently, Liskov (Lis 74) has introduced the notion of clusters in which new operators are defined together with the associated data structure extensions.

Operators

Operator extensions are easily handled by a simple syntactic extension scheme, but for the difficulty of specifying context-sensitive syntax (cf. type checking). Most systems therefore introduce a special construct to allow specification of type checking without the need to know the translator architecture (cf. GPL Gar 68).

Control Structures

This is the most recent area in which extensions have been proposed. Little has been done in this area apart from the initial proposals by Bagley (Sol 74) and the work of Fisher (Fisd 70), many of whose ideas have been incorporated in PPL (Sta 69).

We may view isolation of these programming language components as an attempt to provide an easier method of extending these particular components. Inevit-

ably, however, in doing so, we circumscribe the range of extensions which may be defined (compare the relation between a high level programming language and an assembly language).

In this thesis we will not be particularly concerned with the question of whether or not a particular mechanism supports syntactic or semantic extension; we shall be more interested in the kinds or range of extensions that can be defined or the means of defining semantics of extensions. We therefore introduce an alternative classification scheme in the following sections. Before doing so, however, we introduce some terminology:

1.2 Terminology

For the sake of uniformity, we make use of the terminology defined by Solntseff and Yezerski, where appropriate.

DEFINITION 1-3

We refer to program text expressed entirely in terms of base language constructs as base text.

DEFINITION 1-4

We refer to program text expressed entirely in terms of extension constructs as augment text.

DEFINITION 1-5

We refer to program text expressed in terms of extension or base language constructs (i.e. a combination of base and augment texts) as extended text.

DEFINITION 1-6

We refer to the text produced as the result of processing

or "expansion" of the augment text as the derived text.

We will refer to the metalanguage used to define extensions simply as the metalanguage (except in those cases where ambiguity might arise).

DEFINITION 1-7

We refer to the language in terms of which (the semantics of) extensions are defined as the semantic base. (This term is particularly useful when the semantic base is the base language or extended language itself).

DEFINITION 1-8,9 Binding, Binding Time

Binding (cf. Ibr 74; Weg 68; Els 73) is a process of association and specification as a result of this association. For example, in declaration of an extension, we include specification of the meaning of the extension in terms of derived text. The association of the declaration with the use of the extension in some extended text specifies the action to be taken to produce appropriate derived text. This form of binding occurs during compile-time. The phenomenon of binding time refers to actions rather than orders. An action is defined as a process (transfer of information, operations on operands yielding results) which occurs at run-time. Orders (instructions) are requests for actions and are the units in which translators work; they are produced at translate time.

We distinguish two forms of binding, a static binding and a dynamic binding. In static binding, we are primarily concerned with the translator, in dynamic

1-5

binding with run-time interpretation. In this dissertation we shall be concerned only with static binding (time) which we shall refer to in later text simply as binding (time).

Static binding time is the time (or stage in the translation process) at which the order to be obeyed at run-time is specified or made more specific. In order to make this notion of time as unambiguous as possible, the term "time" is interpreted as widely as possible. Thus, the translation process is conceptually viewed as taking a large number of steps, viewed in turn as occurring at different times. In practice (for example in a one or two pass translator) most of these times will be chronologically the same time.

DEFINITION 1-10

We refer to user-defined statements (associated with an extension) which are executed at binding time as extension-time statements (cf. macro-time statements).

1.3 Classification Scheme and Diagrammatic Representation

The classification scheme reviewed in this section was developed by Solntseff and Yezerski (Sol 74) while the diagrammatic representation was devised by this author in attempt to clarify the classification scheme to himself.

Solntseff and Yezerski have noted the growing number of extensible languages and the extremely varied means of achieving extensibility in implementation. They found it necessary to develop a classification scheme in order to handle comparisons of different systems. This classification is in fact a generalisation of the scheme

proposed by Cheatham (Che 66) for classifying macro facilities. Extension mechanisms are grouped on the basis of the stage of the language-translation process during which augment text is processed. From definition 1-9, we see that this is equivalent to classification according to our notion of binding time.

Six stages of the language translation process are considered:

- (1) lexical analysis,
- (2) syntax analysis,
- (3) production of parse tree or of some other form of intermediate-language text,
- (4) analysis of the intermediate-language text prior to code generation,
- (5) generation of real-machine or abstract-machine code,
- (6) conversion of generated code to a form suitable for direct interpretation by a real machine.

As noted in definition 1-9, we interpret the notion of time as widely as possible to avoid ambiguity, although, in practice, for a given translator, many of these times will be chronologically the same time and the stages therefore improperly distinguished.

We thus distinguish six classes of extension mechanism, one corresponding to each language-translation process:

(1) Type A or Text Macro Extension

In type A extension, the augment text is converted into base text during the lexical analysis stage. The derived language is the base language.

(2) Type B or Syntax Macro Extension

Conversion of the augment text to base text occurs during syntactic analysis of the source text, but before generation of intermediate text or construction of a parse-tree. As in the previous case, the derived text is the base language.

(3) Type C or Intermediate-Language-Generation Extension

The augment and base texts are translated in parallel into texts in the same intermediate language. The extension mechanism operates during the generation of the intermediate language text (or parse tree).

(4) Type D or Intermediate Language Extension

Both augment and base texts are fully analysed, the augment text being converted to text in an "extended" intermediate language. Thus, conversion into a homogeneous text in the standard intermediate language involves manipulation of intermediate language texts.

(5) Type E or Code-Generation Extension

The augment text is converted in parallel with the base text into the same real or abstract machine code at code-generation time.

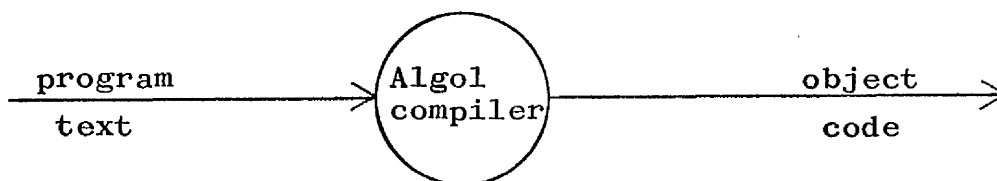
(6) Type F or Load-Time Extension

The augment is converted into a machine code which either requires further processing before it can be interpreted by the real or abstract machine or requires that the set of states of the real or abstract machine be extended by the statements of the metalanguage.

Diagrammatic Representation

We find it useful to indicate precisely how extensions are processed, according to the classification of the extensible language and we therefore introduce the following directed graph representation. It bears some resemblance to the computation graphs (used to represent parallel processing):

Processes or stages of translation are represented by nodes of the graph. Directed edges represent flow of control. Edges may be labelled by data. Thus, for example, an algol compiler might be represented:

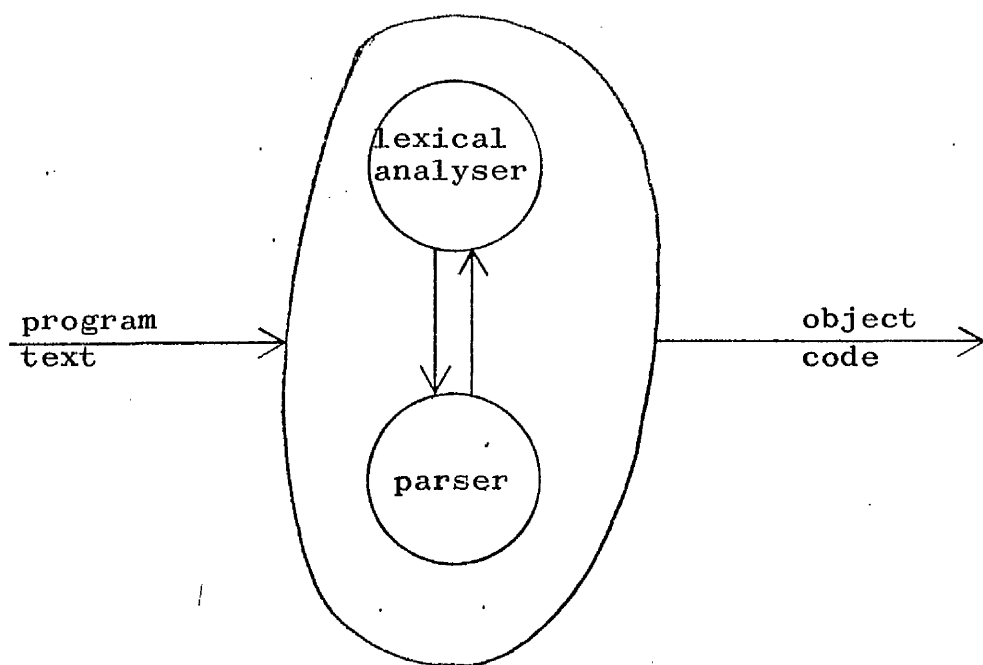


If a node has more than one exit, then the particular exit chosen is uniquely determined by the data.

We find it useful to provide a means of indicating that the precise relationship between two or more nodes is either variable or unspecified. We illustrate this notation by an example:

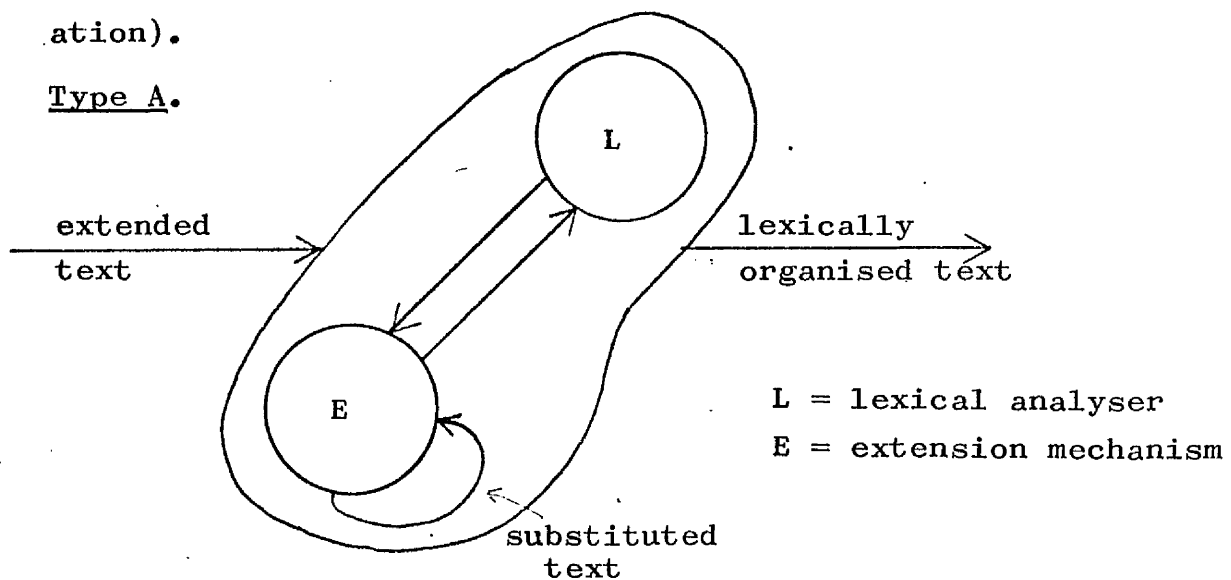
Example 1-1

We consider a translator which we assume to consist of two processes, a lexical analyser and a parsing and code generating stage. We do not know whether source text is completely scanned before parsing or whether the parser calls the lexical analyser to obtain the next text item. We represent this:

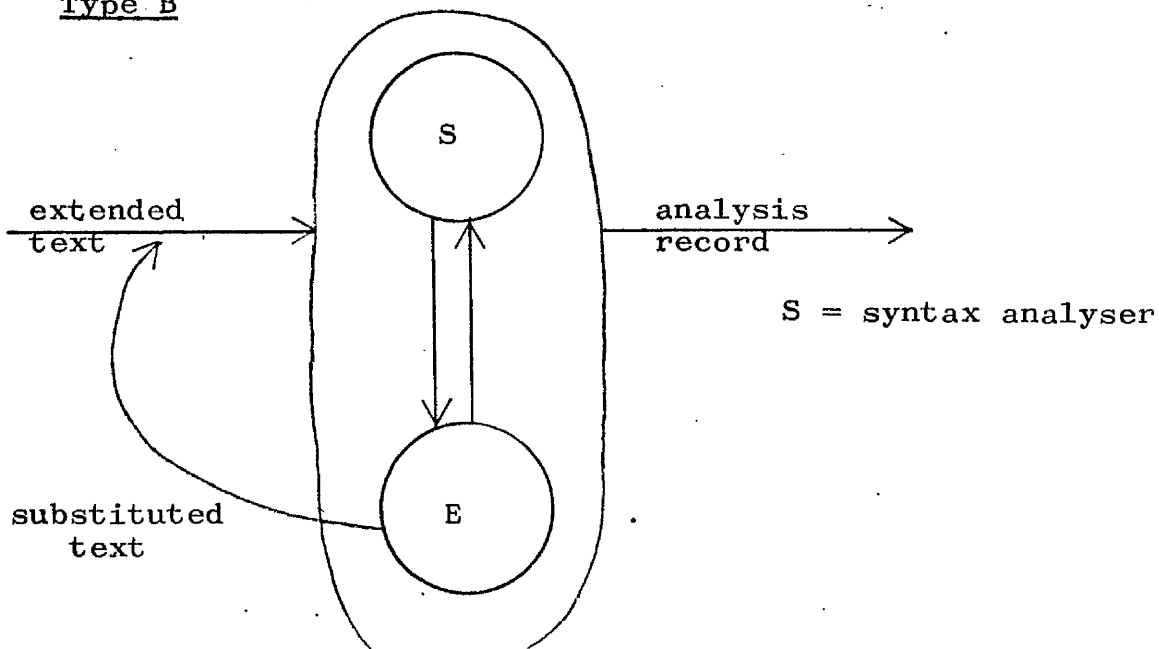


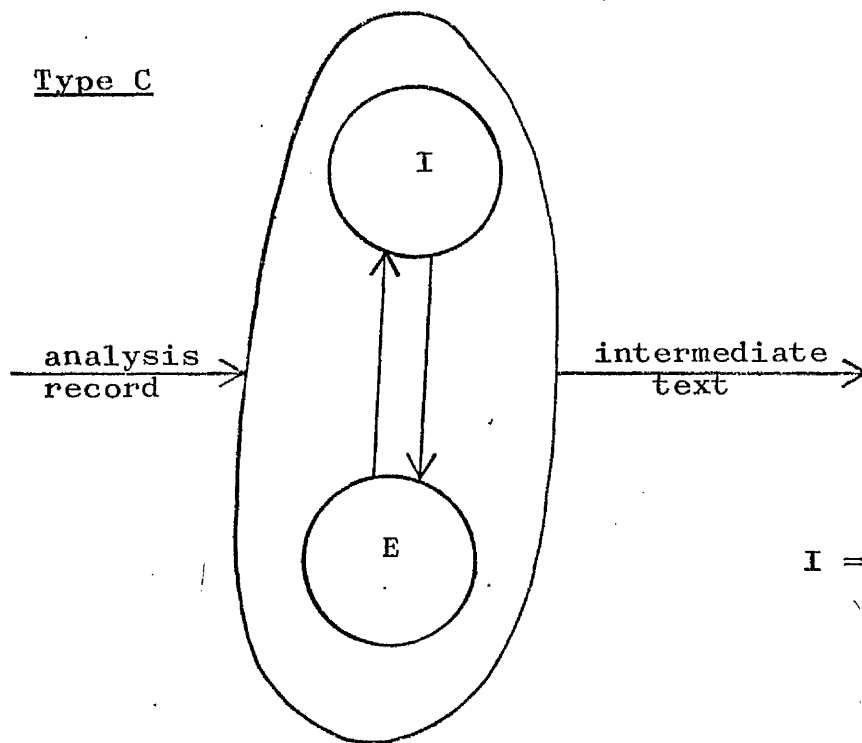
We now consider representation of the various extensible language schemes (according to the most common implementation).

Type A.

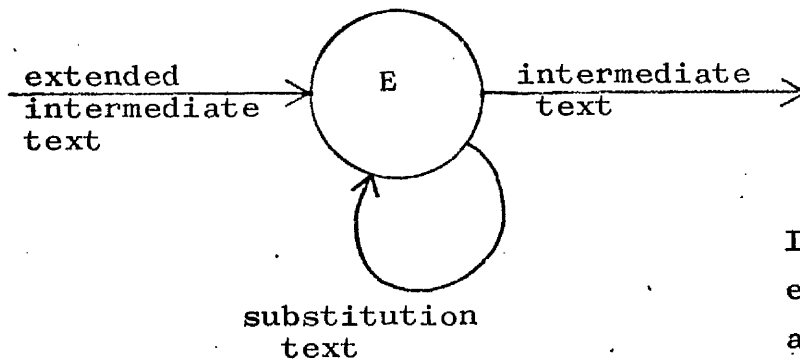


Type B

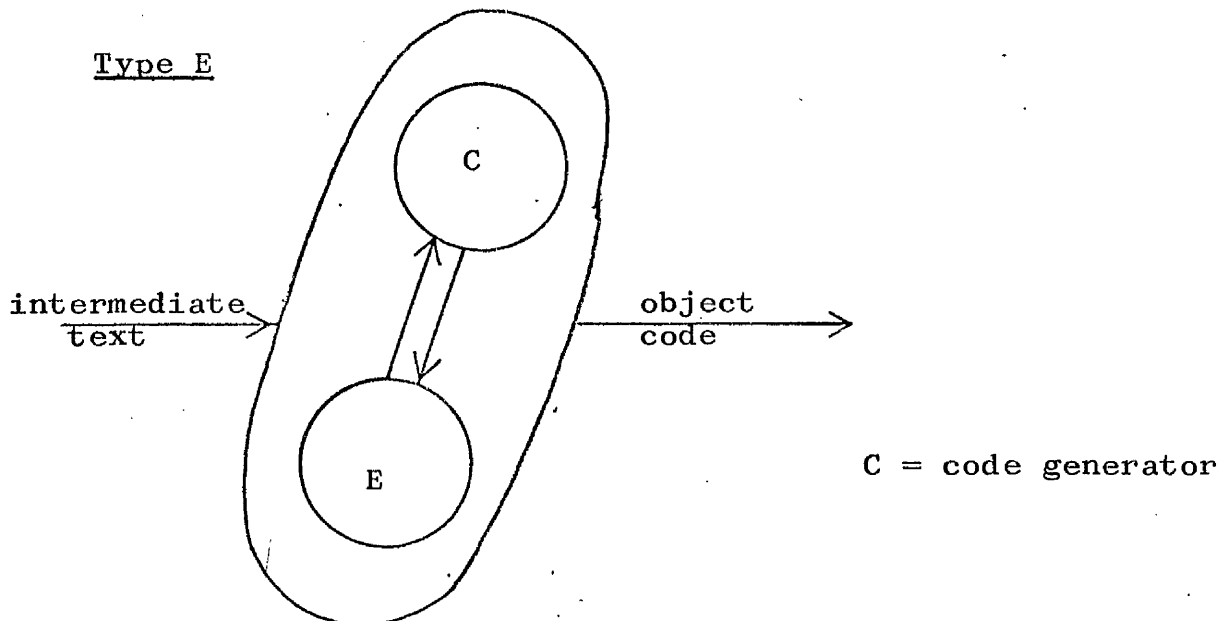


Type C

I = intermediate-
text generator

Type D

In this case, the
extension mechanism E
analyses intermediate-
language text prior to
code generation.

Type E1.4 Brief Survey of Existing Extensible Languages

In this section, we survey briefly the principal or representative languages in each class of extension mechanism. Much of this material is included in the paper by Solntseff and Yezeriski (Sol 74).

(1) Type A or Text Macro Extension

In text macro extension, the augment text is converted into base text during the lexical analysis stage. The derived language is the base language. This form of extension is commonly known as text-macro extension.

McIlroy (McI 60) realised that macro techniques could be applied to the extension of high level languages in the same way as they are used in assembly languages. He observed also that a powerful language is required for the specification of extension-time processes. Most of his ideas have been embodied in subsequently designed macro processors such as GPM (Str 65) and TRAC (Moo 66). These systems define simple text macros which can be recognised with little or no syntactic analysis of the source text.

Systems such as ML/1 (Bro 67), Stage 2 (Wai 73) and Cheatham's (Che 66) MACRO facility allow more flexibility in the form or template of the macro call. Cheatham's MACRO facility does not allow conditional expansion (i.e. allows only pure substitution macros).

Macro systems such as LIMP (Wai 67) and that devised by Grant (Gra 71) allow greater flexibility in the form of macro call by using Snobol 4 patterns to describe this form. McAlgol (Bow 71) allows a similar capability, but leaves recognition entirely in the user's hands. XPOP (Hal 64) includes similar features, but is oriented towards assembly languages.

Example 1-2

With the metalanguage definition

MACRO A MEANS "+B*C+",

the extended text

X = Y %A Z

will give rise to the derived text

X = Y + B * C + Z

Type B or Syntax Macro Extension

Conversion of the augment text to base text occurs during syntactic analysis of the source text, but before generation of intermediate text or construction of a parse-tree. As in the previous case, the derived text is the base language. This type of extension is commonly known as "syntactic macro extension". The syntax of the entire source text is analysed so that in defining the syntax of extensions, the programmer can make use of the syntactic classes used to describe the base language.

Cheatham's SMACRO (Che 66) conforms to this scheme. Leavenworth (Lea 66) provides a similar scheme which allows, in addition, conditional expansion. However, it restricts extensions to the syntactic classes statement and expression. Snap (Fisr 73) and PPL (Sta 69) provide similar schemes.

Example 1-3

We might define a for-statement:

```
STATEMENT MACRO  FOR <V:variable> := <E1:expression>
                  {1 WHILE <E2:expression>} | [2 BY
                  <E3:expression>]
                  {3 TO <E4:expression>} DO <S:statement>
```

MEANS

```
BEGIN LOCAL L1, L2;
    L1 : V := E1 ;
    L2 : IF { 1 E2 THEN BEGIN S ; GOTO L1 }
        { 3 V ≤ E4 THEN BEGIN S ;
          V := V + { 2 E3 } {¬2 1} GOTO L2 }
        END
END;
```

Where "[]" denotes that the enclosed element is optional and "|" denotes alternation. The pair brackets "< >" denote formal or substitution parameters, while the brackets "{n }", where n is an integer, in the macro body indicate that this part of the macro body is to be scanned if and only if there is a corresponding section in the current extended text.

Type C or Intermediate-Language-Generation Extension

In Type C extension, the augment and base texts are translated in parallel into texts in the same intermediate language. The extension mechanism operates during the generation of the intermediate language text (or parse tree).

Examples of type C extensible languages include Proteus (Bel 69), Balm (Chr 69), GPL (Gar 68) and EPS (Chr 69). All allow some form of extension-time capability.

Example 1-4

We might define a new data type, complex: COMPLEX {REAL RP, IP}. If we declare A : COMPLEX, then we may refer to RP(A) and IP(A).

We might then define complex addition:

OPERATOR A + B PRIORITY + ;

{IFF COMPLEX A TAKE

{IFF COMPLEX B TAKE COMPLEX TO BE

COMPLEX (RP(A) + RP(B), IP(A) + IP(B))

IFF REAL B \vee IFF INTEGER B TAKE COMPLEX TO BE

COMPLEX (RP(A) + B, IP(A)) }

IFF COMPLEX B \wedge (IFF REAL A \vee IFF INTEGER A) TAKE

MACRO B + A }

The construction IFF COMPLEX tests the type of variable A and returns a boolean value; IFF ... TAKE ... forms an extension-time conditional in which the consequent is compiled as a closed subroutine, unless the symbol following TAKE is MACRO, in which case the call on the operator is

replaced with in-line coding.

Type D or Intermediate Language Extension

Both augment and base texts are fully analysed, the augment text being converted into text in an "extended" intermediate language. Thus, conversion into a homogeneous text in the standard intermediate language involves manipulation of intermediate-language texts.

In existing systems, extensions are defined in terms of the base language and the user is therefore shielded from details of target code or of target-code generation. Cheatham's computational macros (Che 66) are of type D. They have been extended and refined by designers of other type D mechanisms. The scheme by Galler and Perlis (Gal 67) is specifically oriented towards extension of data structures and operations. The form of data structure extension is fairly limited, relative to other systems, as only an array-constructor is provided. However, the designers were particularly concerned with the generation of efficient code for operations on large data structures (matrix operations, for example). Standish (Sta 69) provides PPL with a much larger number of constructors for data structure extensions.

The Imp system (Iro 70) allows even greater flexibility: the user may define at which pass of intermediate text a particular extension is to be bound.

The actual definition of extensions of this type is frequently similar to the definition of syntax macro extensions (cf. example 1-3).

Type E or Code-Generation Extension

The augment text is converted in parallel with the base text into the same real or abstract machine code, at code generation time.

Systems in this category correspond quite closely to the compiler-compiler notion. The MAD system (Ard 69) was the first system of this kind; it is characterised by a translator with fixed structures, but variable tables. In the ECT system (Sol 74) the structure of the translator itself is altered: it might be better regarded as a tool for producing translators rather than an extensible language as such. Snap (Fisr 73) and Lace (New 68) are also Type E systems.

Example 1-5

We might introduce the statement "INC X [I,J,K] BY 1" to replace "X [I,J,K] := X [I,J,K] + 1" as follows:

```
NEW STATEMENT FORM ("INC <variable> BY <expression> ",
                    "load address (XR1, <2>), * fetch
                    indirect (XR1, <4> ),
                    * apply (* plus, * type (<2>), * type
                    ( <4> )),
                    *store indirect (XR1)").
```

The first string specifies the syntax of the new form and the second the sequence of extension-time instructions which will yield the required target code. An asterisk indicates that the function which follows is a system function. The construction <n> , n integer, denotes the actual parameter corresponding to the n-th syntactic element in the syntax string.

Type F or Load-Time Extension

The augment text is converted into a machine code which either requires further processing before it can be interpreted by the real/abstract machine or requires that the set of states of the real or abstract machine be extended by statements of the metalanguage. No systems of this form have been implemented.

Figure 1-1 summarises the classification of the principal or representative systems according to the scheme described above. It will be noted that some systems allow a choice of binding time.

Classification Language	Text macro Type A	Syntax Macro Type B	Intermediate- Language- Generation Type C	Intermediate Language Type D	Code- Generation Type E
GPM (Strachey)	x				
Macro (McIlroy)	x				
PL/1 compile-time facility	x				
Stage 2 (Waite)	x				
TRAC (Mooers)	x				
XPOP (Halpern)					
Limp (Waite)	x				
McAlgol (Bowlden)	x				
ML/1 (Brown)	x				
Syntax macro (Grant)	x				
Macros (Cheatham)	x	x		x	
Syntax macro (Leavenworth)		x			
Balm (XLISP) (Harrison)			x		
CEL (Spitzen)			x		
EPS (McLaren)			x		
GPL (Garwick)			x		
Proteus (Bell)			x		
AEPL (Milgrom)				x	
Algol C (Galler)				x	
Imp (Irons)				x	
PPL (Standish)		x		x	x
ECT (Solntseff)					x
Lace (Newey)					x
Mad (Arden)					x
Snap (Fisher)		x			x

FIGURE 1-1 Classification of Principal Extensible Languages.

CHAPTER 2

DESIGN CONSIDERATIONS FOR EXTENSIBLE LANGUAGES

2.0 Introduction and Design Criteria

In the previous chapter, we reviewed existing extensible languages and pointed to some of their failings. In particular, we noted that extensible languages are more "error-prone" than they should be.

In this chapter, we attempt to form a better understanding of this intuitive notion of error-proneness. We show why we believe this to be an important design criterion, illustrate some ways in which existing systems fail to meet this criterion, and finally attempt to design a system which is less prone to error or more secure.

The goals in designing an extensible language system should be to provide a system that is

- (a) fast and compact
- (b) capable of producing good object code
- (c) not excessively error-prone i.e. relatively secure.

Since these goals are not mutually independent, it is not possible to achieve them all optimally. In this thesis, we have chosen to give added weight to goal (c), the goal of security. We justify this decision in the following sections: we first pursue the notion of security in programming languages in general, subsequently consider why this is important for extensible languages, and finally consider the implications for extensible language systems.

2.1 The Design of Secure Programming Languages2.1.0 Introduction and Overview

In this section, we develop an informal theory

concerning one particular aspect of program reliability. The reliability of a particular program is some function of the number of failures and the consequences of those failures. Study of reliability thus subsumes such topics as:

- (a) proof of correctness,
- (b) robustness and recoverability,
- (c) debugging,
- (d) program structuring for reliability,
- (f) language design for program reliability.

It is this last aspect with which we are concerned in this thesis and which we have chosen to call language security.

Previous work in this area has consisted essentially of identifying error-prone constructs and making intuitive and intelligent guesses about how to improve individual cases. More recently, however, Gannon (Gan 75) has drawn up a list of language design decisions which he expects to influence program reliability.

In order to apply ideas of security to the design of extensible languages, we need a theory which is able to predict, more precisely, the effect of design decisions on program reliability. In this section, therefore, we attempt to develop a theory of this kind. In so doing, we make the assumption that a programmer is capable of correctly selecting appropriate constructs from a programming language (of "manageable" proportions cf. Algol 60, Pascal) in which the appropriate use of each construct is well-defined. We argue also that we cannot always protect the user from himself, and thus, there comes a point when security must depend on the goodwill and

self-discipline of the programmer, himself.

2.1.1 Notion of Security

That the reliability of software is affected by language design is adequately demonstrated by studies of "characteristic errors" of programming languages cf. (Ich 74; Gan 75).

DEFINITION 2-1

We define the security of a programming language to be some function of the probability of prevention or (at worst) detection of programming error. Some measure of the consequence of error and the difficulty of debugging might also be included.

We would expect that any measure of the security of a particular language will be influenced by the skill and experience of the programmer(s), the complexity of the problem environment and the suitability of the language to this environment. It is perhaps more meaningful therefore to consider how the security of a particular language might be increased.

DEFINITION 2-2

We therefore define a secure language as one in which error is prevented as far as possible and in which failure to detect error which does occur is "acceptably rare"; in short, a language which is resistant to error.

Figure 2-1 illustrates the common situation in which certain errors introduced into a program result in illegal programs (mapping T_S) and thus detection of error, while others result in legal programs (mapping T_I) and are not

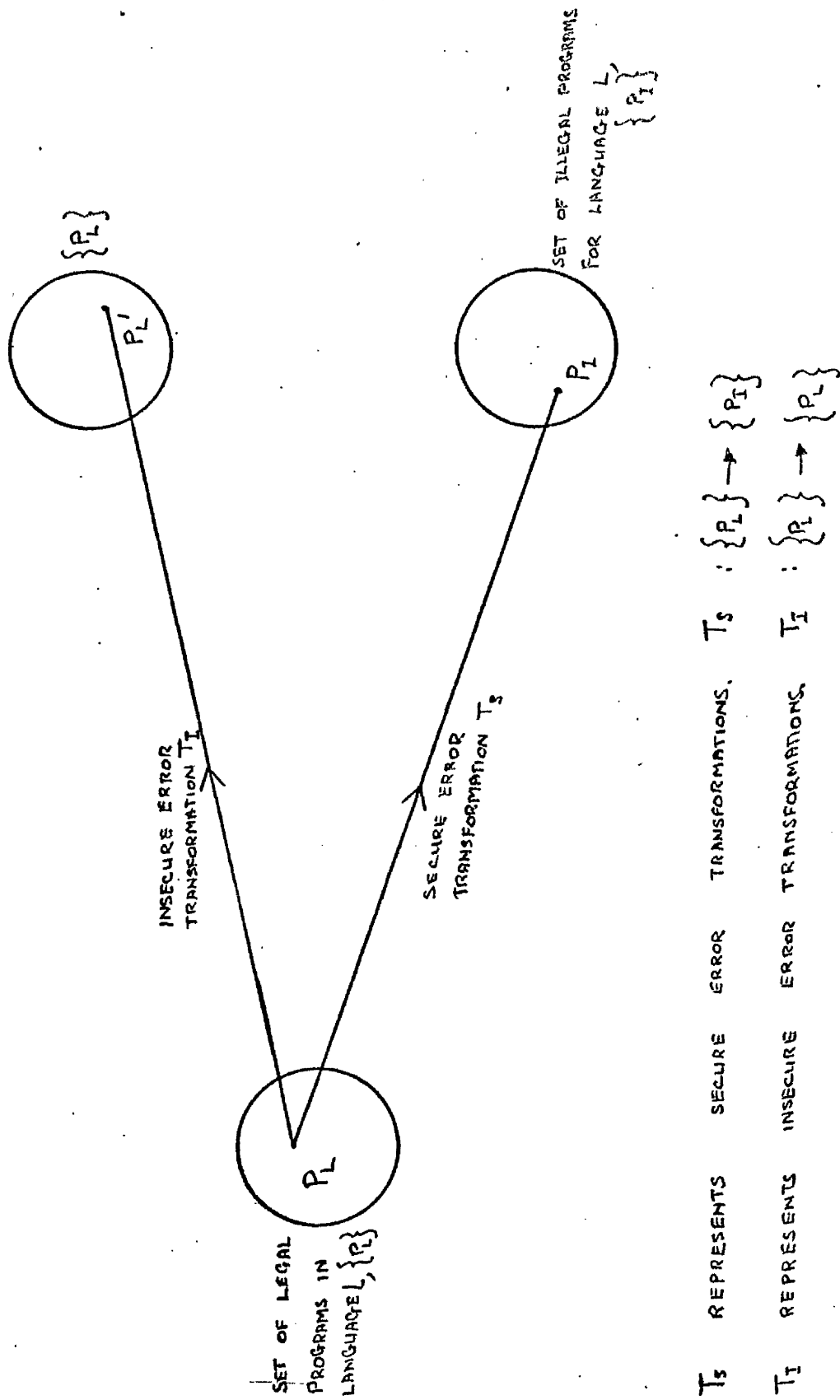


FIGURE 2-1 ERROR DETECTION.

detected. Ideally therefore, T_I is a null mapping.

In this context, we use the term (programming) error to indicate that the program in which error occurs does not faithfully represent solution of the particular problem involved. An error may thus be attributable to logic error in the formulation of the algorithm, coding error in the formulation of the program, language error, punching or clerical error.

2.1.2 Security, The State of the Art

Programming languages have, in general, been designed with aims such as machine independence, efficiency, generality. There is, however, increasing interest in the prevention and detection of programming oversight and error and a realisation that successful abstraction can contribute to both these aims cf. Hoare, Galler (Dah 72; Gal 74).

In the absence of any real understanding of the functioning of the human brain or of the causes of error occurring in the programming activity in particular, there are essentially two ways of approaching the design of secure languages; an empirical approach, and a combined intuitive and empirical approach:

(1) Empirical Approach

One approach is to consider the kinds of error which arise in programming and to deduce ways of preventing (or detecting) these errors. Various studies of this type have been carried out (Ich 74; Sim 73; You 74). The work of Ichbiah and Rissen in studying the characteristic errors of languages is of particular interest (cf. appendix G). This approach, however, tends to yield limited (though

important) results related to simple errors in a local context e.g. the Algol 60 end-comment is prone to error.

(2) Combined Intuitive and Empirical Approach

A more promising alternative is to use intuition and intelligent guessing (based on programming experience and study of characteristic errors) to identify error-prone constructs and devise (hopefully) more secure alternatives. This approach is likely to yield more spectacular advances as it essentially involves the formulation of a theory concerning secure language design. Examples of this approach include consideration of the goto by Dijkstra (Dij 68), the pointer by Hoare (Hoe 73), and scope rules by Wulf and Shaw (Wul 73).

Weissman (Weis 74) has considered how well programmers understand programs. By constructing sets of programs which include features he believed might affect the "psychological complexity" of a program, he obtained statistically significant results about factors affecting program readability: mnemonic identifiers, comments, paragraphing. Hoare (Hoe 72) and Wirth (Wir 74) discuss the value of abstraction, while Gannon (Gan 75) discusses various design decisions which influence security.

There appears, however, to have been little general discussion in the literature as to the precise manner in which design decisions might be used to increase security. We suspect that ideas of this kind are widely used, but presented as "fait accompli" in new programming languages e.g. Pascal. In the following sections, we therefore attempt to draw up an informal theory embracing these ideas.

While we find it possible to consider a more solid and machine-oriented view of secure language features, the design of a secure notation seems wholly human-oriented. We therefore find it convenient to consider notation and features independently.

2.1.3 Influence of the Features of a Language on Security

In this section, we show essentially that appropriate "high level" abstractions, with tight restrictions, aid security. We are forced to make certain assumptions about programmer discipline and goodwill, but we argue that the use of structured programming makes these assumptions plausible.

We consider the choice of the features of a language for security. We use the term features loosely to denote the set of primitive semantic computations of which a language is comprised. While most programming languages are, at least theoretically, universal (i.e. they can express any computable function), there is however, a considerable difference in the sets of computations that can be conveniently and efficiently specified in each language, according to the choice of features of the language cf. (Gal 74).

The features of a high level programming language are developed principally with the intention of permitting the programmer to specify necessary detail, but divorcing the programmer from the need to consider unnecessary detail: or, equivalently, they are designed to form an abstraction which is well-suited to easy and natural solution of problems in a particular area. Thus, for example, cf. Hoare (Dah 72), to the hardware of a computer,

and to a machine code programmer, every item of data is regarded as a mere collection of bits. However, to the programmer in Algol 60 or Fortran, an item of data is regarded as an integer, a real number, a vector or a matrix - the same abstractions as those that underlie the numerical application areas for which these languages were primarily designed.

A second major (and perhaps often coincidental) advantage of use of a high-level language is that it may significantly reduce the scope for programming error. As we shall show, this is true particularly if the abstraction is carefully chosen. For example, Hoare (Dah 72) observes that in machine code programming it is all too easy to make stupid mistakes such as using fixed point addition on floating point numbers, performing arithmetic operations on boolean markers, or allowing modified addresses to go out of range. In high level languages, such errors may be prevented by using the same operator for all forms of addition, by disallowing arithmetic operations on boolean variables, and by subscript checking.

In this section then, we investigate the design of secure abstractions. In effect we justify the intuitive notion that a high level and appropriate abstraction reduces the scope for programming error. We consider first an intuitive human-oriented view and subsequently a machine-oriented view on a more solid foundation.

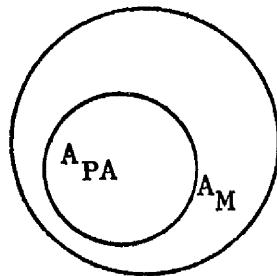
2.1.3.1 Human-Oriented View

We introduce the idea of "undesirable" actions allowed by a programming language and relate this to Parnas' idea of transparency. We define also the notion of

instability of language features.

Consider a programmer working in problem area PA and using a high level language L_{PA} which is implemented on a real or abstract machine M.

It is probable that a large number of actions or sequences of actions allowed in M are non-useful and in some cases actually undesirable for problem area PA. The figure below illustrates this situation. We will assume that M allows only a finite number of alternative actions and sequences of actions, A_M . A_{PA} represents the set of actions and sequences of action required for problem area PA.



Ideally, therefore, we should regard the appearance of the actions or sequences of actions represented by $A_M - A_{PA}$ as errors when they appear in programs for problem area PA and hence also in language L_{PA} .

These ideas are of course applicable to any forms of abstraction, as we illustrate in the following examples.

Example 2-1

Consider a typical electrical plug and socket used to connect circuits carrying an alternating current. Both plug and socket have 3 connections: earth, live, neutral. There are therefore 6 possible ways of connecting the plug and socket, but only the combination earth-earth, live-live, neutral-neutral is normally regarded as useful. If

the plug consists of 3 physically independent terminals, then any of the 6 combinations may occur with possible disastrous effects. The commonly used simple abstraction in which the 3 terminals of the plug are physically bound together in the form of an isosceles triangle permits the plug-socket connection to be made (assuming the use of a hammer is excluded! cf. programmer discipline, below) in the correct manner only.

Example 2-2

Consider a simple programming language P used purely to count the number of males and females entering a building. The programming language allows statements of the form:

```

<program> ::= increment <variable> by <constant>
<variable> ::= MALES | FEMALES
<constant> ::= <integer>

```

The base machine M upon which this language is implemented allows instructions of the form

```

<program> ::= <variable> := <variable> + <constant>

```

We list the possible sets of actions of P and B:

P

```

increment MALES by <constant>
increment FEMALES by <constant>

```

B

```

MALES := MALES + <constant>
FEMALES := FEMALES + <constant>
MALES := FEMALES + <constant>
FEMALES := MALES + <constant>

```

The last two forms of action are legal on the base machine, but not useful to our particular problem area. Programming language P therefore affords more protection for this problem area.

Thus, in general, we say that a programming language is insecure if it allows undesirable (sequences of) actions to be specified. For a typical programming language, however, it is neither possible to list all sequences of actions nor is it such a simple matter to determine undesirable actions. We must consider the meaning of undesirable more carefully.

2.1.3.1.1 Undesirable Actions in Programming Languages

In a sense (leaving aside for the time being arguments of efficiency and alternative algorithms), there is only one desirable action for a particular problem, all other specifiable actions being (currently) undesirable. In other words, ideally we would have a separate single-statement programming language for each new problem into which we merely substitute the current parameters of the problem.

This ideal situation may in fact occur in the form of dedicated machines (e.g. industrial process control). In general, however, this solution is not economically feasible: a programming language must therefore be sufficiently low level to allow solution of an economically acceptable range of problems. We recall, however, (cf. section 0.0) the trend towards higher-level programming languages and thus towards less general solutions to solving problems.

The size of the problem area (to which a programming language is oriented) is thus determined by economics and the (un)desirability of particular sequences of action is determined by their (un)desirability to a "significant" number of problems in the problem area.

We consider some further examples in this new light:

Example 2-3 Variable Types and Coercion

Distinct categories of data objects, such as real and integer in numerical analysis, apples and oranges frequently occur in real-world problem areas. In some cases, these distinctions may be reflected in the hardware realisation e.g. real, integer. In general, the association of one type of object with an object of a different type is not useful and hence undesirable. Thus, the expression x apples * y oranges is not normally useful. The association of types and type rules with the data objects of a program thus ensures that the abstract machine defined by the programming language does not violate real-world situations. Conversely, automatically invoked coercion tends to undermine the benefits of type checking. Under certain conditions, this may be useful, as with integer to real conversion in numerical analysis (cf. widening in Algol 68 (Wij 69)); but real to integer coercion (cf. Algol 60) is frequently dangerous.

Pointers may cause additional problems by allowing access to objects whose types are unknown. This difficulty is avoided in Algol 68 and Pascal by requiring that pointers be declared with the type of data they reference (Hoa 73; Wir 74; Gan 75).

In conclusion, we would expect security to be

enhanced by a strongly typed language such as Pascal and decreased by a typeless language such as Bliss (Wul 70). The introduction of integer subranges, read-only variables and user-defined scalars in Pascal may be viewed as a stronger form of typing.

Example 2-4

We list several simple and obvious undesirable actions:

- (a) The use of the value of a non-initialised variable is in general, not useful.
- (b) A loop of the form "1 : goto 1" is of little value except, perhaps for instruction timing.
- (c) Programs of the form "goto 1 ; X := Z" or "X := Z ; X := Y" are normally regarded as non-sensible (Tsi 73).

We feel intuitively that appropriate "higher level" abstractions allow fewer undesirable actions. Thus, for example, we feel intuitively that a for-statement provides a more secure way of describing iteration with a known number of repetitions than does a goto-statement. However, in this case, the question of undesirability is by no means so clear cut for several reasons:

- (a) The meaning of an "appropriate and higher level" abstraction is by no means well-defined.
- (b) Replacing a goto-statement by a whole series of new "higher level" control structures does not necessarily reduce the kinds of undesirable actions permitted.
- (c) The question of undesirability is easily coloured by the effect of notation rather than that of features alone.

We therefore support these intuitive ideas with a machine-

oriented approach. Before doing so, however, there are several further ideas which we wish to consider from a human-oriented viewpoint. We develop first a notation to describe these ideas and subsequently discuss ramifications of these ideas.

2.1.3.1.2 Notion of Transparency and Overtransparency

We wish to develop a suitable notation to describe these ideas about security: we find that they can be conveniently linked to the notion of transparency developed by Parnas (Par 72) as an aid to the design of hierarchically structured systems. In this section, therefore, we consider the terminology developed by Parnas, adapting and extending it to suit our particular context.

Machine-Oriented Interpretation of High-Level Language Programs

Since the concept of transparency is machine-oriented, we must first digress to consider how high-level languages may be interpreted in a machine-oriented fashion. This has the dual advantage of allowing easier discussion of examples and of removing irrelevant syntactic sugaring which we might otherwise allow to colour our judgement.

We assume that each high-level language considered is described by a suitable grammar. A program expressed in terms of a high level language may therefore be represented by the corresponding parse tree. We consider the transformation of the parse tree to an abstract syntax tree in which most of the superfluous (from a machine-oriented viewpoint) structure is discarded, leaving a more convenient computational model cf. (McK 74b). This process

is illustrated in figure 2-2. Each operator in the parse tree is associated with the syntactic class from which it is descended. Syntactic classes and their corresponding edges are deleted from the structure.

By traversing the abstract syntax tree in pre-order (root, left node, right node), we may further reduce this structure to the usual mathematical form of a function or mapping. In figure 2-2, this function is therefore:

if ($\leq(X,1)$, $:= (X, + (53,X))$)

where operators are treated as functions. The usual machine-oriented interpretation is more closely related to infix expressions than this prefix form:

X, 1, \leq , ifjump, 53, X, +, $:=X$

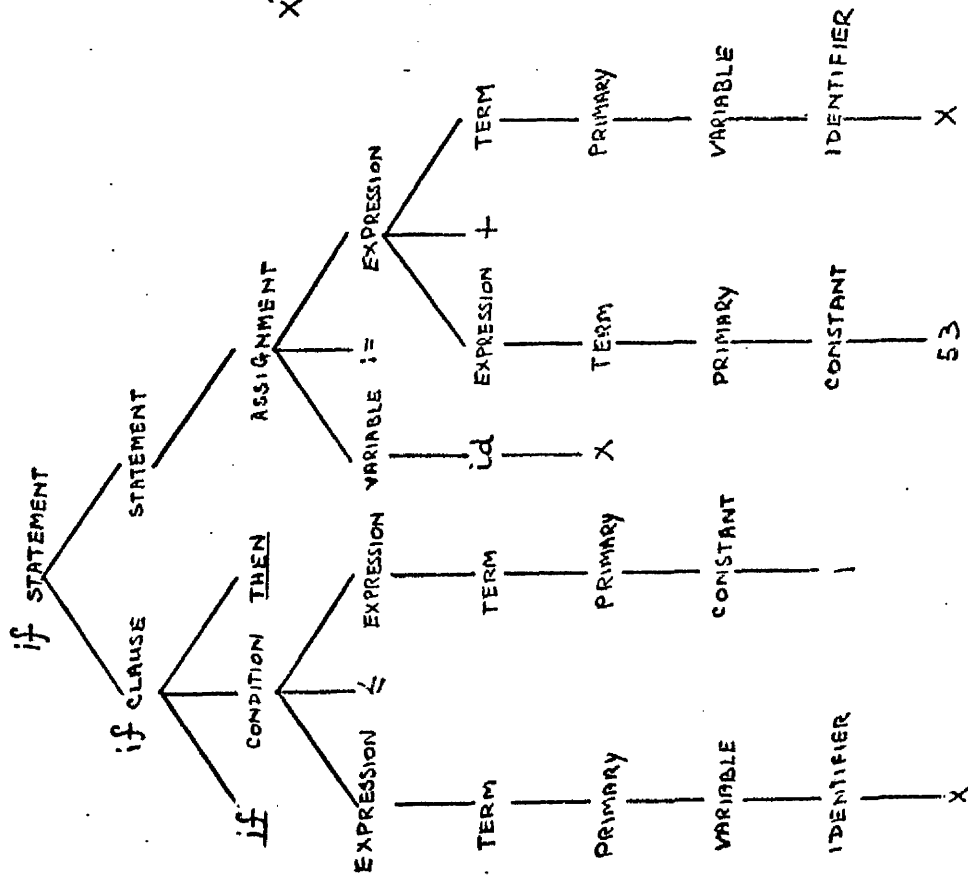
This interpretation may be extended to a complete set of language constructs.

Transparency

We consider a typical stage in the design of a high level language or a hierarchy of high level languages (or indeed any hierarchically structured systems). We assume that we have a well-defined lower level and are considering the design of the next highest level. The lower level may be either hardware or an intermediate level in our software design. We shall refer to either as the base machine. We assume that we are considering a proposal for a new abstraction to result in a new programmable machine which we shall refer to as the abstract (or virtual) machine.

We must determine the set of states which is possible for the base machine under arbitrary programs in the

PARSE TREE



CORRESPONDING ABSTRACT SYNTAX TREE

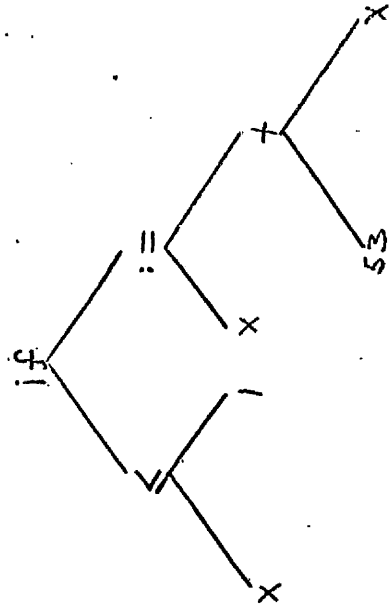


FIGURE 2-2 ABSTRACT SYNTAX TREE

"language" of the base machine. Also of interest is the set of state sequences which can be obtained by arbitrary base machine-language programs.

For any given implementation of the abstract machine, we can determine the set of base machine states and sequences of base machine states which is obtainable by running programs written for the abstract machine.

DEFINITION 2-3

If the abstract machine and its implementation are completely transparent, then any base machine state and any sequence of base machine states which we can obtain by programming the base machine are also obtainable by programming the abstract machine.

In the more common situation where some base machine sequences cannot be obtained by programming the abstract machine, we term the missing state sequences loss of transparency.

Parnas is interested principally in efficiency: if the missing state sequences are necessary to efficient programming of the abstract machine, then this loss of transparency is undesirable.

However, from the point of view of security, if the missing state sequences are unnecessary or undesirable (cf. above) then security is increased, and so this loss of transparency is desirable.

DEFINITION 2-4

If the abstract machine exhibits undesirable transparency, for a given problem area, we say that it is overtransparent for that problem area.

Thus, if we reconsider examples 2-3 and 2-4, we may say that programming language abstractions which allow typeless variables or use of undefined values are overtransparent for general problem areas.

We find it useful to examine the notion of overtransparency more closely and to distinguish two separate categories.

Given that a language is non-ideal, we should not be surprised to find that there are several different ways of expressing a particular action in that language e.g. in Algol W, a for-statement may be synthesised from while- and assignment statements.

If the primitives used to synthesise more sophisticated structures are themselves "desirable" in situations where more sophisticated structures are inappropriate, then we say that these primitives are partially overtransparent. If this is not the case, we say that the primitives are universally overtransparent.

DEFINITION 2-5

If, for a given application area, a particular language construct is overtransparent for all problems in that area, we say that the construct is universally overtransparent for that application area (since it allows (sequences of) base machine states which are never required in any situation).

Example:

In a high level language environment, we consider untyped variables as universally overtransparent (cf. example 2-3). Often what the programmer really needs is not untyped variables, but an abstraction allowing data

packing (Wir 74).

DEFINITION 2-6

If, for a given application area, a particular language construct is overtransparent for the programming of certain actions, but the transparency it offers is necessary and useful for the programming of certain other actions, we say that the construct is partially overtransparent for that application area.

Example:

The transparency offered by the while-statement in Algol W, for example, is required to specify iteration where the number of repetitions is unknown; it is, however overtransparent when the number of repetitions is known in advance - a for-statement is more appropriate in this situation.

We will often refer to universal overtransparency simply as overtransparency.

How do we determine which features are (universally) overtransparent? In a non-ideal language, there is a sense in which every base language operation is (or may be) necessary to program some action conveniently and efficiently; and hence, a sense in which no operation is universally overtransparent. We can neither predict nor optimally (as regards security or efficiency) cater for all constructs which might be required cf. (Lis 75). All we can hope to achieve is a language in which a large number of problems may be "near-optimally" solved.

In short, therefore, we regard a construct as universally overtransparent if it is undesirable for (the programming of) a "significantly" large number of actions,

where the meaning of "significant" is determined by economics.

2.1.3.1.3 Ramifications of Overtransparency

We discuss the relation between overtransparency and

- (a) Structured Programming,
- (b) Programmer Discipline and Goodwill,
- (c) Error Diagnostics,
- (d) Formal Specification and Implementation of Programming Languages.

(a) Structured Programming

As we have observed, it is not usually possible to present a programmer with the most secure abstraction for each different problem he has to solve. In general, a programming language must be low level enough (and hence overtransparent enough) to be applicable to a whole problem area. The technique of structured programming cf. Dijkstra (Dah 72) encourages the programmer himself to devise an abstraction ideal for his particular problem. Using the method of structured programming by top-down stepwise refinement, for example, the programmer refines a program written using this ideal abstraction, to successively lower level abstractions, until the program is completely expressed in terms of the actual programming language.

In this way, the programmer works with a secure abstraction at each level of refinement of his program, allowing only as much transparency as is necessary at each particular level. Dijkstra (Gut 75), by considering the amount of reasoning required to understand an arbitrary

program, has produced an argument supporting the intuitive feeling that it is easier to produce correct programs by successive refinement of an abstract program. We might thus view structured programming as a technique for the secure handling of overtransparency of a programming language for a particular problem.

We have three further points to make:

The notion of procedures or subroutines common to many programming languages can in fact be regarded as a special case of structured programming in which only a very restricted form of abstraction is permitted; and in which refinement of levels is implicitly defined and automatically carried out.

Henderson and Marneffe (Hen 72; Mane 73) have pointed to the likelihood of introducing errors in the program refinement process. It is possible that a secure extensible language system might reduce the frequency of this form of error by making the refinement process more automatic, and debugging more convenient.

Since there is no explicit mechanism to enforce each level of abstraction at the appropriate point in the refinement process, successful exploitation of this technique relies on programmer discipline in using only features from the current (and not lower) level of abstraction, and also in the choice of suitable abstractions.

(b) Programmer Discipline and Goodwill

Since programming languages are, in general, non-ideal, we might expect that (in any particular language) there are many distinct, but semantically equivalent ways of defining solution to a particular problem cf. partial

overtransparency. This is indeed the case in existing programming languages. Leaving aside the question of different algorithms, we find that this redundancy may arise in two ways.

In the following examples, we refer to an Algol-like language which includes for-, while-, if-, goto- and assignment-statements.

(1) The design of similar language features, each oriented towards slightly different situations. Thus, for example, a for-statement might be simulated by a while-statement together with suitable assignment-, if- and goto-statements. We might regard the for- and while-statements as different levels of the same abstraction.

(2) The result of natural redundancy in the language. For example, an assignment of the form "A := B" ($B > \emptyset$) might be expressed, rather pathologically as

"A := \emptyset ; WHILE A < B DO A := A + 1"

Equally pathologically, a goto-statement might be simulated by a for-statement e.g.:-

<u>GOTO</u> 1		<u>FOR</u> I := 1 <u>STEP</u> 1 <u>UNTIL</u> \emptyset <u>DO</u>
<statement 1> ; vs		<u>BEGIN</u>
1:		<statement 1>
<statement 2>		<u>END</u> ;
		<statement 2>

While the first example might be forgiven, most programmers would regard the second two examples as blatant misuse of the language constructs. It seems likely therefore that in the encouragement of secure programs we will be forced to rely to some extent on programmer goodwill and discipline. We return to this problem in the subsequent

machine-oriented discussion and show that with programmer goodwill, structured programming may relieve this problem.

(c) The Effect of Overtransparency on Error Diagnosis

We reconsider Parnas' discussion of a hierarchy of abstract machines (such as a hierarchy of programming languages). We show that in addition to allowing the programming in the abstract machine of undesirable sequences of base machine states, overtransparency may in certain circumstances cause poor diagnostics.

If an abstract machine allows the programming of sequences of action which are illegal on the base machine, or (in a hierarchically-defined system) on any lower level abstract machine, poor diagnostics may result from an inability to relate the base machine violation to the true cause of error, at the appropriate level. This situation is illustrated in figure 2-3. We find it useful at this point to consider programs as mappings in a manner similar to Manna (Mann 68). In fact, we have already shown how a high level language program may be regarded as a function by considering abstract syntax graphs (cf. transparency above). We regard a program as a function or mapping from a set of m ($m \geq 0$) distinct variables $\bar{x} = (x_1 \ x_2 \ \dots \ x_m)$, called input variables, to a set of n ($n \geq 1$) distinct variables $\bar{y} = (y_1 \ y_2 \ \dots \ y_n)$ called output variables. We will also refer to a set of r distinct program variables $\bar{z} = (z_1 \ z_2 \ \dots \ z_r)$ used to hold intermediate results. We will assume that it is possible for a variable to be both an input and an output variable i.e. in general

$$\{x_i, 0 \leq i \leq m\} \cap \{y_j, 1 \leq j \leq n\} \neq \emptyset.$$

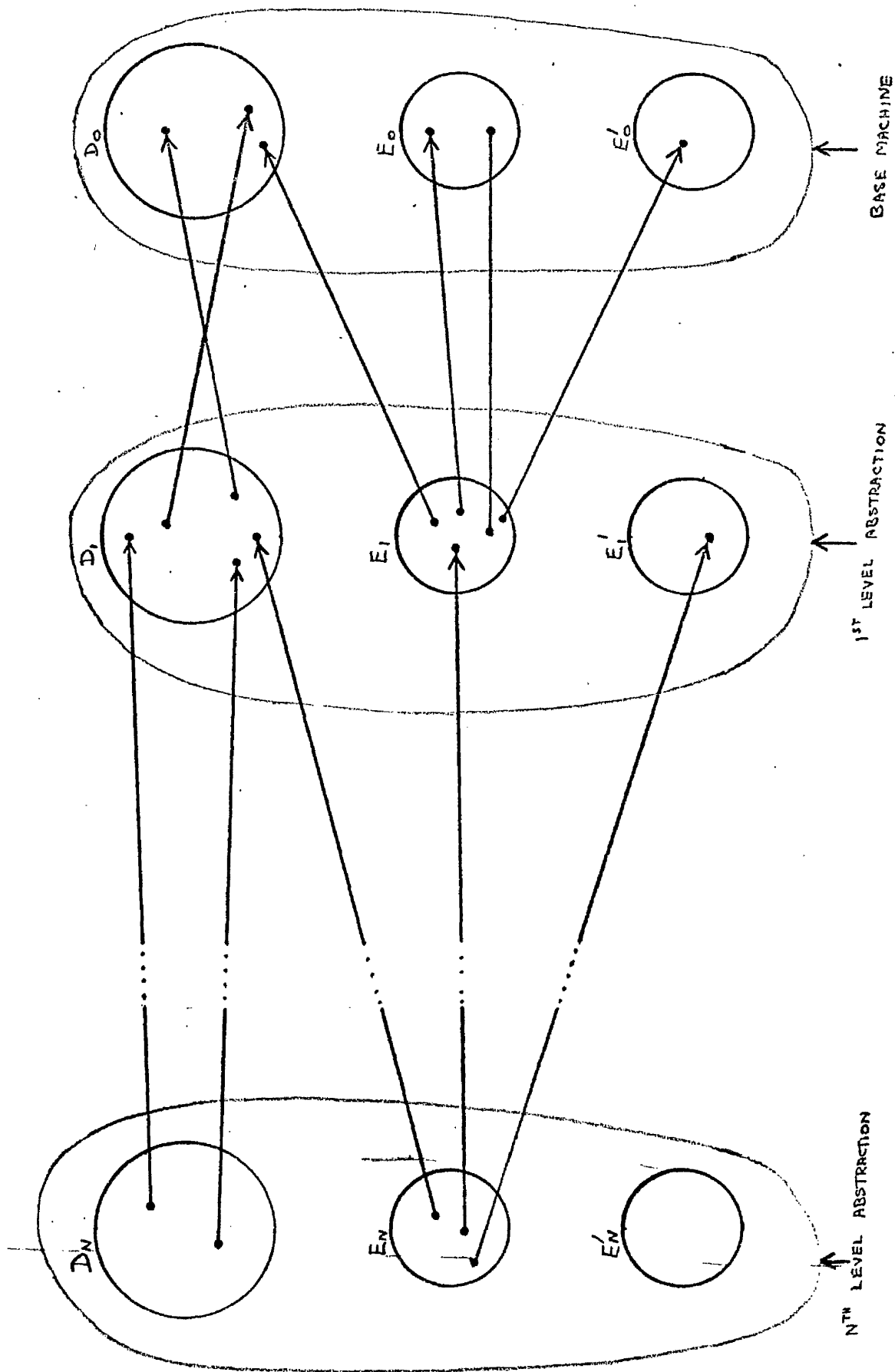


FIGURE 2-3 OVERTRANSPARENCY AND POOR DIAGNOSTICS

Thus, if we regard the program section of figure 2-4 as a complete program, $\bar{x} = (X)$ and $\bar{y} = (X)$.

In figure 2-3 then,

- (1) D_i represents the set of actions and sequences of actions permitted by the i^{th} level abstraction,
 $0 \leq i \leq n$.
- (2) $E_i \cup E_i'$ represents the set of actions and sequences of actions of the i^{th} level abstraction which are specifiable, but undesirable, $0 \leq i \leq n$.
- (3) E_i' represents the set of undesirable actions and sequences of action of the i^{th} level abstraction which are detected as errors, $0 \leq i \leq n$.

Edges represent mappings from (sequences of) actions at the $j + 1^{\text{th}}$ level to the corresponding (sequences of) actions at the j^{th} level, $0 \leq j \leq n-1$. The sets E_j' are terminal (i.e. mappings from these sets to lower level abstractions are null mappings) as they represent detected errors. Any element of E_{j+1} (i.e. any undesirable sequence of actions at level $j+1$ which is undetected) which is mapped on to an element of E_j' will thus be detected at level j . Since errors of this kind are detected at the wrong level of abstraction, they are frequently badly diagnosed. Outright failure to detect an error thus emerges as the worst case of poor error diagnosis. The above discussion is in fact an over-simplification, but adequately illustrates the point we are trying to make.

Example 2-5

Consider the assignment " $I := 0.9$ " where I is integer. If the programming language invokes automatic type conversion from real to integer, the value of I after

assignment is 0. Division by I at any subsequent (and perhaps remote) point in the program is likely to cause a real machine violation which cannot easily be related to the true cause of error.

Example 2-6

We consider a simple example from the IBM operating system OS/360. We can distinguish a user-oriented filing-system abstraction in which files may be created or deleted, catalogued or uncatalogued, and a lower level system abstraction in which files may be attached or detached. If an error occurs at user level because a file specified for attachment is uncatalogued or deleted, this error is not detected until the system level: at this point "attach failure" is the sole diagnostic message.

(e) The Relation of Formal Specification and Implementation of a Language to Overtransparency

We have assumed until this point that overtransparency was a matter of concern for language design only. It is, however easy to show that it is influenced also by the formal specification and implementation of the language.

In general, neither the language formally defined nor the language implemented will precisely model the intended language: for example, it is likely that the formal specification will not exclude all non-sensible programs and may not define all intended context-sensitive restrictions.

Examples:

- (1) The two consecutive statements "Z := Y ; Z := X ;" are not normally regarded as useful.

- (2) The bounds of indexed data structures (arrays for example) may not be properly defined.
- (3) Restrictions on the use of variables whose values are undefined may not be properly specified.

This overtransparency caused by the formal specification or the implementation (or both) reflects, in part, the inadequacy of the current state of the art (although vastly improved from the days of Fortran when many features were insufficiently and hence ambiguously defined).

The current practice is to handle some of the above-noted restrictions by use of English-language qualifiers in the formal definition and by semantic action in the implementation. Many restrictions are ignored because of the difficulty of implementation and hence the result is overtransparency.

Ideally, a secure language would be defined such that only sensible programs were also legal. Ideally, also, legality should be a purely syntactic problem (Wat 74; Kos 71), and it seems unlikely that language restrictions will be properly implemented until this is the case. In this respect, the development of formal syntactic definition systems such as affix grammars (Kos 71) and canonical substitution systems (Led 69) are undoubtedly of assistance. Affix grammars, for example, are capable of describing restrictions on variable types and use of variables whose value is undefined.

2.1.3.1.4 Stability

In addition to the notion of overtransparency, we feel it is also important to have some notion of the stability of language features. We say that certain

language features are unstable^{*} if they are "particularly prone" to undetectable error. We cannot at present define "particularly prone" formally, but we hope to give an intuitive understanding of the meaning. We justify the introduction of the term in(stability) by observing that this idea (although not described as such) appears to have been used in the elimination of undisciplined pointers and side-effects (in functions) in the language, Pascal (Wir 74; Hoa 73), the elimination of the unrestricted goto (Dij 68) and in restriction of scope of variables (Wul 73) and nested if-statements (Wein 75). We consider 3 examples:

Example 2-7 The Pointer

The following discussion is taken from Wirth (Wir 74):

When programming in assembly code, probably the most serious pitfall is the possibility of computing the address of a storage cell that is to be changed. The effective address may range over the entire store. A very essential feature of high level languages is that they permit a conceptual dissection of the store into disjoint parts by declaring distinct variables. The programmer may then rely on the assertion that every assignment affects only that variable appearing to the left of the assignment operator in his program. He may then focus his attention on the change of that single variable, whereas in machine coding he always has - in principle - to consider the entire store as the state of the computation. The

*This notion appears similar to that of "secondary effects" considered by Beckman (Bec 75).

necessary prerequisite for being able to think in terms of safely independent variables is the condition that no part of the store may assume more than a single name. Whereas this highly desirable property is sacrificed in Fortran by the use of the "equivalence" statement, Algol 60 loses it through the generality of its parameter mechanism and rules of scope. The use of undisciplined pointers allows reference to variables under a limitless number of alternative names. Security in pointer handling can be improved drastically through the following measures:

- (1) Each pointer variable is allowed to point to objects of a single type only (or to none); it is said to be bound to that type.
- (2) Pointers may refer only to variables that have no explicit name declared in the program.
- (3) The programmer must explicitly specify whether he refers to a pointer itself or to the object to which the pointer refers (no automatic coercion). This rule helps to avoid ambiguous constructs and complicated default conventions liable to misunderstanding. Both Pascal and Algol W provide disciplined pointers, while Algol 68 and PL/1 permit relatively undisciplined pointers (Hoa 73b).

Example 2-8 Side-Effects in Functions

The use of side-effects in functions may similarly destroy the above-noted desirable property of safely independent variables as in the expression "F(3) * X" where F : INTEGER PROCEDURE F (INTEGER VALUE I);

```

BEGIN
    X := 2*X;
    I
END_F

```

Example 2-9 Global Variables

Wulf and Shaw (Wul 73; Gut 75) extend Dijkstra's arguments against GOTO's to global variables - variables defined and modified over large portions of text. The authors argue that (a) keeping track of global variables is difficult and (b) global variables complicate the process of understanding a program segment whose actions depend on them. The authors point out that the problem is not so simple as the GOTO problem since (a) there is no "single offending construct" and (b) there are no accepted alternatives which avoid it. Desirable attributes of a more restrictive mechanism are enumerated:

- (1) The scope of a name should not automatically be extended to inner blocks.
- (2) The right to access a name should be granted by mutual agreement between creator and accessor.
- (3) Access rights to a structure and to its sub-structures should be decoupled.
- (4) It should be possible to distinguish different types of access.
- (5) Data definition, name access and storage allocation should be decoupled.

Instability need not necessarily imply overtransparency or vice versa, although in the above examples instability was corrected by reducing transparency; the original structures were therefore regarded as both unstable and overtransparent.

2.1.3.2 A Machine-Oriented View of Overtransparency

Having shown that a human-oriented view of overtransparency is intuitively appealing, but prohibitively

difficult to develop beyond the idea of restrictions to prevent undesirable actions, we attempt to make reparations with a machine-oriented view.

We argue essentially that under controlled conditions (such as those provided by top-down structured programming) use of appropriate "higher level" features in a program enhances security.

Given that we cannot, in general, design ideal languages, we are thus also able to deduce the best means of organising a small number of languages in such a way that we may solve a wide range of problems in a relatively secure manner.

Our approach is to show that we can improve the security of a language for a given program or program section by

- (a) reducing (or minimising) uncheckable redundancy in the specification of the algorithm, and
- (b) increasing (or maximising) checkable redundancy (in the form of restrictions or assertions).

We will avoid isolated discussion of the security of (the features of) a language for 2 reasons:

- (1) In general, there will be several equally valid and possibly equally secure means of solving a given problem.
- (2) As we have shown, security depends greatly upon the particular choice of constructs used to define the chosen algorithm, and hence, on programmer discipline.

We will thus consider only the means of specifying a given algorithm or subalgorithm and discuss how it might be more securely specified.

Information Theory and Redundancy

From the machine-oriented viewpoint, the design of secure programming languages has distinct similarities to areas of information theory (Ash 65). Information theory is concerned with the reliable communication of information, given that the correct signal is transmitted but that "noise" may distort and cause errors in the signal received. In programming, the cause of "noise" is little understood and its effect may lead to much more complex patterns of error which may consequently be harder to detect or prevent. Despite these differences, we nevertheless evaluate two ideas for detection and prevention of error suggested by analogy with information theory:

(a) Checkable Redundancy.

(b) Uncheckable Redundancy.

(a) Checkable Redundancy

In information theory, checkable redundancy aids the detection of error. In the case of programming languages, checkable redundancy must, at least trivially aid detection of error (Wir 75). We define checkable redundancy to mean redundancy which (1) conveys no new information about the specified algorithm if correct, but (2) specifies restrictions or assertions which may be used to determine the validity of the algorithm (in particular directions) but which are not evident from the specification of the algorithm itself.

Checkable redundancy might therefore take the form of invariants and assertions cf. (Flo 67b). We will assume that assertions hold "at a point" in a program while invariants are a stronger form of assertion, valid through-

out the entire program. We would expect therefore that security will be greatest when the number of assertions and invariants is maximised. (This is a necessary but not sufficient condition.) Presumably some optimal (minimal) form of defining all possible assertions and invariants can be found, but this need not concern us.

Assertions and invariants may be language imposed or programmer defined. If programmer defined, it might be argued that with increasing checkable redundancy, the incidence of error in specifying invariants and assertions is likely to increase. We accept this as inevitable, but argue that the situation is at least no worse. We consider the 2 possible cases:

- (1) Error in an invariant or assertion causes failure to detect the real programming error. This situation is no better, but no worse than before.
- (2) Error in an invariant or assertion causes detection of an error which does not in fact exist. We argue that the severity of this error is small as, in contrast to non-detection of a real error, it ought to be well diagnosed. Further, assertions can usually be sensibly defined only during program development; thus if an assertion is incorrect, we would expect the program will frequently be incorrect also. If nothing else, specification of assertions ought at least to encourage clear thinking and more careful programming.

The action of a program may often appear "stupid" to the layman - witness the demand for £0.0. The layman has a background experience of "what is sensible" in this context or in the real world. Programs should have sufficient information to allow a similar idea of sensible

actions in its environment.

Perhaps, strictly speaking, we should not regard assertions concerning input data or the real world environment as checkable redundancy since they do contribute information to the program. Such assertions should be part of the program specification. Unhappily, however, many programs fail to check data either exhaustively or at all. We consider some examples of checkable redundancy.

Example 2-10 Variable Types

In the association of a type with each program variable together with type rules defining the permissible combinations in which variables of particular types may appear in expressions and assignments, we are in effect introducing invariant relations (over the scope of validity) of the objects in a program. Automatically invoked coercion undermines the strength of these invariants while subranges, defined scalars and read-only variables impose further restraints.

Example 2-11 For Statements

Algol 68 and Algol W permit only a disciplined form of for-statement: in contrast to Algol 60, expressions in the for-statement are evaluated once only (at loop entry time). Although primarily introduced for optimisation purposes (Hoa 66), this does, however introduce assertions about expression evaluation (Knut 74). This is, however, at the expense of forcing explicit programming (with fewer language-imposed assertions) of problems requiring a less restricted form of iteration. However, as Knuth (Knut 73), Wirth and Hoare (Hoa 66) observe such problems do not occur frequently in practice.

Example 2-12 Local Variables

The declaration of a variable as local asserts that the variable may be accessed only within a well-defined scope.

Example 2-13 Programmer-Defined Assertions

Simple deductive assertions would not appear to be helpful. For example, after executing the statement "X := Y + 1" it appears somewhat banal to assert that "X = Y + 1". This form of assertion does not, however meet our definition of checkable redundancy. Programmer-defined assertions appear to be useful in two situations:

- (1) Assertions about input data or the real world (cf. above).
- (2) When the programmer identifies a special meaning (such as a procedure or the expansion of some "higher level instruction" in top-down structured programming) with a particular series of instructions it is worthwhile defining assertions to ensure that meaning is properly achieved.

e.g. Consider expansion of the abstract instruction mod in "A mod B" to:

"A - ((A div B)*B)"

From the definition of mod, the programmer can make the assertion $0 \leq A < B$.

(b) Non-Checkable Redundancy

We would expect, in information theory that non-checkable redundancy would increase the incidence of error by entailing the transmission of a larger amount of apparently unique and non-redundant information. Conversely

we would expect that reducing (or minimising) the length of the signal by avoiding non-useful redundancy would decrease the incidence of error.

The parallel situation for programming languages is not so straight-forward. While we would expect that redundancy in the form of mere repetition will detract from security by increasing the opportunity for error and programmer boredom, there are other less easily answered problems.

In reducing non-useful redundancy in the specification of the algorithm proper, we often in effect, increase the number of distinct instructions by developing new special-purpose (higher level or aggregate) instructions. This effect may, however, be countered by deletion of lower level instructions. We make the following assumption about programmer discipline and programming error.

Assumption

Given a programming language of "manageable proportions" (e.g. Pascal, Algol W) in which each instruction has a well-defined and non-overlapping purpose, we assume that a programmer can, in general, correctly select instructions appropriate to his purpose. We show (cf. below) that structured programming can aid this selection process. If we do not accept this assumption, then we must accept that no programming language can ever be secure.

If we do accept this assumption, we would expect reduction (or minimisation) of non-checkable redundancy to aid security. The minimally redundant form is achieved when a program consists of a single instruction in which each parameter is specified once only and there are no

intermediate variables. In terms of functions, this ideal situation is achieved when the function $f(\bar{x}, \bar{y}, \bar{z})$ representing the program consists of a single function in which parameters \bar{x} and \bar{y} are minimally specified and $\bar{z} = 0$.

We hasten to reassure the reader that we are considering redundancy from a machine-oriented viewpoint. The so-called syntactic sugaring (considered later) is useful only from the human-oriented viewpoint and has no influence upon the security of features.

We observe that the aims of maximisation of assertions and minimisation of redundancy are not mutually independent. In a minimally redundant program only environment assertions can be specified. However, since this ideal is rarely achieved, the combined approach is generally useful.

Example 2-14

Consider the set operator \in . Suppose we wish to determine whether $v_1 \in \{v_2, v_3\}$, where v_1, v_2, v_3 are variables. In an Algol-like language, this would be written as " $v_1 = v_2 \vee v_1 = v_3$ ".

This has machine-oriented form $\vee(=(v_1, v_2), =(v_1, v_3))$. We may minimise redundancy by using a single instruction for $\vee(=(), =())$ and minimising parameter specification: $\in(v_1, v_2, v_3)$ or, with syntactic sugaring $v_1 \in \{v_2, v_3\}$. This is similar to the use of powersets in Pascal.

In the following examples, we will omit much of the commentary for the sake of brevity.

Example 2-15

Consider an Algol-like program section which inter-

changes the values of two variables A and B:

$$W := A; \quad A := B; \quad B := W;$$

$\equiv \text{seq } (:=(W,A), := (A,B) , := (B,W))$ where seq denotes a sequence of instructions.

We minimise redundancy by deleting the work variable, minimising the specification of A and B and reducing the number of instructions to 1:

$$\longleftrightarrow (A, B)$$
$$\rightleftharpoons A \longleftrightarrow B$$

Example 2-16

Consider using some form of (real) machine code to access the I^{th} element of an array A:

get I

store I in R where R, S are registers

store base address of A in S

fetch indirect R,S

```
seq (get (I), store (I,R), store BA (A,S),
fetch indirect (R,S))
```

Redundancy is minimised by avoiding repetition of I, taking R and S to be implicitly defined by the single instruction replacing the above sequence: index (A, I)

$$\equiv A \cdot [I]$$

Example 2-17 GOTO-statement

It is already fairly generally accepted (Dij 68, Wul 72, Wir 74, Knut 74) that the unrestricted GOTO as it appears in Fortran and Algol-like languages is too permissive for commonly used high level languages and their associated problem areas. We consider a few examples of minimising redundancy in sections of programs using the GOTO.

Example 2-18

Consider the section of program:

IF A = B THEN

BEGIN

P := Q;

GOTO ON

END;

S := R ;

ON:

$\equiv \text{seq}(\text{IF}(=(A,B), \text{seq}(:=(P,Q), \text{GOTO}(\text{ON}))), :=(S,R), \text{label}(\text{ON}))$

We might reduce (but not minimise) redundancy:

IF (=(A,B), := (P,Q), := (S,R))

\equiv IF A = B THEN P := Q ELSE S := R

In this case we avoid minimising redundancy since we expect the two assignment statements to act as true parameters and to change from problem to problem. For particular specialised problem areas, however, minimum redundancy might indeed be useful.

Example 2-19

Consider the program section:

I := 1;

WHILE I < N DO

BEGIN

A [I] := \emptyset ;

I := I + 1

END

$\equiv \text{seq}(:=(I,1), \text{WHILE} (<(I,N), := (\text{index}(A,I), \emptyset), := (I, +(I,1))))$

We may reduce redundancy by minimising specification of I:

$$f(I, 1, 1, N, := (\text{index}(A, I), \emptyset))$$

$$\equiv \text{FOR } I := 1 \text{ STEP } 1 \text{ UNTIL } N \text{ DO } A \begin{bmatrix} I \end{bmatrix} := \emptyset$$

Once again we avoid minimising redundancy if we expect the statement " $A \begin{bmatrix} I \end{bmatrix} := \emptyset$ " to vary. If this is not the case, then by minimising redundancy, we obtain:

$$:= (A, \emptyset)$$

$$\equiv A := \emptyset$$

Example 2-20

The principle of minimisation of redundancy is capable of uniformly handling cases of inappropriate abstraction e.g. for-while construct in Algol 60 where a simple while-statement would be more appropriate. Consider the pathological assignment construct (cf. section 2.1.3.1.3(b)):-

$$A := \emptyset; \text{ WHILE } A < B \text{ DO } A := A + 1$$

$$\equiv \text{seq} (:= (A, \emptyset), \text{ WHILE } (< (A, B), := (A, + (A, 1))))$$

Minimising, we obtain

$$f(A, B)$$

or $A := B$

Until now, we have accepted a rather intuitive notion of "higher level" and "less transparent". From the foregoing examples, it appears as though minimising redundancy and maximising assertions satisfies these intuitive notions. We therefore choose to define higher level and less transparent from this machine-oriented viewpoint.

The technique of structured programming by top-down stepwise refinement is by definition a heuristic method of producing the realisation of a given algorithm in such

a way that near-minimal* redundancy (within the limitations of the particular language involved) is guaranteed. Thus, given programmer goodwill, structured programming should aid the selection of appropriate instructions (cf. assumption above).

2.1.3.3 Language Features for Security - Summary and Conclusions

From an intuitive point of view, we made the hypothesis that two factors cause insecure features:

- (1) overtransparency,
- (2) instability.

We observed problems in defining "higher level" and "less transparent", but hopefully obtained a better understanding of the basic aims of abstraction.

A machine-oriented view postulated minimising redundancy in the specification proper and maximising assertions concerning this specification. This approach supported our intuitive ideas concerning "higher level" and "less transparent". It also led us to believe that structured programming (if properly used) guarantees relatively secure specification of programs within the limitations of the language used.

Improving the security of a language by minimising (reducing) redundancy thus tends to take on the character of

- (a) identifying primitive (to the language) sequences of instructions which might be replaced by a single

* Near-minimal, since in general there will be several equally valid expansions.

aggregate instruction (cf. locally maximising security), and

- (b) identifying primitive (to the language) groups of data objects which might be replaced by a single (aggregate) data structure.

The designers of Algol 60 were perhaps the first to recognise the value of aggregate instructions of this form e.g. for-statement, while-statement, compound statement, if-statement. These aggregates have gradually been refined to produce: case-statements, repeat-statements, restricted for-statements, while-until statements.

Comparable forms of data structure aggregates have been realised only much more recently e.g. Garwick (Gar 68), Standish (Sta 69). Once the importance of data structure aggregates had been recognised, however, research rapidly gained impetus and might indeed be considered to have surpassed interest in aggregate instructions, since it is now common-place for programming languages to allow data structure extensions e.g. Pascal, Algol 68. Current research into recursive data structures (Hoa 73b; Hoa 75) and clusters (Lis 74; Lis 75) appears to hold at least as much promise.

To keep languages to manageable proportions, only commonly occurring sequences of instructions (groups of data objects) should be identified as suitable candidates for aggregation. Suppose, in the absence of ideal languages we were to design a small number of languages in which we might with relative security handle a wide range of problems. It follows from the above that the most successful approach is to identify problems requiring

similar features with a single language.

Figure 2-4 summarises examples of security-improving features.

2.1.4 Influence of Syntax and Pragmatics on Security

We consider two topics concerning the design of secure syntax and pragmatics:

- (a) Notation
- (b) Instability

(a) Notation

We consider four aspects of notation: natural notation, syntactic sugaring, structure and syntax.

(1) Natural Notation

Experience in the use of both natural and programming languages suggests that a notation which is natural (based on the user's background experience) reduces the difficulty of formulating correct programs (Wein 71; Gan 75).

For example, in an experiment by Gannon (Gan 75), errors in arithmetic expressions were examined. Fewer errors occurred under left-right evaluation with traditional precedence than under right-left evaluation with equal precedence (as in APL).

If, however, the traditional notation is particularly error-prone, there might well be a case for persevering with an alternative notation.

(2) Syntactic Sugaring

We might expect that the choice of suitably mnemonic syntactic sugaring would reduce error incidence. Weissman (Weis 74) has in fact produced statistically significant

Examples of Improving Security of Language Features			
Stability	Minimising (Reducing) Redundancy		Maximising (Increasing) Program Assertions
	Aggregate Instructions	Aggregate Data Objects	
Disciplined pointer " GOTO " global variables No side-effects No call-by-name Disjoint procedure parameters Restrictions on depth of nesting of control structures	<u>Control Structure</u> For-statement Case-statement If-statement Compound statement While-statement Repeat-statement <u>Others</u> User-defined extensions	Arrays Records Lists Strings Files <u>Others</u> user-defined extensions e.g. clusters, recursive data structures	Strong Typing Subranges Local variables Access (read/ write) For-statement restrictions <u>Others</u> Programmer-defined assertions

FIGURE 2-4 Security-improving language features.

results showing that this is indeed the case for variable identifiers. We would expect further, that the choice of "meaningful" (in terms of the programmer's environment) symbols to delimit parameters in if- and for-statements, for example, in Algol 60 is much more informative than the corresponding choice of delimiters in Fortran. Too much redundancy in this form, however becomes laborious, unreadable and prone to clerical error (Gan 75).

We might include under this heading commentary, formatting and paragraphing which Weissman has also shown to have a significant effect on programming. It is also interesting to note that similar advice is given to technical writers (Rat 66).

Assertions and invariants may have a similarly pedagogical effect (cf. section 2.1.3.2).

(3) Structure

Appropriate high level structures reflect the underlying structure which they denote more obviously to the reader. We suspect that this is because the reader obtains more information concerning the structure from local context. Knuth (Knut 74), for example says that GOTO's and machine-like programs are devoid of structure or, more precisely, it is difficult for our eyes to perceive the program structure.

Example 2-21

Consider the use of GOTO to implement a repeat statement:

1:

S;

vs

IF \neg B THEN GOTO 1

REPEAT

S

UNTIL B

where B is some boolean expression and S a statement. The structure of the program section is immediately obvious from the local context (REPEAT) in the second example, but not in the first. This difference is trivial in small programs, but becomes much more critical in larger programs where the structure may be much more difficult to discern.

The use of procedures (subroutines) or any other form of modular decomposition will similarly supply structuring information.

Besides aiding the understanding of structure, it is at least equally important that language abstractions do not obstruct the understanding. We consider two examples:

Example 2-22

The use of nested if-statements to denote alternative courses of action which lie on an equal footing is poor, as it suggests a nested structure which does not in fact exist in the abstraction it represents. Figure 2-5 shows a more appropriate structure. This idea is substantiated by Weinberg's research (Wein 75). Weinberg has in fact suggested imposing restrictions on the depth of nesting.

Example 2-23

Knuth (Knut 74) considers the problem of a loop which is performed " $n + \frac{1}{2}$ " times. One common practice for avoiding use of GOTO in such loops is to duplicate the code for the section of the loop to be repeated the extra $\frac{1}{2}$ time:

The nested if-statement

```
"IF B1 THEN S1
  ELSE IF B2 THEN S2
    ELSE IF B3 THEN S3
      .
      .
      IF BN THEN SN "
```

is a poor substitute for the equivalent statement

```
"ONE OF BEGIN
  B1 : S1 ;
  B2 : S2 ;
  B3 : S3 ;
  .
  .
  BN : SN
END "
```

FIGURE 2-5

Appropriate structuring.

" S ;] $\frac{1}{2}$] $n + \frac{1}{2}$	
<u>WHILE</u> \neg B <u>DO</u>			
<u>BEGIN</u>			
<table border="0"> <tr> <td>T ;</td> <td rowspan="2">] n</td> </tr> <tr> <td>S</td> </tr> </table>			T ;
T ;] n		
S			
<u>END</u> "			

where B is a boolean expression and S and T are statements.

This structure is better reflected by statements of the form:

```
" DO : LOOP
  S ;
  IF B THEN EXIT LOOP ;
  T
OD "
```

or by the (higher level) structure proposed by Dahl (Knut 74):

```
"LOOP:
  S
  WHILE B
  T
REPEAT "
```

As we have observed cf. section 2.1.3.2., structured programming provides one means of guaranteeing choice of structures which reflect the underlying abstractions which they represent, as well as possible within the limitations of the language.

(4) Syntax

Wirth and Hoare (Hoa 66; Hoa 73) observe that while it is of course possible to analyse complex syntactic structures, that both human and computer have difficulty in doing so. This results in greater occurrence of error

and misunderstanding on the part of the programmer, and poor detection of error on the part of the machine. It would seem wise, therefore, to restrict the syntax of a language to the simplest form compatible with a natural and suitably mnemonic notation.

In particular, it appears useful to avoid apparently arbitrary context-sensitive restrictions, when possible (Wein 71). For example, if the GOTO is deleted from Algol 60 and replaced by suitable higher level control structures, such restrictions as (a) entry to a block is through block beginning only, or (b) entry to a for-statement is through statement beginning only, become unnecessary.

(b) Notational Instability

We say that a notation is unstable if it is not possible to detect commonly-occurring trivial errors such as mispunching or omission of a symbol.

Example 2-24 Transpositional Errors

Consider the statement $A[I] := A[I] + 1$. If the closing bracket is transposed with the symbols "+ 1", the error in the resulting text $A[I] := A[I + 1]$ is undetected (Gan 75). This error may be avoided by using the equivalent statement INC $A[I]$.

Example 2-25 Errors of Omission

Omission of the ";" symbol following an Algol 60 comment or end-symbol may cause instability e.g.

"END X := X + Y". In Algol W, end-comment error is avoided by restricting the comment at this point to a single identifier e.g. END OF_LOOP X := X + Y".

Example 2-26 Clerical Errors

In Fortran, misspelled identifiers are incorrectly assumed to represent declarations of new variables. Mandatory declarations prevent this form of error.

Example 2-27 Defaults

Defaults frequently cause instability by allowing the programmer to be imprecise when precision is in fact important (contrast abstraction).

e.g. (1) In Algol 60, parameters are by default, call-by-name.

(2) In Snobol 4, an assignment expression is by default the null string, as in "S = ".

Use of "fail-safe" defaults may in some cases be acceptable.

Unstable notation is readily detected by study of characteristic errors and is usually easily avoided by simple modifications to notation or features.

2.1.5 Conclusions and Summary

In the foregoing, we have developed an informal theory of language design for security. As Gannon (Gan 75) has observed, we cannot at present hope to prove such a theory, although we can present supporting evidence. Previous studies of programming errors (Gan 75; Ich 74) support rather than conflict with the ideas of this theory as does the brief survey of characteristic errors of Algol W and Algol 60 based on Pirie's work (Pir 75), appendix G. As we have indicated, this theory ties in with and is therefore supported by the notion of structured programming. In the final analysis, however, the value of this theory will stand or fall according to study of the characteristic

errors of the extensible language designed in chapter 3.

Cost

Gannon (Gan 75) has demonstrated that very small changes in language design can have a considerable effect on security, without involving major sacrifices in other design criteria such as efficiency or generality.

Use of higher level abstractions will inevitably reduce efficiency for actions not specifically catered for, but provided frequently occurring cases are well-accommodated, this penalty is small (cf. section 3.1.1.1 common special cases). Further, Knuth and Dijkstra (Knut 74) consider the application of "disciplined optimisation" to well-structured programs.

Run-time efficiency will however be affected by checking of invariants and assertions, and compile-time efficiency by strict enforcement of context-sensitive restrictions. Against this, however, must be measured the decrease in programming and debugging effort, machine-time used in development, and the increase of confidence in software.

2.2 Security of Extensible Languages

2.2.0 Introduction

Why do we consider security to be such an important feature in the design of extensible languages? If nothing else, our reasons for considering security of programming languages in general, apply equally to the base and extended forms of extensible languages.

We believe that security is, in fact, of even greater importance here, because errors can occur not only

in the usage of the base and extended languages, but also in the definition of extensions to the base; the variability open to the user is considerably increased (Fel 68; Sol 74); the user has the power to define insecure extensions (e.g. poor abstraction with unstable features).

Furthermore, since extensions are often hierarchically defined, we might expect "sensitivity" of higher level extensions to error in the lower level extensions in terms of which they are defined; if the user is permitted to modify lower level extensions or base language constructs, then he has the power to subvert these hierarchically defined structures.

In this section we shall be concerned principally with the identification of those areas of the definition structure which we consider are in general overtransparent. (We need not concern ourselves with choice of syntax and pragmatics since this varies little from general programming languages).

We must consider in this case not only the security of the base language and the extensions defined (i.e. of the extended language), but also the security of the metalanguage used to define extensions. We find it convenient to introduce the following abbreviated notation:

DEFINITION 2-7

We shall refer to that part of the metalanguage (of an extensible language) used to define the semantics of extensions as the semantic metalanguage.

DEFINITION 2-8

We shall refer to that part of the metalanguage (of an extensible language) used to define the syntax of

extensions as the syntactic metalanguage.

In the following sections, we consider first the security of the base and extended language and subsequently the security of the metalanguage.

2.2.1 Security of the Base Language and Extended Language

The concept of security of the base language is identical to that of programming languages in general; it therefore needs no further expansion. The same concepts are also applicable to the extended language but some expansion is however necessary.

Extended Language

We consider (a) Programmer Discipline and (b) Overtransparency.

(a) Programmer Discipline

We consider the parallel to the question of programmer discipline in simple programming languages cf. section 2.1.3.1.3(b). In the context of extensible languages, this implies that while we can aim at ensuring that the metalanguage used to define extensions is secure, we cannot, in general, ensure that the new language constructs defined by the programmer are in fact themselves well-designed and secure additions to the base language.

(b) Overtransparency of the Extended Language

We recall from section 2.1.3 that we consider a programming language overtransparent for a particular problem area if it allows non-useful sequences of actions. In the context of an extensible language, a new dimension is added to this notion. The aim of extensibility is to

adapt to new problem areas: to provide constructs more appropriate for these problem areas. We observed in section 2.1.3, that overtransparency depends on the problem area. It is therefore possible, that constructs in the base language which were not overtransparent for the base language problem area, may in fact be overtransparent for the problem area which the extended language is intended to serve.

Since we refer to this result again, we find it convenient to present it as a simple proposition:

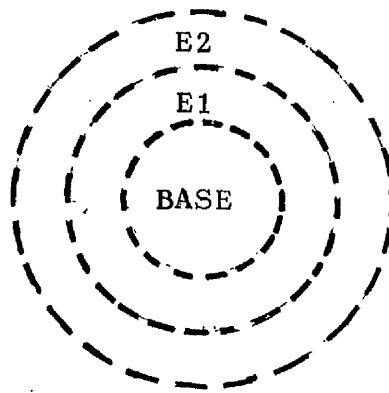
Proposition 2-1

A construct may become overtransparent in the extended language while being both useful and necessary in the base language.

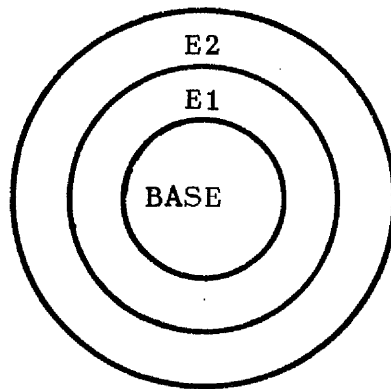
Reasoning

The justification for this proposition appears above.

One method of dealing with this form of overtransparency would be to consider extension through a hierarchy of abstract machines (cf. section 0.1): a completely new extended language being defined in terms of the original base language or in terms of another extended version of the language. Overtransparency is thus avoided in new problem areas as only constructs of the current version of the language are accessible. Figure 2-6 illustrates this system. In most conventional forms of extension mechanism, it is (usually) possible, at any point to access any construct from the base or any extended version of the language.



- (a) Conventional form of extension mechanisms. Dialects indicated in the diagram are BASE, $\text{BASE} \cup \text{E1}$, $\text{BASE} \cup \text{E1} \cup \text{E2}$. Broken rings indicate that inner levels are also accessible.



- (b) Extension by bootstrapping. Dialects indicated in the diagram are BASE, E1, E2. Solid rings indicate that inner levels are inaccessible.

FIGURE 2-6

Extended Language and overtransparency

2.2.2 Security of the Metalanguage

Since the metalanguage used to define extensions can be regarded as a programming language, our existing notion of security of programming languages is applicable. We have little to add in the area of choice of syntax and pragmatics for security. However, the implications of overtransparency for the metalanguage has received little attention and is therefore worthy of individual consideration.

Overtransparency of the Metalanguage

Typically, an extension is added to the base language by defining its syntax in a syntactic metalanguage such as BNF, for example; usually this structure is related to the grammar of the existing language, in order to define the context in which it is applicable.

The semantics of an extension is defined in a semantic metalanguage such as real or abstract machine code, or text of the base and/or extended language. This definition is related to the corresponding part of the syntactic definition. Semantic definitions are frequently hierarchically structured.

Example 2-28

We consider the definition of a while-statement in an Algol 60-like language. We introduce this construct by modifying the existing syntactic class `<statement>`. The production

$$\langle \text{statement} \rangle ::= \langle \text{if-statement} \rangle \mid \langle \text{assignment statement} \rangle \mid \langle \text{do-statement} \rangle$$

becomes

$$\begin{aligned} \langle \text{statement} \rangle &::= \langle \text{if-statement} \rangle \mid \langle \text{assignment-statement} \rangle \mid \\ &\quad \langle \text{do-statement} \rangle \mid \langle \text{while-statement} \rangle \\ \langle \text{while-statement} \rangle &::= \underline{\text{WHILE}} \langle \text{expression} \rangle \underline{\text{DO}} \langle \text{statement} \rangle \end{aligned}$$

We might define the semantics of this construct by specifying the semantically equivalent base language constructs:

```
"DO : LOOP
  IF <expression> THEN <statement> ELSE LEAVE LOOP FI
OD "
```

We must investigate what we feel is overtransparent in these forms of hierarchical structures. Before doing so, however, we digress to consider the general forms and representation of these structures and the manner in which they are modified. We consider (a) Syntax Graphs and (b) Semantic Graphs.

(a) Syntax Graphs

We can represent syntactic metalanguages which correspond to Chomsky Type 2 grammars (Hop 69) by a directed graph. We call this graph the syntax graph:

- (1) Each non-terminal or terminal of the grammar is represented by a node of the graph.
- (2) For each production of the grammar, a directed edge is drawn from the non-terminal on the left-hand-side to each terminal or non-terminal on the right-hand-side of the production.

This representation does not distinguish different alternatives in the definition, nor does it associate an order with the edges: this is however irrelevant to our

purposes here. In addition, this representation is unsuitable for context-sensitive grammars (i.e. Chomsky Type 0 and 1), but since we use the syntax graph for the purposes of illustration only, we can afford to ignore this short-coming.

Example:

We consider the type 2 grammar:

$S ::= a A S$

$S ::= a$

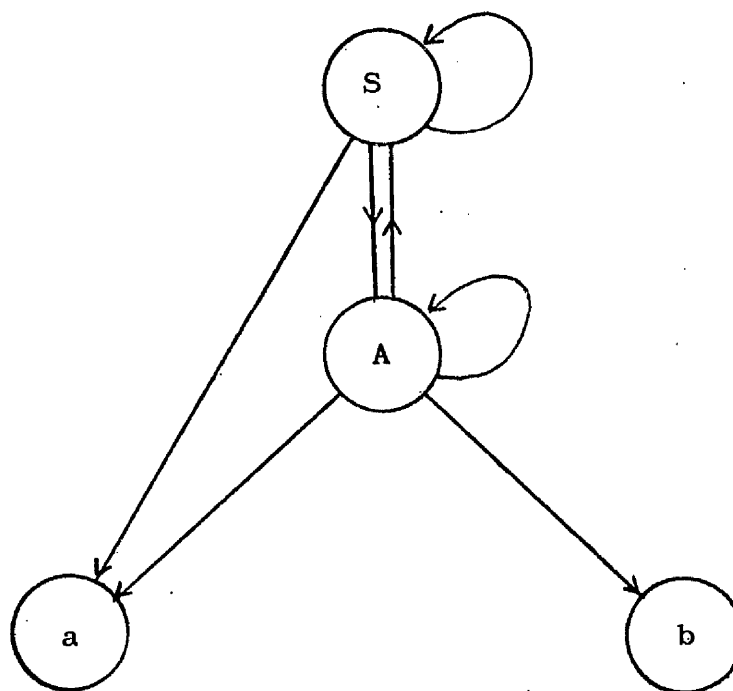
$A ::= S b A$

$A ::= b a$

$A ::= S S$

where S , A are non-terminal and a , b are terminal symbols.

The corresponding syntax graph is:



When extensions to a language are defined, the syntax graph may, in general, be modified by deleting or replacing existing nodes and edges or by inserting new

nodes and edges in the syntax graph. In some systems, new nodes are always joined to existing nodes by one or more edges i.e. the modified syntax graph is connected; in other systems, this is not the case i.e. the modified syntax graph is disconnected.

(b) Semantic Graphs

Although rather artificial, we find it useful, at least conceptually, to represent semantic features by nodes of a graph. The semantic features of a language or its extensions may, as we have seen, be defined hierarchically or in terms of a real or abstract machine code. As in the case of syntax, we can represent semantic definition by a directed graph, which we call the semantic graph:

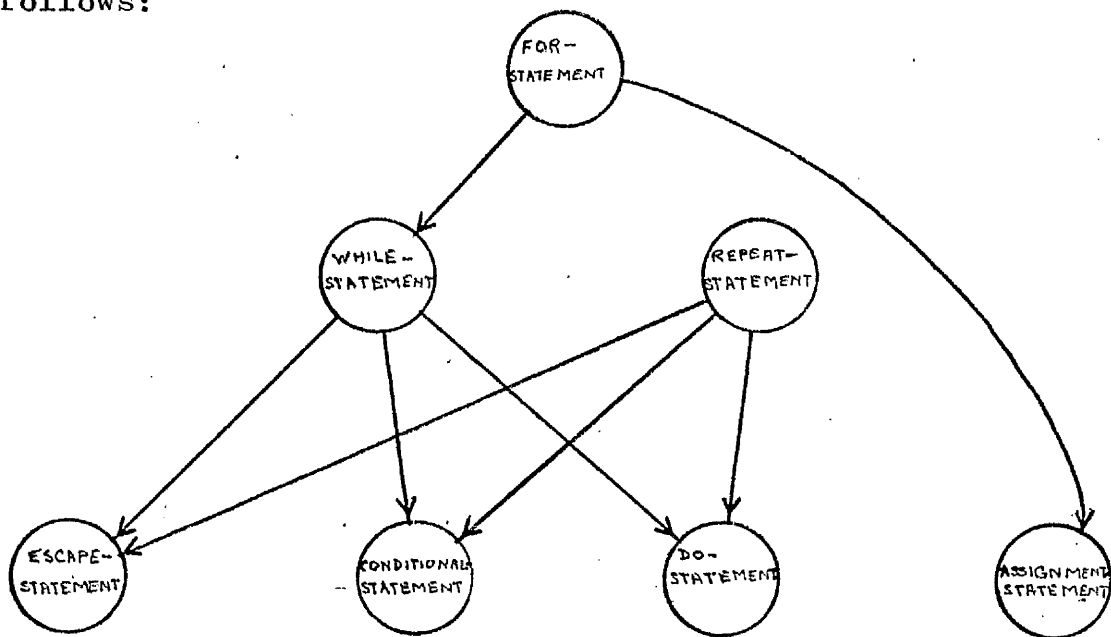
- (1) Each feature of the language is represented by a node of the graph.
- (2) A directed edge is drawn from each node to each of the nodes (if any) in terms of which this feature is defined.

In general, although nodes may be hierarchically defined, they may not be mutually recursive: semantic graphs, therefore, are usually of a much more restricted form than syntax graphs.

Example 2-29

We consider the definition of typical high level language control structures. The base language consists of a conditional-statement, an escape-statement and a do-statement. While- and repeat-statements are defined in terms of these, and a for-statement in terms of the while-statement. The corresponding semantic graph is as

follows:



Since the semantic graph is generally a fairly simple graph, it is usually possible to distinguish distinct hierarchical levels. As in the case of syntax graphs, extensions may be introduced by replacing or deleting existing nodes and edges or by defining new nodes and edges. In this case, the modified graph is always connected.

Model of a Secure Extensible Language System

Having considered simple models of definition structures for syntax and semantics of base languages and the manner in which these may be modified to introduce extensions, we proceed now to consider overtransparency of these models. We present the results of this investigation as simple propositions followed by stronger assertions which can not, in general be proved, but merely substantiated.

From these simple propositions and assertions, we deduce properties of a model for a secure extensible language system. We emphasise that at this point, we are concerned principally with the theoretic aspect of this model, paying less attention to efficiency or practicality (contrast realisation, below). That is, we distinguish two structures: the logical structure of the model and its physical structure. The primary concern at this stage is to build a good logical structure - one which is secure. A good logical structure, however, does not necessarily imply a good physical structure - one which is efficient.

Proposition 2-2

Suppose the semantics of extensions are hierarchically defined. Consider a metalanguage abstraction, X, which allows subversion of the semantic base (e.g. the base language together with existing extensions) in terms of which some extension is already defined; and a metalanguage, Y, which does not allow subversion, but is otherwise identical to X. Then X is more transparent than Y.

Reasoning

DEFINITION 2-9

Consider a hierarchically defined structure in which one of the (higher level) components L, say is defined in terms of (among others) a (lower level) component M say. We say that the structure is subverted if the meaning of M is altered (while the definition structure of L remains fixed).

Proposition 2-2 is thus trivial, and true by

definition. However, we regard it as important as we consider the form of metalanguage described both unstable and universally overtransparent. This, we cannot prove, but justify as follows.

Consider a hierarchy of extensions (whose semantics are) defined in terms of lower level extensions and base language constructs e.g. as in example 2-29. If the metalanguage abstraction allows the subversion of some node in the structure, then the meaning of each and every extension defined (either directly or indirectly) in terms of this node is altered at one and the same time. We compare this situation to the use of side-effects or undisciplined pointers in programs, or to the modification of programs written in conventional programming languages. Here, experience has shown it difficult to take into account all the implied effects on perhaps remote program parts (cf. secondary effects (Bec 75)). Hence, we consider languages which allow subversion of the semantic base, unstable.

We do not in any case, expect the ability to subvert lower level structures to be particularly useful. Hence, we consider the ability (universally) overtransparent. It would, however, be possible to allow re-definition of features which do not form the definition base of any other extension without incurring the hazards of subversion.

Conclusions from Proposition 2-2

We introduce a conceptual notion to handle this problem of newly defined extensions interfering with the semantic base. It is convenient to consider that extensions are defined by a series of preprocessors. For

example, suppose we have a base language B and processor P_B and we subsequently define extensions E_1, E_2, E_3 (with associated preprocessors P_1, P_2, P_3) in that order. If we have a program $P_{R_{B+3}}$ in the extended language L_{B+3} , it can be translated by activating the preprocessors in reverse order of definition. We illustrate this in figure 2-7.

The preprocessors must be applied strictly in reverse order of definition, in order to prevent subversion of the semantic base.

This conceptual view is useful, as it apparently frees the user from the need to know the architecture of the existing processor(s). It protects the integrity of processors (and not of the language) so that variability, in the extended language, of base language constructs is still possible.

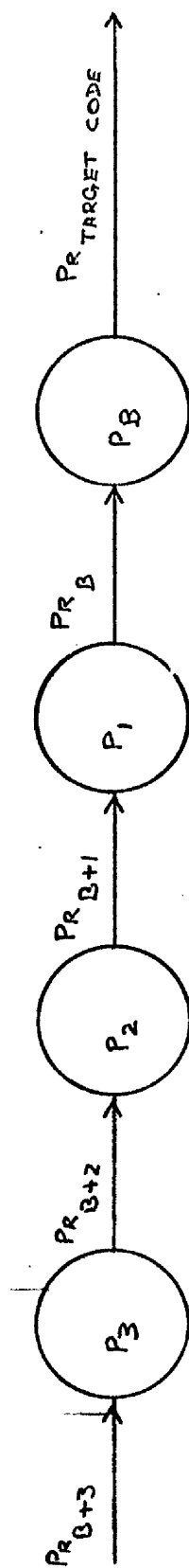
Proposition 2-3

A metalanguage abstraction which allows the definition of extensions without explicitly and fully defining the context of applicability (i.e. defining the syntactic relation relative to other constructs), is more transparent than one which does not.

Reasoning:

This proposition is by definition true, but we are once again more concerned with the stronger and less provable conditions of instability and overtransparency.

While the definition of extensions becomes (apparently) easier if the context of applicability can be left undefined, the extension may then be invoked (in certain cases) in a



KEY

P_i , $i = 0, 1, 2, 3$ INDICATES PREPROCESSOR

PR_{B+i} , $i = 0, 1, 2, 3$ ARE PROGRAMS IN LANGUAGE L_{B+i} , RESPECTIVELY

FIGURE 2-7 EXTENSION PREPROCESSOR

context in which it was not intended to apply. Since it is a non-trivial problem to take into account and examine all contexts in which an extension may be applied and since such contexts may in fact be altered by future extensions (cf. side-effects, pointers, secondary effects), we regard such systems as unstable and universally overtransparent.

In effect, when the context is undefined, the syntax graph is disconnected.

Conclusions from Proposition 2-3

Our model of a secure system is as yet inadequate as we have made no provision for defining context. Each preprocessor must therefore (a) define the context and (b) scan the whole source text of programs to ensure that the extension it implements is applied in the appropriate context only.

Proposition 2-4 is particularly relevant to those (practical) systems in which the derived text consists of base (or extended) language text.

Proposition 2-4

Consider a metalanguage abstraction X, say, which does not specify the complete checking of the form of (substitution) parameters used in the generation of the semantically equivalent (substitution) string; and a metalanguage abstraction Y, say, which does specify complete checking. Then X is more transparent than Y.

Reasoning:

The proposition is true by definition, and indeed X is also overtransparent, since illegal parameters may be

substituted.

Conclusions from Proposition 2-4

This idea is already essentially included in our model. We include this separate proposition as it points to a flaw found in several existing systems cf. section 2.3.

Proposition 2-5

Consider a metalanguage abstraction X, say, in which derived text consists of low level constructs; and a metalanguage abstraction Y, say, in which derived text consists of higher level constructs. Suppose for a particular extension e1, say, it is possible to define the meaning of e1 in Y, using less transparent constructs than is possible in X. Then, for extension e1, we may say that X is more transparent than Y.

Reasoning:

This result follows directly from the correspondence to (simple) programming language abstraction. Considering this correspondence further, we would consider that X is in fact overtransparent for extension e1.

Conclusions from Proposition 2-5

In view of this proposition and the parallel experience with programming languages (cf. section 2.1.3), we would expect to find that there is no single level of abstraction capable of securely or conveniently defining the semantics of every extension. Indeed, we have observed cf. section 1.1 that existing extensible languages provide different means of defining extensions to data structures,

operators, control structures and statements. We would expect our model therefore to include different means for defining different kinds of extensions (as above) and different levels of abstraction to allow definition of different levels of extension (cf. section 2.1.3) - each means of defining extensions and each level of abstraction oriented towards a particular class of extensions at a particular level. (Compare classes of programming language, each class oriented towards a particular problem area, and each programming language in the same class oriented towards solution of different levels of problems within that problem area).

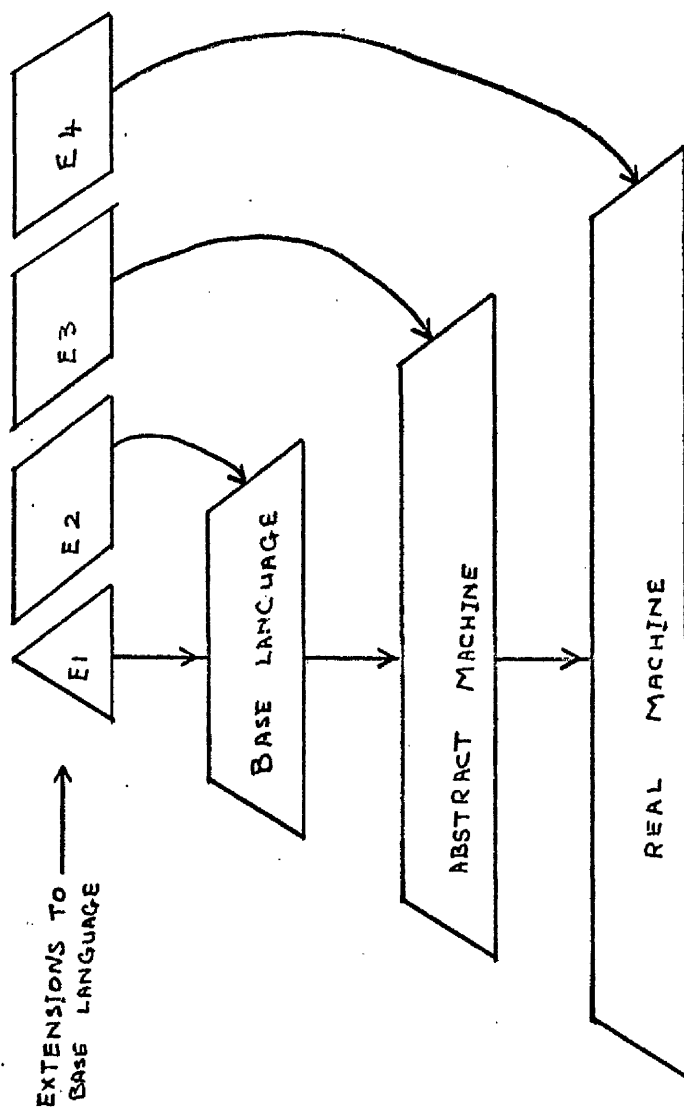
Example:

Many languages provide a separate means for defining data structure extensions. Pascal, for example, allows data structures to be defined at a relatively high level, while Jorrand (Jor 71) allows more flexible definitions, at a much lower level (cf. section 1.1).

Figure 2-8 illustrates some possible levels of abstraction for defining the semantics of syntax extensions. E1-E4 are extended versions of the base language: E1, E2 defined in terms of the base; E3 defined in terms of the abstract machine; E4 defined in terms of the real machine.

Practical Realisation of the Model

The model as described so far is too inefficient and too cumbersome to be of use in practice. We consider two alternative practical realisations which we shall term model M1 and model M2.



E1, E2, E3, E4 ARE EXTENDED VERSIONS OF THE BASE LANGUAGE

FIGURE 2-8 MEANS OF DEFINING SEMANTICS OF EXTENSIONS

Model M1

It is without doubt both inefficient and hard on the programmer to insist on the definition of a complete preprocessor (which scans the complete program text to check context) for each extension defined. One method of dealing with this situation is to define a whole group of extensions or the whole extended language together in a single preprocessor, thus reducing overheads on programmer and machine. An advantage of this system is that it is then a simple matter to exclude undesired/overtransparent features of the base language from the extended language, thus incorporating the result of proposition 2-1. We note that this model is effectively equivalent to bootstrapping an extended language (in terms of lower levels of the language). Thus, we effectively build up a hierarchy of discrete languages $L_B, L_1, L_2 \dots L_n$. We would expect that, in general, L_i will be defined largely in terms of L_{i-1} ($1 \leq i \leq n$).

Model M2

An alternative method of realisation is to allow the base language processor to be modified so that it can translate extensions as well as base language features. There is nothing intrinsically wrong with this approach provided we can ensure that (a) the existing part of the processor remains unaffected and hence (b) the model can still be viewed conceptually as levels of preprocessors. Provided the base language translator is syntax-directed this is relatively easy: preprocessors are effectively defined by modifying the existing grammar; the notion of levels of preprocessor holds valid provided we insist on

strict order of priorities for invocation of extensions, ensuring that no extension is ever re-applied to the substitution string it produces (if any).

However, we consider that this simple realisation is unstable. This model allows existing language constructs to be altered in the extended language. In model M1, this seemed perfectly reasonable, since we were considering the definition of a complete new language. However, in this case, we are incrementally extending the base language: we consider that this form of variability in an existing language is prone to error; the grammar of the extended language becomes hard to define because it may be ambiguously specified, ambiguity being resolved by reference to priorities. (This is important because we have to modify the grammar in order to define extensions in this model).

We consider therefore an alternative means of preventing subversion of the existing processor. If we insist that the grammar describing the base language and the extensions be unambiguous then, there is no possibility of the semantic base being subverted. The priority of extensions is now irrelevant, (since only one production rule is applicable at any one time) and hence can be ignored.

In model M2, we are effectively protecting the language from subversion rather than merely protecting the processor, as in Model M1.

The principal difficulty with model M2, in practice is that of dependence on the translator architecture: it may be necessary to have a good knowledge of the grammar of the existing language, in order to be able to define extensions.

Conclusions on Realisation of the Model

We have developed in the foregoing, two possible models for the realisation of secure extensible language systems. In practice, most existing extensible languages are related to model M2 (cf. below). Systems related to model M1 have tended to be used only for the transportation of languages rather than as a means of providing extensibility as such.

We would expect that in practice, a combination of both models would provide the best system: using a language (cf. abstract machine) hierarchy to define a completely new level of language and to avoid overtransparency of the extended language (cf. proposition 2-1); and using model M2 to allow incremental extension.

Since the implementation of a secure hierarchically defined system of languages causes little controversy, we propose to consider this no further. We observe, however, that in order to satisfy proposition 2-5, the semantics of language L_i should, as far as possible be defined in terms of language L_{i-1} . We would expect considerable inefficiency to arise here unless implementation is handled with care (for example, passing identifier tables and post-lexically analysed text from processor i to processor $i-1$ cf. section 5.1).

In the following sections, we consider the security of existing extensible language systems by relating them to model M2: we show that there is considerable room for improvement. Finally, in this section, we direct our attention towards proposals for a more secure extensible language system by considering model M2.

2.3 Security of Existing Extensible Language Systems

We attempt to highlight the insecure features of existing extensible language systems, referring in particular to principal or representative systems. We are concerned chiefly with insecurity inherent in the design of the system and therefore concentrate mainly on the overtransparency of the metalanguage used to define extensions. All the systems considered are related to model M2 of the previous section.

We find it convenient to discuss systems under a suitable classification scheme. Solntseff's classification scheme (cf. section 1.3), however, is not ideally suited to our purposes, particularly with regard to groups B, C and D. Within these groups, binding-time is irrelevant to security: it is more appropriate (cf. foregoing discussion) to consider the means of defining semantics. We propose therefore to reclassify systems in these groups:

- (a) Type BCD 1 : those systems within groups B, C and D
which define the semantics of extensions
in terms of base or extended language
substitution text.
- (b) Type BCD 2: those systems within groups B, C and D
which define the semantics of extensions
in terms of special purpose abstractions
e.g. abstractions oriented towards the
definition of data structures.

Figure 2-9 shows the classification of the principal systems under the modified scheme.

Type A

There are many variations of the simple macro-

Classification System	Type A	Type BCD1	Type BCD2	Type E
GPM (Strachey)	x			
Trac (Mooers)	x			
Stage 2 (Waite)	x			
Limp (Waite)	x			
McAlgol (Bowlden)	x			
- (Grant)	x			
Imp (Irons)		x		x
AEPL (Milgrom)		x		x
PPL (Standish)		x	x	x
- (Cheatham)	x	x		
- (Leavenworth)		x		
GPL (Garwick)		x	x	
Proteus (Bell)		x	x	
Algol C (Galler)			x	
- (Schuman)			x	
Mad (Arden)				x
Lace (Newey)				x
ECT (Solntseff)				x
Snap (Fisher)		x		x

FIGURE 2-9 Classification of principal extensible language systems.

processor, such as GPM (Str 65), Stage 2 (Wai 70), Trac (Moo 66). These systems are preprocessor systems in which (since extensions are bound before parsing, and possibly before lexical analysis) extensions take priority over base language constructs and a new extension (usually) takes priority over existing extensions. Basically, the notion of macroprocessors seems to conform quite closely to our theoretic model of a secure system. However, as far as security is concerned, there are some basic flaws in the design.

- (1) Extensions are defined without binding them to existing structures; as a result, an extension may be applied in the wrong context (cf. proposition 2-3).
- (2) Although new extensions take priority over existing extensions and base language constructs, the stages of pre-processing are not rigidly enforced. The result is that subversion can in fact occur (cf. proposition 2-2).
- (3) Few macroprocessors check the legality of parameters used in substitution strings (cf. proposition 2-4). The result is that substitution of illegal parameters frequently occurs; this may lead to chaos in further expansions.

The syntax and pragmatics of the metalanguage in macroprocessors is usually primitive and hence insecure:

- (1) Notation used in macro calls is excessively primitive e.g. poor mnemonic notation, excessive use of bracketing (leading to instability cf. section 2.1.4).
- (2) Notation for referring to substitution parameters frequently lacks a mnemonic or natural form.

Some features of the metalanguage tend to be unnatural and more closely related to the formal model of macro-processors (i.e. Markov Algorithms (Weg 68)) e.g. macro-time statements to allow iteration in GPM.

Distributed-name macroprocessors such as Limp (Wai 67), McAlgol (Bow 71), ML/1 (Bro 67) and Grant's system (Gra 71) allow an improved form of macro call (though still fairly simple) and improve macro-time statements. However, they do nothing to remove the basic flaws inherent in macroprocessors.

The discussion on realisation of our theoretic model shows that we do not consider macroprocessors a viable means of achieving a secure extensible language system.

Type BCD 1

Typical in this group are the mechanisms designed by Irons (Iro 70), Milgrom (Mil 71) and Standish (Sta 69). Extensions are defined by modifying a context-free grammar (Chomsky Type 0 in the case of Milgrom's system, AEPL) and by adding new rules to the grammar.

Since priority of extensions is not strictly enforced and since, in general the non-ambiguity of a context-free grammar is undecidable (Hop 69), it is in fact possible for the semantic base to be subverted, either accidentally or otherwise (cf. proposition 2-2). In AEPL, there is even greater possibility of subverting the base, since rules of the grammar (describing the semantic base) may be explicitly deleted or replaced.

Furthermore, Irons and Milgrom exacerbate the problem by permitting the association of user-defined priorities (other than those proposed in proposition 2-2) with the

various extensions, thus allowing further scope for subversion.

In his paper, Irons points out that unintentional ambiguities are easily introduced and that correction of this occurrence may be very difficult. In some versions of the Imp language, ambiguities have arisen which have been so annoying to repair that they have simply been left in. (There are, however, situations in which ambiguity can be helpful cf. section 2.4).

Since the problem of ambiguity does exist in these systems, the user's task is made more difficult. It is probably necessary for the user to have an intimate knowledge of the grammar defining the semantic base, in order to avoid the pitfalls of introducing ambiguity.

Irons attempts to reduce the need for grammar knowledge by the use of defaults, with automatic infilling of syntactic classes. However, as Irons himself observes, this process is itself prone to error and can lead to definitions which were not intended.

The notation used to specify grammar rules is usually BNF: this notation tends to be longwinded since options or alternatives, for example, have to be written out longhand.

Neither Imp (Irons' system) nor PPL (Standish's system) has much extension-time capability. Imp has no data types (McIlroy (Sch 71a)).

Bell's system, Proteus (Bel 69), also belongs to this group. However, it fails to define the context in which extensions may be applied, and hence suffers from the principal defect of type A systems (cf. proposition 2-3).

In effect, the user can to some extent influence the context of applicability of an extension by associating a priority with it; but, far from redeeming the situation, this does in fact aggravate it, since subversion of the semantic base also becomes possible (cf. proposition 2-2).

Other systems in this group tend to be fairly restrictive. The systems proposed by Garwick (GPL (Gar 68)) and Leavenworth (Lea 66) do use a deterministic grammar (cf. Model M2), but they permit extension of the syntactic classes statement and expression only, and that in a fairly restrictive form.

The notation for definitions is improved in Leavenworth's system by the use of special symbols to indicate options or alternatives (i.e. an extended BNF notation), but is impaired by an awkward method of referring to substitution parameters, reminiscent of that used in simple macroprocessors.

Cheatham's system (Che 66) also avoids ambiguity by using a precedence grammar. We would expect, however, that it is not, in general easy to extend a precedence grammar without having a good knowledge of the existing grammar. No conditional expansion is possible.

Type BCD 2

Usually, this form of extension is used only for data structure or operator extensions (cf. section 1.1), although Standish (Sta 69) has used it to include extension of control structures. (We do not have sufficient information to discuss this latter aspect further).

Typical in this group are the systems by Galler and Perlis (Algol-C (Gal 67)), Garwick (Gar 68), Standish

(Sta 69) and Bell (Bel 69). The Standish system is used in many languages which are not normally regarded as extensible e.g. Algol 68 (Wij 69), Pascal (Wir 70).

In these systems, redefinition of items in the same scope is usually outlawed and hence there is no question of subversion (cf. proposition 2-2). If an item is redefined in an inner scope, reverse order of priority (cf. model M2) holds. The context of applicability is pre-defined and therefore constitutes no problem (cf. proposition 2-3).

Algol C allows only simple data structure extensions, as the sole means of defining new structures from primitive structures is by use of an array constructor. Galler and Perlis pay considerable attention, however, to the generation of optimum code from operations on defined data structures.

The Imp system is insecure as it allows typeless data structures (cf. section 2.1.3). The Bliss language (Wul 70) provides no type checking at all.

Algol C and GPL permit too much flexibility in the manipulation of pointers (cf. section 2.1.4).

It is interesting to note therefore that extensions of data structures and operators in this category are, in general, defined with relative security. This is perhaps due to the fact that these are regarded more as an integral part of the language rather than extensions to the language: and all restrictions of the language itself therefore apply also to the "extensions". This is perhaps even more true of the more recent notion of clusters in which data structures are defined together with the associated operations (Lis 74; Lis 75).

Type E

Typical systems are those by Newey (Lace (New 68)), Standish (PPL (Sta 69)), Arden (Mad (Ard 69)), Fisher (Snap (Fisr 73)), Solntseff and Yezeriski (ECT (Sol 74)).

Extension mechanisms in this group are purposely presented on a very low level in order to allow maximum flexibility and maximum efficiency. Considerable knowledge of both language and translator architecture as well as the target machine code is usually necessary to the definition of extensions in this kind of system.

Since extensions are not hierarchically defined, the problem of subversion is less critical, and, indeed, the ability to subvert or to contract the language and translator may be viewed by some as desirable cf. (Sol 74).

Snap and ECT are based on context-free grammars. We might expect that flexibility would be little impaired if a deterministic grammar were employed to reduce difficulty. Newey and Arden, for example use precedence grammars.

Most systems in this group attempt to define the semantics of extensions in terms of an abstract machine code, where possible, in order to reduce transparency.

Some systems e.g. ECT, allow individual rules of the grammar (which describes the language) to be explicitly deleted or replaced, as well as allowing explicit deletion and replacement of compiler routines. The flexibility allowed to the user is thus so great that we wonder whether it can reasonably be regarded as anything other than an experimental system in the hands of a competent few.

Conclusions

We conclude that while Type A extensions cannot be made secure in practice, there is considerable scope for improving extensions of type BCD 1; existing systems of type BCD 2 seem relatively secure; systems of Type E are very transparent, but for some extensions, this is regarded as necessary, though we question the advisability.

Standish's system, PPL, combines the three methods of extension, types BCD 1, BCD 2 and E most successfully (cf. proposition 2-5) although we have observed that there is room for considerable improvement, particularly with regard to type BCD 1.

No existing system takes any steps to avoid overtransparency of the extended language (cf. proposition 2-1).

2.4 Proposals for a Secure Extensible Language System

We recall from section 2.2 that we expect the most successful system to combine model M1 (bootstrapping) with model M2. We recall also that we do not intend to pursue the idea of abstract machine hierarchies any further in this dissertation.

The discussion in section 2.3 has shown that no existing system matches the ideals of our secure model. Standish's system, PPL comes closest to providing different levels of defining semantics, but is overtransparent, particularly with regard to extensions in group BCD 1. We consider that the definition, in PPL, of extensions in groups BCD2 and E is relatively secure and therefore we need consider these no further. We direct our attention instead towards definition of extensions in terms of a substitution strings i.e. type BCD 1.

We consider:

- (a) Syntactic Metalanguage
- (b) Semantic Metalanguage
- (c) Parameter Substitution
- (d) Declaration of Extensions

(a) Syntactic Metalanguage

We consider the most suitable syntactic metalanguage from the point of view of security. We recall, that since we are considering model M2, that the grammar of the language is already restricted to the deterministic (i.e. unambiguous) subclass of grammars.

We find it interesting to note, at this point, that while Irons acknowledges the problem of unintentional ambiguity, he observes that it can in fact be useful in building certain extensions. It has previously been noted by Floyd (Flo 67) and others that, under certain circumstances, non-deterministic algorithms are easier to specify than the equivalent deterministic systems. We propose, however, to resist the lure of using non-deterministic means to specify deterministic systems until such time as the difficulties involved in correctly constructing non-deterministic algorithms (cf. security, complexity) are better understood (cf. section 5.1). Griffiths (Grif 74) observes, in any case, that with a deterministic method of syntax analysis it becomes possible to execute semantic routines during the syntax analysis process, thus saving a pass of the source text.

We want to further restrict our grammar to a subclass (if it exists) in which:

- (1) There is a relatively efficient algorithm to determine

whether or not a given grammar belongs to that subclass.

- (2) It is in practice possible to determine merely by inspection whether or not a given grammar belongs to that subclass (so that extensions are relatively easy to define).
- (3) The subclass of grammars is sufficiently powerful to allow a natural notation to be described without the need for severe contortions of the notation.

The most obvious first move would seem to be to restrict the grammars to the subclass deterministic type 1, since there has been little work carried out in the area of practical type 0 recognisers. It is still possible to handle such context-sensitive restrictions as type checking within this subclass (Kos 71). Some work on the construction of practical recognisers for deterministic type 1 languages has been carried out (Wat 74). However, since this research is in its infancy, and since it is not clear how easy it will be in practice to construct and specify such grammars, it would appear prudent to avoid their usage in this research at this stage (cf. section 5.1).

We consider instead, the subclass of grammars known as deterministic context-free or LR(k) grammars (Hop 69). Some of the difficulties of introducing extensions are caused by the fact that analysis is effected partly by semantics rather than solely by syntax (Ard 69) leading to dependence on translator architecture. Thus, a disadvantage of using context-free syntax is that type-checking is carried out by semantics. As a result, in order to define operator extensions, either the translator architecture must be known and altered or a special purpose

means of defining operator extensions must be defined. The former method is too low level and transparent; the latter method is used in all existing systems. The same problem arises in attempting to extend data structures (or any other features which require context-sensitive checking), although it is in any case necessary for other reasons (cf. section 1.1) to introduce a special purpose scheme to define new data structures.

LR(k) Grammars

De Remer (Rem 71) points out that the general LR(k) recogniser is still fairly slow; while Anderson (And 71) notes that it is hard to determine by inspection whether or not a grammar is LR(k), although an algorithm does exist. We consider precedence or extended precedence grammars hard to extend or recognise by inspection, although recognisers are relatively efficient.

We propose to adopt the subclass of grammars LL(1) for the following reasons:

- (1) The grammar is fairly simple (cf. section 2.1.4 (Hoa 73)) and it appears as though simple extensions can be defined principally from knowledge of the language and without the need to know the grammar of the language (contrast Imp, for example).
- (2) McKeeman (McK 74) suggests that it is in fact fairly easy, with a little practice, to specify LL(1) rules. Tentative investigations (cf. Appendix C) suggest that, in practice, it is relatively easy to determine simply by inspection, whether or not a given grammar is LL(1). There is an algorithm to determine whether or not a given grammar is LL(1) (Grif 74).

- (3) Efficient recognition is possible using the method of recursive descent.

We must consider whether the subclass LL(1) is too restrictive. The programming language Pascal has, in our opinion, a good and natural notation. Its syntax can be defined in terms of an LL(1) grammar without any obvious compromise of language design or notation. Pascal was in fact designed to have an LL(1) syntax; some small amount of language manipulation was necessary to this end, but this was neither excessive nor difficult. We support these claims concerning LL(1) grammars in appendix C.

Notation

We noted in the previous section that Imp, AEPL and PPL use a fairly primitive notation for specifying the syntax of extensions, namely simple BNF. A more promising and natural method of handling such specification might be to use an extended form of BNF notation, to allow more natural definition of optional and alternative syntactic forms. This is a development of the notation used by Garwick and Leavenworth. The pattern structure of string processing languages (Bob 68) might be usefully employed to denote extended BNF rules; indeed, Grant (Gra 71) has employed Snobol 4 patterns for this purpose.

(b) Semantic Metalanguage

Since we are considering extensions of type BCD 1, the semantic metalanguage is a substitution string consisting of base or extended language text. For simple extensions, pure substitution text is sufficient for definition purposes. However, for more complex extensions, it is necessary to provide extension-time statements (cf.

section 1.3). In some existing systems, there is no provision for extension-time statements cf. Cheatham (Che 66), in others, relatively fixed and restricted forms are allowed cf. Leavenworth (Lea 66). We consider it appropriate to allow a fair amount of flexibility in this area and propose to allow a wide variety of extension-time statements by allowing the use of constructs of a complete programming language at this point. Since we are handling strings at this level, a string processing language would seem eminently suited to this purpose.

Thus far, then, we have proposed a string processing language as the basis for our extensible language mechanism. In order to prevent subversion of that section of the string processing language program which defines the semantic base of the language to be extended, we must prevent extension-time statements altering the procedures or the values of variables in the existing program. We can however allow execution of procedures and read access to variables.

(c) Parameter Substitution

We consider the kind of parameter substitution required in the generation of a substitution string. We consider first, the parallel situation in macroprocessing. Brown (Bro 71) considers that three kinds of parameter substitution are commonly useful:

- (1) immediate-value substitution
- (2) delayed-value substitution
- (3) name substitution

However, because of the large overheads involved, most macroprocessors allow only one type of parameter substitution.

The distinction between nested macro-calls and parameters tends to blur when dealing with the more flexible forms of "macro-call" at play in our extensible language system. In the simple form of macro-call, there is a strict one-to-one correspondence between formal and actual parameters; the actual parameters may be macro-calls themselves. In the more flexible system at hand, a macro call may be nested at any point within another macro call (it need not correspond to an actual parameter as such). We consider an example to illustrate this point:

Example:

Suppose we have a "syntax macro" of the form:

```
"FOR <variable> := <expression_1> STEP <expression_2> UNTIL
    <expression_3> DO <statement> "
```

We introduce a new syntax macro:

"T0 <expression> D0 <statement> "

with the meaning

"STEP 1 UNTIL <expression> DO <statement> "

An example of the nested call of these two macros is:

"FOR I := 1 TO N DO A [I] := \emptyset ".

In order to ensure the correct expansion of this statement the inner macro call must be evaluated first. However, from the definition of the first macro, there is no clear indication as to whether the phrase:

"TO N DO A [I] := \emptyset "

is to be regarded as a parameter or a nested macro-call. If it is regarded as a parameter, then it must be called by value to ensure correct expansion.

To illustrate further difficulties, we consider

another example:

Example:

We assume that the macro

```
"FOR <variable> := <expression_1> STEP <expression_2>
    UNTIL <expression_3> DO <statement> "
```

is defined by the substitution string

```
"BEGIN
  <variable> := <expression_1> ;
  WHILE <variable> < <expression_3> DO
    BEGIN
      <statement> ;
      <variable> := <variable> + <expression_2>
    END
  END "
```

We consider the parameters:

If <variable> recognises a simple base language variable, then value substitution is impossible (if not meaningless) in this context, because of the difficulty of specifying semantic action (since generation of code to store an expression in an assignment statement must be delayed until the expression has been evaluated). Only name substitution is appropriate.

In the case of the parameter <statement>, either name or value substitution is meaningful, although value substitution may be inefficient if the statement is long.

Provision of an explicit mechanism for handling several kinds of parameter substitution seems, therefore, insecure and prohibitively difficult to organise and implement.

We attempt to simplify the usage problem:

- (a) Name call is automatically invoked for base language constructs.
 - (b) Value call is automatically invoked otherwise.
- For less common usage, the user has the capability of organising his own form of substitution using the features of the string processing language. (We feel justified in making more common usages easier cf. section 3.1.1.1, even at the risk of making some less common usages harder.)
- (d) Declaration of Extensions

We observe that while model M2 excludes the possibility of re-declaration of extensions within the same scope, it does not disallow re-declaration of extensions within inner scopes i.e. it does not exclude the possibility of associating extensions with block structure. However, we consider that excessive variability of the meaning of language constructs within a particular program is prone to error, and we do not intend to consider the idea further. We note in contrast, however, that most systems of group BCD 2 do allow the definition of new data structures to be associated with block structure.

2.5 String Processing Languages

We have indicated an interest (cf. section 2.4) in the so-called pattern-directed string processing languages. A survey of many of the existing string processing languages is to be found in Bobrow (Bob 68) and Sammet (Sam 69).

Pattern-directed string processing languages have a common basic structure related to the Markov algorithm

(Weg 68), and hence, are theoretically equivalent to the Turing Machine (Hop 69).

A pattern consists of a sequence of alternate and optional elementary patterns. Elementary patterns may be string literals, string variables with previously assigned values or pre-defined variables which denote a particular class of substrings.

During pattern matching operations, a specified string is compared to a pattern. If the string matches one of the set of strings defined by the pattern, then it is transformed according to a format associated with the pattern.

We proceed in this section to consider the suitability of the various existing string processing languages to forming the basis of our extensible language mechanism. We recall that we have already determined that patterns should recognise LL(1) structures (cf. section 2.4) and that our extensible language mechanism should be compact, efficient and secure (cf. section 2.0).

Patterns in Panon 1B (Car 66) are in the form of so-called "Generalised Markov Algorithms". We consider that patterns are presented on too low and overtransparent a level to be useful in our extensible language system, (cf. section 3.2.2.3). Patterns in Panon 1B are capable of representing Type 2 grammars.

The form of patterns present in Axle (Bob 68) and in Floyd's Production Language is very similar to that in Panon 1B and so we reject these representations for the same reasons.

Since the capabilities of Snobol 4 (Gris 71) and Comit (Yng 63) are in fact similar, and since, in most cases,

the notation in Snobol 4 is by far the clearer, we consider Snobol 4 only. Ambit (Chr 65) is similar in spirit to Snobol 4 but borrows from Algol 60 for its overall program structure. With even a brief survey of string processing languages, it has become abundantly clear that Snobol 4 is the only language capable so far, of offering us the basis for the kind of extensible language system we desire. Snobol 4 was in fact the basis of Grant's extensible language system (Gra 71).

Snobol 4

We consider the viability of a Snobol 4-like language as the basis of a secure extensible language system. We consider first security and then efficiency.

Security of Snobol 4

(cf. section 2.1)

Features:

As indicated in the previous section, we would like to restrict the grammar of the language to LL(1). Snobol 4 patterns allow recognition of a class of grammars wider than context-free (cf. Gimpel (Gim 73)).

Control structures in Snobol 4 are very primitive (cf. (Gra 71; Knud 74)) and correspond closely to a form of goto-statement. Griswold (Gris 74) has suggested some embellishment, but this is in the direction of improving iteration structures and does little to reduce (over)-transparency in other directions.

We would expect that the ability to dynamically change the type of a variable or to re-define functions, procedures and operators at run-time is liable to lead to confusion.

Syntax and Pragmatics:

We feel that the notation used in Snobol 4 is in many ways poor. In some cases, a single symbol denotes different operations in the same construct, according to local context. For example, the "=" symbol may denote assignment or part of a string-replacement operation; the "space" symbol may denote concatenation or pattern matching or part of a string-replacement operation. Griswold (Gris 74) agrees that this may lead to confusion in learning of the language and also results in a non-uniform parsing strategy.

We consider that use of the symbols "= " to denote the null-string assignment "=NULL" is inviting error (cf. Appendix G). The use of default options is similarly prone to error.

We would like explicit declaration of variable identifiers and their corresponding types (cf. section 2.1).

We feel that, perhaps due in part to the power and flexibility of the language, that Snobol 4 does not attain the natural and elegant structure of, for example, Algol 60.

Efficiency of Snobol 4

The design philosophy of Snobol 4 emphasises flexibility at run-time (Gris 72) and avoids features that bind components of a program at compile-time. As a consequence of this philosophy, functions, procedures and operators may be re-defined and patterns constructed at run-time. Efficiency was not then of paramount consideration. In contrast to this philosophy we would like in our system to fix, at compile-time, as many components of a program as possible.

Waite (Wai 73) points out that the overheads of constructing patterns in particular, are heavy (because of the need to copy and adjust addresses of the machine representation) and that patterns tend to be heavily used. We should aim therefore, to avoid unnecessary flexibility of patterns at run-time (cf. section 3.2.2.3).

Since Snobol 4 patterns are not deterministic, back-up may therefore be necessary, thus further increasing overheads. This is particularly true if some semantic action taken as the result of a successful subpattern match has to be undone - indeed, this is not in general possible, and semantic action must therefore be delayed to avoid this occurrence.

The facility in Snobol 4 allowing indirect referencing involves large run-time overheads (Wai 73) in ensuring that multiple copies of string values do not arise. We would not expect this facility to be of particular value to our application, but the penalty is incurred whether or not the facility is actually used.

We conclude, therefore, that it should be possible to build a language much better suited to our purposes.

In the following chapter, we attempt to design such a language by drawing on the experience of Snobol 4, in particular, and attempting to blend with it some of elegance and structure of the Algol-like languages.

2.6 Conclusions

In this chapter, we considered the design of a secure extensible language system. We first considered the meaning of security with respect to simple programming languages, introducing the notion of overtransparency, in

particular. Subsequently, we considered the importance of security in extensible language systems, and developed a model for a secure extensible language mechanism. We demonstrated that no existing system meets the ideals of this model, and finally proposed a new system based on a string processing language.

CHAPTER 3

DESIGN OF A SECURE STRING PROCESSING
LANGUAGE3.0 Introduction

In this chapter, we set out to design a string processing language (Snip) through which we may realise an extensible language system as envisaged in the previous chapter. We propose to design a base language to facilitate portability and to enable us to build upon and adapt this language by self-extension.

While it is essential that the syntax, pragmatics and semantics of a language be developed hand in hand, we find it convenient, in presentation, to describe (as far as possible) semantic aspects separate from both syntactic and pragmatic aspects.

Section 3.1 considers general design principles, and section 3.2 the design of Snip itself.

3.1 General Principles of Language Design

The first part of this section considers principles of design for language features in general and subsequently for base languages; it discusses the balancing of conflicting design criteria with particular reference to security. The second part of this section considers the design of syntax and pragmatics.

3.1.1 Design of Features3.1.1.1 General Programming LanguagesBackground

Whatever the proponents of language design may say, this subject remains very much an art or serendipity rather

than a science proscribed by a thesis of predictions or recommendations. Each language designer has his own point of view and the consensus of opinion is debated, much as by any competent artist or musician. What is important is the extent to which designers have separated out and labelled the distinct, often conflicting, properties which a language may or should have. The "art" of language design concerns the choice of particular emphasis and combination of each of these issues.

Language Properties

The following language properties are widely considered.

- | | |
|--|-------------------|
| (1) Modesty (Simplicity) | (6) Orthogonality |
| (2) Elegance | (7) Involution |
| (3) Efficiency | (8) Security |
| (4) Generality | (9) Portability |
| (5) Constructs for
Common Special Cases | |

(1) Modesty

Dijkstra (Dij 72), McKeeman (McK 74) and Hoare (Hoa 73) consider that a programming language should be modest (unambitious and simple). We define a programming language to be modest if it is problem-oriented.

(2) Elegance (Dij 72)

We define a programming language to be elegant if it is small and simple.

(3) Efficiency

A programming language is efficient if it allows

generation of fast and compact code by a fast and compact compiler (Hoa 73; Wir 74). In this context, however, we refer principally to run-time efficiency.

(4) Generality

Generality implies that the language is rich and powerful (and equivalent to a recursive function subset) e.g. union of all problem-oriented languages. Generality may also apply to individual language constructs.

(5) Constructs for Statistically Common Special Cases (CSC)

Wirth and Hoare (Hoa 66; Hoa 73) propose that a language should include purpose-defined constructs for (statistically) common special cases.

(6) Orthogonality

A programming language is orthogonal if its feature sets have independent, and independently understood purposes cf. (Wij 69).

(7) Involution

A programming language is said to be involuted if its features, once defined, may be used anywhere they make sense cf. (McK 74).

(8) Security

A programming language is secure if it is resistant to error (Hoa 73).

(9) Portability

A programming language is said to be portable if it is easily transported in terms of itself cf. Poole and Waite (Poo 73).

Primary Aims Proposed by Various Designers

We consider areas of general agreement and contention regarding the above-noted issues together with some examples. Since many of the properties are not mutually compatible, such discussion essentially consists of the identification of primary and secondary aims, the secondary aims being applied only when they support rather than conflict with the primary aims.

We consider:-

- (1) Generality vs Modesty and Elegance
- (2) Generality vs CSC and Efficiency
- (3) Orthogonality and Involution vs Modesty

(1) Generality vs Modesty and Elegance

Dijkstra (Dij 72) considers that it is not the richness and power of a language which is important, but its modesty and elegance, while McKeeman (McK 74) and Hoare (Hoa 73) similarly suggest that utmost simplicity must be the overriding criterion. McKeeman considers over-ambition the most common error in language design.

(2) Generality vs Constructs for Special Cases and Efficiency

CSC

Wirth and Hoare (Hoa 66) point out that the efficiency of a language is particularly sensitive to the trade-off between generality of individual constructs and the restriction of constructs to allow optimisation of special cases: it is frequently the case that (optimisable) special cases of constructs are heavily used, while full generality rarely is. The simplest means of achieving this

optimisation is by forcing the more general, but rarely used constructions to be explicitly programmed in terms of lower level constructs.

Example (a)

The for statement in Algol 60 is very general: consider the statement

"FOR V := A STEP B UNTIL C"

where A, B and C are expressions. The expressions B and C are not bound at block entry and their values are recalculated at each iteration. This prevents register optimisation of the more common special cases in which the expression values remain unchanged. For-statements of Algol W, 68 and Pascal are restricted to allow this optimisation.

(b) In section 2.5, we noted that the generality of Snobol 4 patterns prevents optimisation of several special cases.

Similarly, Hoare (Hoe 73) notes that it is important to consider the choice of operators for large data objects carefully. Arithmetic expressions involving small data objects can be evaluated very efficiently, because intermediate results are small enough to be retained in high speed registers, or recovered from main store in a single operation. When the operands are too large for such a course of action, it becomes much more efficient to use updating operations, when possible. This may considerably reduce the amount of work involved, but at least reduces storage requirements by using the space occupied by one of the operands to store the result and/

or intermediate results.

Example:

Consider a string assignment of the form:

"A := A concat B"

Provided append operations of this form are denoted by a special case construct such as

"A append B"

the value of A may be updated merely by copying string B.

Efficient Object Code

Hoare (Hoa 73) considers the beliefs that efficiency is no longer important (a) because of cheaper and faster hardware or (b) because efficiency can be regained by use of a sophisticated optimising compiler to be fallacies. He suggests that it is far better for users themselves to take advantage of any such freedom to write better structured and clearer programs (cf. security). The only solution is to design a language which is (a) sufficiently expressive that most optimisations can be expressed in the language itself (cf. CSC above) and (b) simple, clear, regular and free from side-effects so that a general language-independent optimiser can simply translate an inefficient program into a more efficient one (with guaranteed identical effects) written in the same source language.

(3) Orthogonality and Involution vs Modesty (Simplicity)

Wijngaarden (Wij 69) has suggested that language constructs should be chosen to have clearly independent properties and uses, which may be independently understood.

Example:

Algol 60 has two kinds of for-statement essentially of the form:

"FOR <variable>:=<expression>STEP <expression>UNTIL <expression>
DO ... "

and

"FOR <variable> := <expression> WHILE <boolean expression> DO ... "

These two forms have overlapping areas of applicability, and hence, are not orthogonal.

If a language is involuted then a feature, once introduced may be used anywhere it makes sense (McK 74).

Example:

In Algol-like languages, arithmetic expressions may be used as array subscripts or actual parameters.

Hoare (Hoa 73), however, considers that aims of orthogonality and involution insofar as they contribute to overall simplicity are an excellent means to an end, but that as a substitute for simplicity they are very questionable indeed.

Example:

Constructions of the form

"X := IF N = 3 THEN 1 ELSE GOTO L FI"

in Algol 68.

Wirth (Wir 74) and McKeeman (McK 74) also have reservations concerning the property of involution. They observe that careful attention should be paid to the use of two or more language features in combination. It is quite possible for certain features to be acceptable individually, while posing intractable problems when combined. McKeeman notes that the addition of features to

a language is thus by no means a linear process as far as the translator is concerned: in extreme cases, the addition of a single feature could double the size of the translator.

Example:

Individually, dynamic arrays or records cause no severe problems. However, this is not so if records are allowed to include fields which are dynamic arrays: the offset of fields from the record base is then no longer constant, but varies with the size of the dynamic array fields.

The Author's View

We have already indicated that we consider security to be our primary objective and have considered the possible effect on efficiency and more particularly, on generality (cf. section 2.1.3). With the passing euphoria over "universal languages" (cf. section 0.1) and the increased emphasis on small problem-oriented languages, it is in any case probably widely accepted that complete generality is neither desirable nor attainable.

Like Hoare, we consider the aims of orthogonality and involution subsidiary to that of security.

Use of purpose-defined constructs for common special cases would appear the most cost-effective method of increasing security. Knuth's studies of use-statistics for Fortran programs (Knut 73) and similar studies by Wichmann (Wic 73) and Ibrahim (Ibr 74) on Algol 60 and Algol W programs (respectively) lead us to believe that in all three languages, a small number of very simple constructs are very heavily used, while most other constructs are rarely used. Wirth (Wir 75), however, appears to

contradict the view of Wortman and Ibrahim that all languages will exhibit similar patterns of use statistics, suggesting that we might expect different results when structured programming techniques are used. We would still expect, however that a small number of constructs will be heavily used although the pattern of use may change considerably. It seems likely therefore that special case constructs are likely to be of great value, and that in optimising complex constructions, language designers have, in the past, tended to optimise the wrong features.

3.1.1.2 Base Languages

Having considered design principles for general programming languages, we consider how these principles relate to the design of base languages. We define a base language in terms of aggregates and primitives as follows: We define primitives to be the (minimal) base set of operations out of which all language constructs may be synthesised.

An aggregate is a combination of primitives.

A base language consists of the set of primitives together with a small set of aggregates (excluding those for constructing transparency-reducing higher levels).

General

Base languages are by no means unique to the study of extensible languages. They have, for example, been considered by Poole and Waite (Poo 73) in the design of hierarchies of abstract machines for portability, by Mitchell (Mit 70) in the bootstrapping of interactive programming

systems, and by Iliffe (Ill 69) in the design of improved real machines. While our aims in providing a base upon which to build extensions and our criteria for evaluating such a base will differ from those in the above systems, there is an area of commonality upon which we can build and adapt as necessary.

While some work has been carried out on the design of bases for extensible languages, there has been remarkably little discussion of this in the literature (cf. (Sol 74)).

Design Criteria

The criteria for design of a base language which is to be extensible include all those for general design together with the additional aim of extensibility.

A base language is extensible if it can be extended to include desired higher level constructs.

As we have observed, these design criteria are not mutually compatible and indeed are in some cases highly conflicting. In appendix H, we analyse these criteria to determine which of these quantities are

- (a) the same,
- (b) similar,
- (c) dissimilar,
- (d) independent of each other.

The aim is to simplify the design problem.

A matrix indicating how these quantities correlate with each other was constructed. This matrix was checked by quantifying the degree of correlation and correcting the matrix for inconsistencies. This process was carried out iteratively. The quantities involution and orthogonality

are shown to be the same. We therefore consider orthogonality alone in the remaining part of this discussion.

The results are summarised in figure 3-1. This suggests that major areas of conflict are

- (a) Generality vs Efficiency and Security
- (b) CSC vs Generality
- (c) CSC vs Elegance and Orthogonality.

We make the following observations by way of support or explanation of these results.

(1) Extensibility vs Efficiency

Base language design concerns the criterion of extensibility; overheads due to the extension mechanism itself do not therefore enter the discussion.

(2) Security vs Efficiency

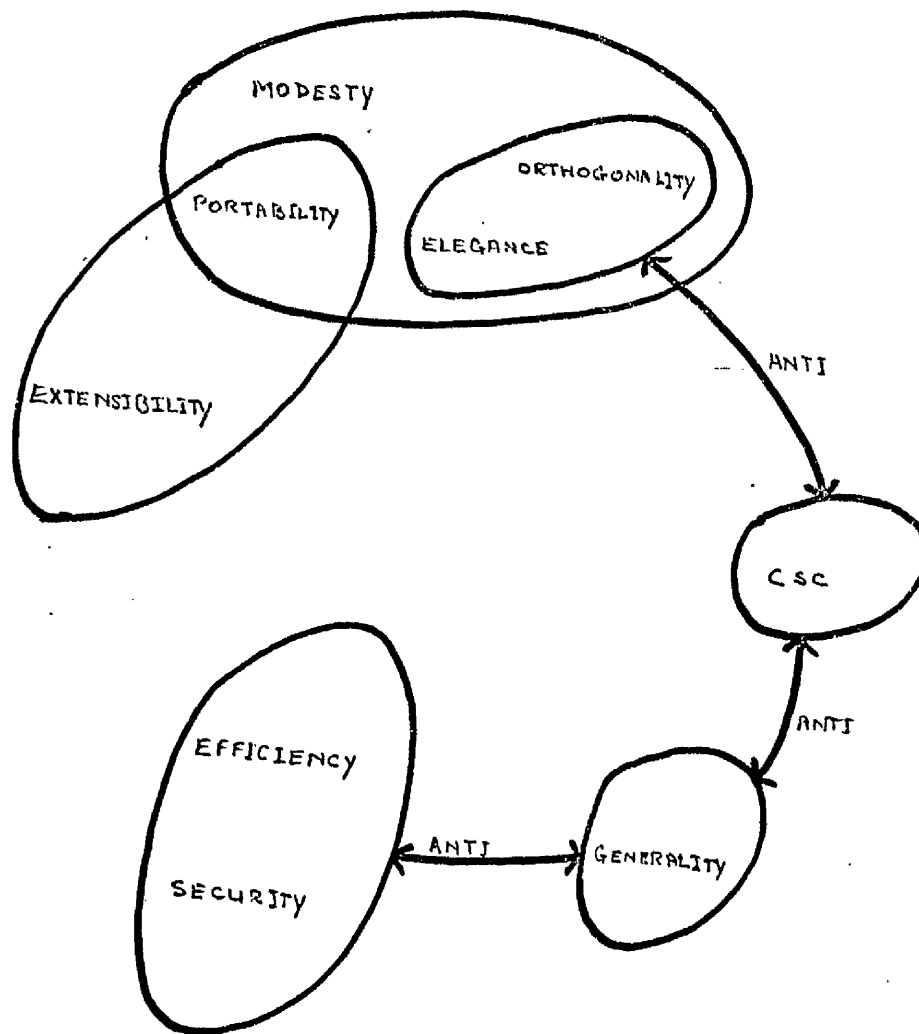
Since at this point we are considering language features, security concerns only overtransparency and instability. We observed (cf. section 2.1.3) that security is greater when the language is problem-oriented. Owing to overheads of the extension mechanism, efficiency will be greatest under the same conditions.


(3) Portability vs Security

We assume that portability is achieved through a hierarchy of abstract machines (discussed in chapter 4).

(4) Security vs Extensibility

Since we have already decided that an ideal language is in general impossible to achieve, we do not expect extensibility to detract from security provided there exists a secure extension mechanism.



ANTI  \Rightarrow HIGHLY CONFLICTING

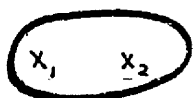
 \Rightarrow X_1 AND X_2 RELATIVELY DEPENDENT

FIGURE 3-1

General Characteristics of Base Languages

Having thus considered the major conflicts and similarities of the design criteria, we consider in terms of these the implications for general characteristics of base languages. We consider 3 characteristics.

- (a) transparency,
- (b) pitch or level,
- (c) size.

We must essentially answer the questions:

- (a) How transparent should the base language be to the real machine?
- (b) Should the base be pitched at a high or a low level?
- (c) Should the base language be large or small or perhaps minimal? (Minimal in the sense that the language is universal and that in addition no construct in the language is an aggregate of primitives i.e. a recursive function subset). Which features should appear in the base language and which as extensions?

We first consider the implications of the various design criteria on a base language composed of a union of a (set of) primitives B1 and special case aggregates B2.

- (1) Modesty \Rightarrow $B1 \cup B2$ is problem oriented.
- (2) Elegance \Rightarrow $B1 \cup B2$ is small.
- (3) Efficiency \Rightarrow $B1 \cup B2$ is small (but not minimal).
- (4) Generality \Rightarrow $B1 \cup B2$ is union of all problem oriented languages; B2 chosen general.
- (5) CSC \Rightarrow B2 emphasised, and chosen for common special cases.
- (6) Orthogonality \Rightarrow B2 small.
- (7) Security \Rightarrow $B1 \cup B2$ oriented to problem area.

- (8) Portability \Rightarrow B1 \cup B2 small.
- (9) Extensibility \Rightarrow B1 \cup B2 transparent to problem primitives.

From this summary, we construct a table showing the relation between the design criteria and general characteristics cf. figure 3-2. We distinguish only three relations \checkmark 0 X.

We construct 3 star diagrams cf. figures 3-3a, b, c to illustrate these conflicts:

\checkmark is represented by a long vector
 0 " " a vector of intermediate length
 X " " a short vector

Since we have already chosen our primary aims as those of security, efficiency and extensibility, we can carry this discussion of general characteristics further:

(a) Transparency cf. Figure 3-3(a)

The principal conflict is between security and generality; the base should not therefore be chosen transparent to the real machine, but transparent to the problem area.

As we observed in proposition 2-1 certain constructs which are not regarded as universally overtransparent in the base language, may be so regarded in extended versions of the language. However, since (in our extension mechanism) extensions are not defined by levels of processor, universal overtransparency cannot be removed by defining extensions (contrast partial overtransparency, case (c)). Thus, low level constructs are not disabled in extended versions of the language. This situation must instead be handled by a new hierarchical level of the

Characteristic Criterion	Transparent to real machine	Low level	Small
CSC	0	X	0
Efficiency	0	X	✓
Security	X	X	0
Extensibility	0	✓	0
Portability	0	0	✓
Modesty	X	X	✓
Elegance	0	0	✓
Orthogonality	0	✓	✓
Generality	✓	✓	X

KEY

- ✓ : criterion well-satisfied by this region of
characteristic space
- 0 : characteristic independent of criterion
- X : criterion badly-satisfied by this region of
characteristic space

FIGURE 3-2

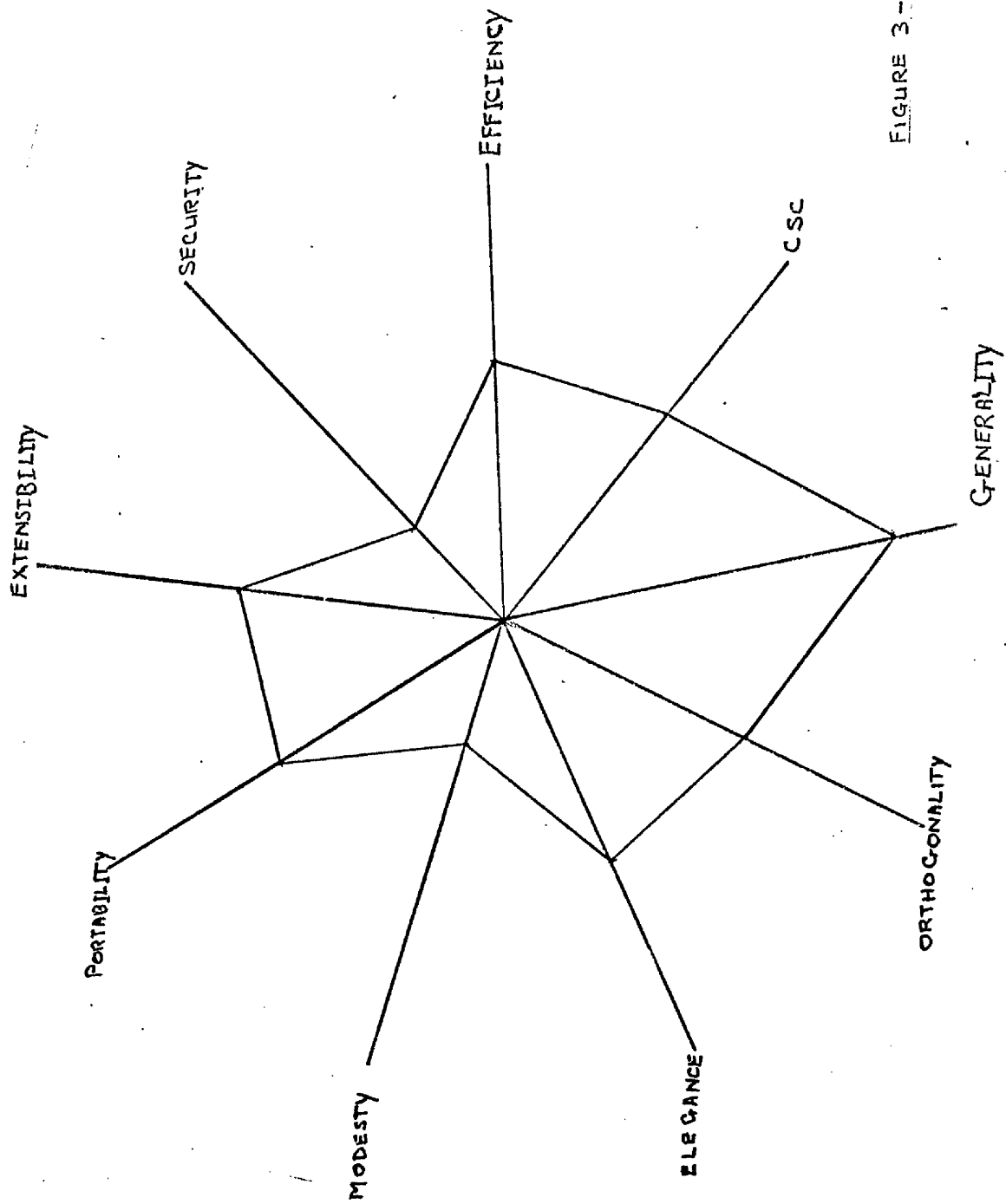


FIGURE 3-3a(1)

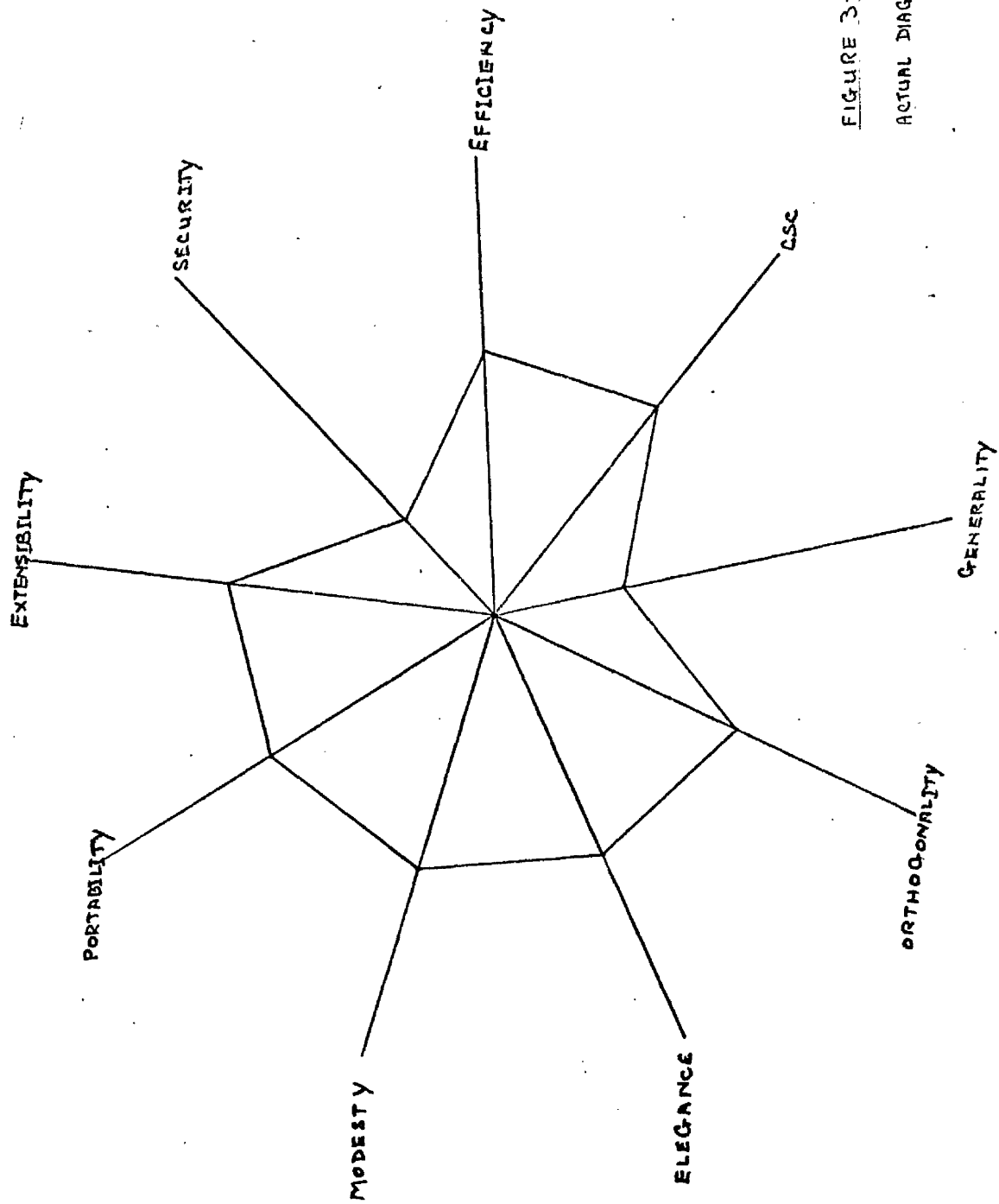


FIGURE 3-3a(2)

ACTUAL DIAGRAM FOR SNIP

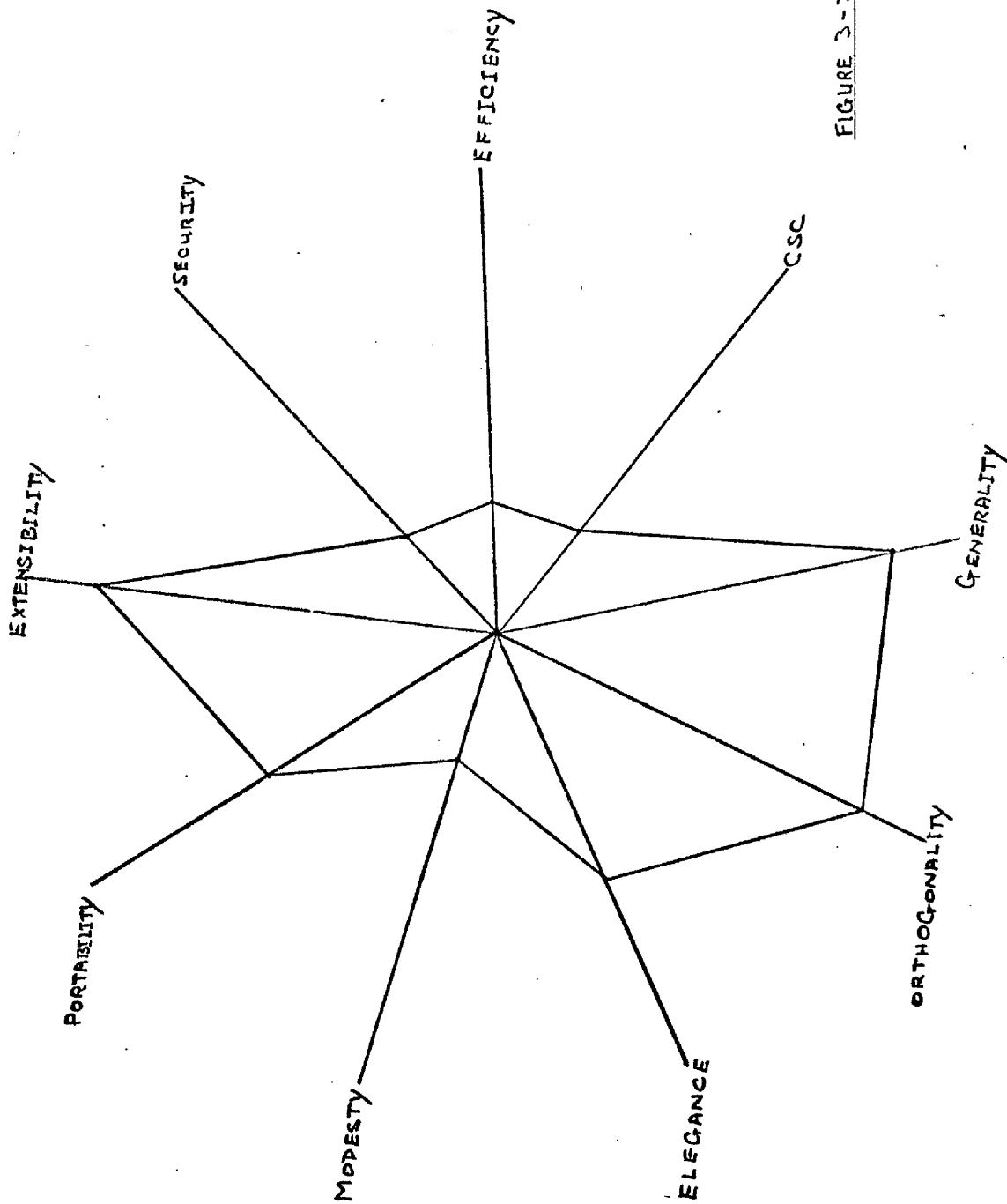


FIGURE 3-3 b00

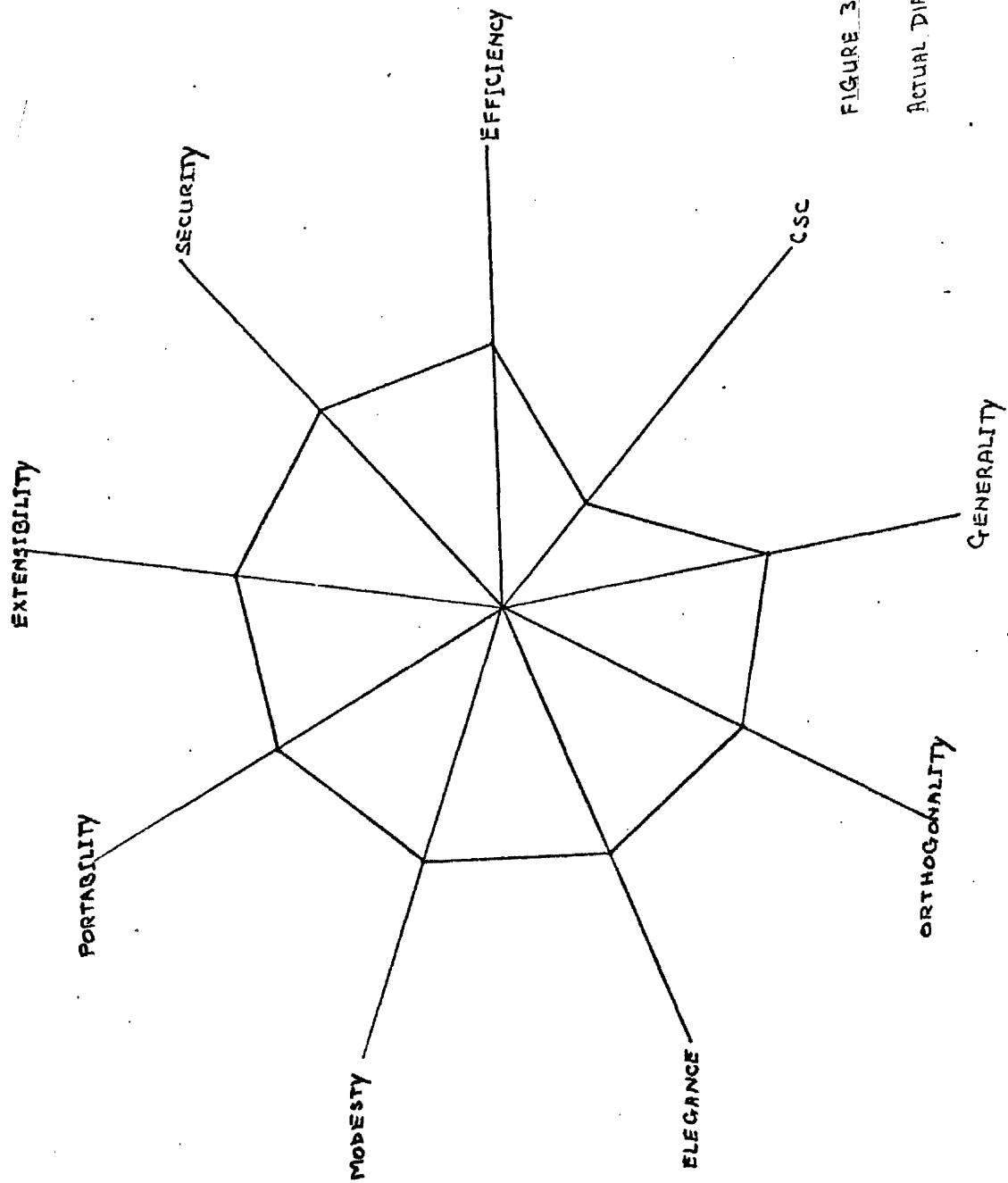


FIGURE 3-3 b (2)

ACTUAL DIAGRAM FOR SNIP

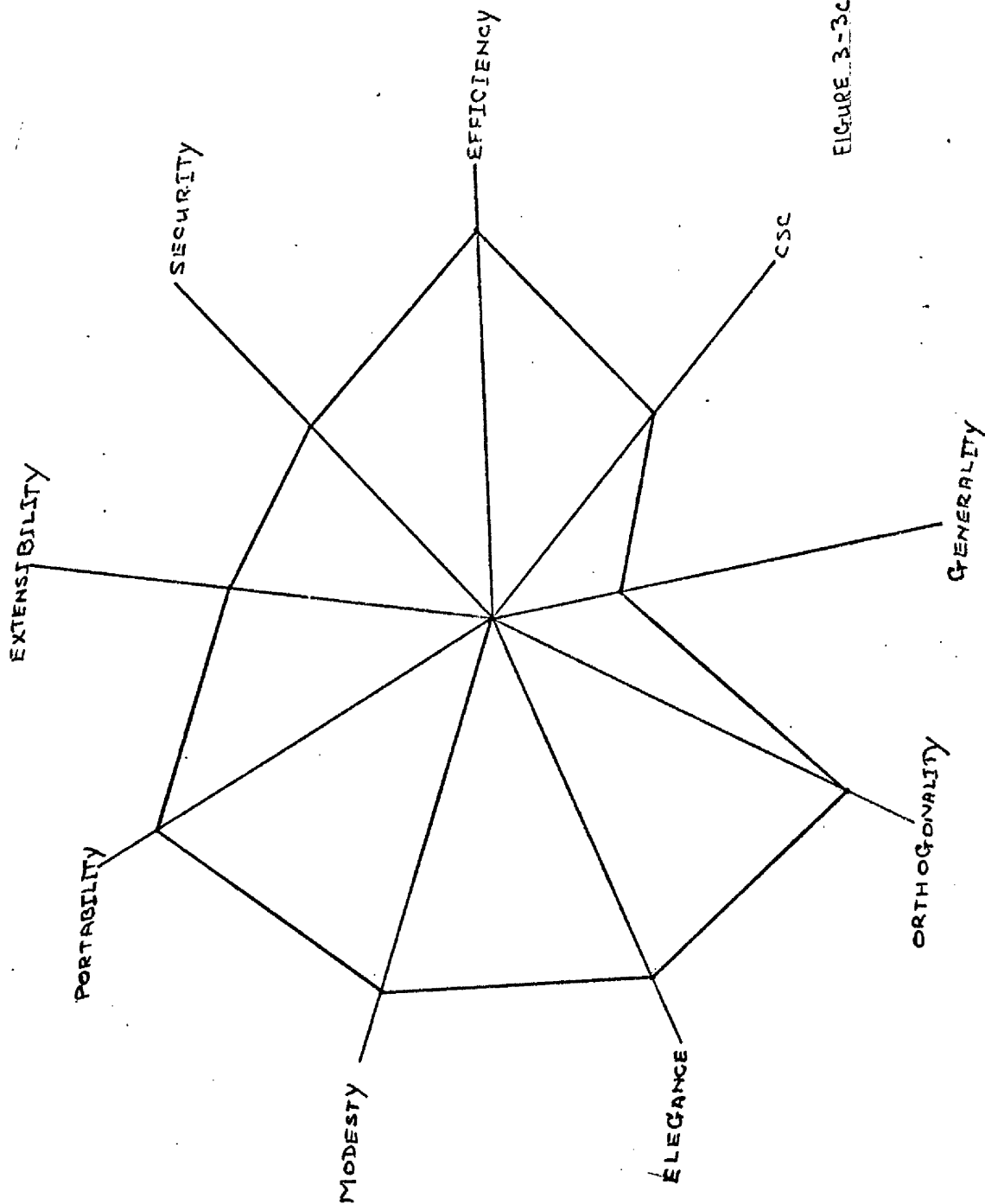


FIGURE 3-3c(i)

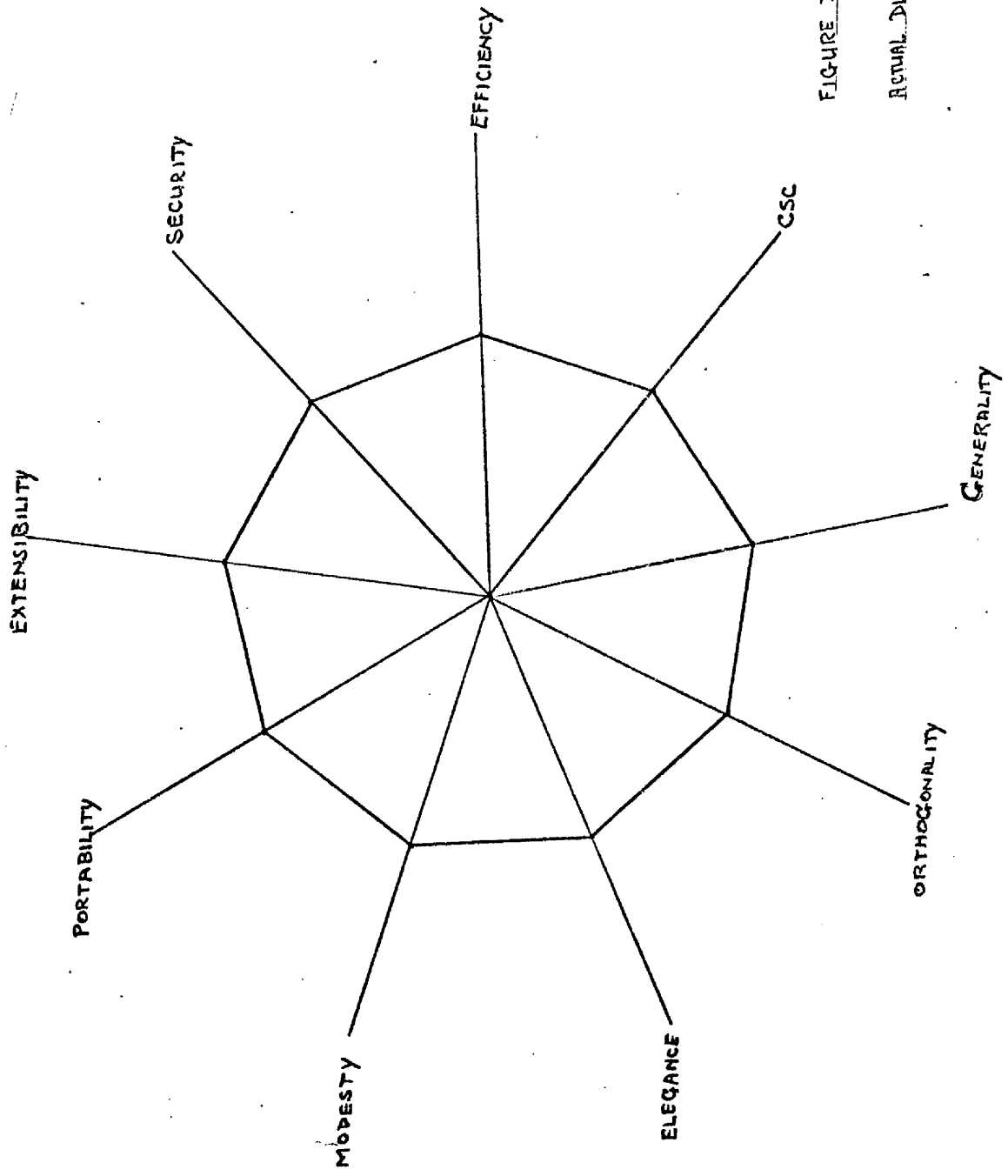


FIGURE 3-3c(2)

ACTUAL DIAGRAM FOR SNIP

language cf. section 2.4. This notion is not unlike Wirth's discouragement of combining different levels of abstraction in the same language (Wir 74).

(b) Level cf. figure 3-3(b)

In order to be extensible, the base language has to be fairly low level, but not necessarily transparent to the real machine.

(c) Size cf. figure 3-3(c)

The base language should be chosen fairly small. Security is not critical here since partial overtransparency caused by minimality can be removed by extension (contrast universal overtransparency). We propose that a small base be chosen to which should be added constructs in the following categories:

(1) For efficiency, the base should include special case constructs which we expect to be used frequently in most extended versions of the language. We would expect efficiency of individual constructs to decrease with increased depth of nesting of definition and therefore feedback from usage statistics may, in the long term, play a large part in the evolution of the base language cf. Woolley (Woo 71), Garwick (Gar 68).

(2) The base should include features which will be required in most versions of the extended language, but which we cannot easily define by extension. In practice, we find it difficult to organise the definition of highly context-sensitive features (e.g. procedure parameters) cf. section 2.4. We propose therefore that such features be included in the base or that an alternative method of

defining these extensions is introduced cf. sections 1.2, 2.4.

3.1.2 Design of Syntax and Pragmatics

In designing the syntax and pragmatics in which to clothe the constructs of the base language, we are concerned with the following design aims:

(a) Translatability

(b) Security

(a) Translatability

We desire a language in which syntax and pragmatics are chosen so that constructs are easily recognised and good code can be produced. Under these conditions, we say that a language is translatable.

The choice of syntax is important, since recognition of a context-free structure together with table processing to handle context-sensitive features represent a considerable proportion of compilation time (Hoa 73). Theoretical consideration of analysers (Hop 69) leads us to expect better results with simpler grammars.

Pragmatics should be chosen to inform the translator of worthwhile optimisations (cf. CSC section 3.1.1.1). For example, the for-statement in Algol W informs the translator that register optimisation is possible.

(b) Security

We must attempt to design secure syntactic and pragmatic forms for the base language features as suggested in section 2.1.4. We reject a strict trial and error method of designing such syntax and pragmatics as useless; and propose the heuristic of building on the

experience of existing languages, conforming to the form of these languages, where suitable; and using studies of characteristic errors to highlight insecure areas.

In the long term, study of characteristic errors found in programs written in the language itself should be of great value.

In contrast to the previous section, we find little conflict between these design criteria. The introduction of secure syntax and pragmatics does introduce some overhead, which, within reasonable bounds, we are readily prepared to accept.

3.2 Design of the Snip Base Language

Section 3.2.1 contains a brief summary of the Snip base language. At this point, the reader unfamiliar with Snip might find it profitable to consider the Snip Report, appendix A.

Subsequently, section 3.2.2 discusses the design of Snip features at length, while section 3.2.3 contains a discussion of the design of the syntax and pragmatics of Snip.

3.2.1 Summary of Snip

The description of Snip, both here and in appendix A, is presented along the lines of the Pascal Report (Wir 70) which we regard as a valuable document from the viewpoint of language presentation. We would expect, however, that reference to some more formal document would be necessary to resolve any remaining doubt over possible interpretations, and to ensure correct implementation.

A Snip program consists of two essential parts: a description of actions which are to be performed, and a description of the data to be manipulated by these actions. Actions are described in Snip by statements, and data by declarations and definitions.

The data are represented by values of variables. Every variable occurring in a statement must be introduced by a variable declaration which associates an identifier and a data type with that variable. A data type essentially defines the set of values which may be assumed by that variable.

The basic data types are the scalar types: integer, boolean. Structured types are defined by describing the

types of their components, and by indicating a structuring method. The structuring methods differ in the selection mechanism serving to select the components of a variable of the structured type. In Snip, the structuring methods are: vector structure, file structure and string structure.

In a vector structure, all the components are of the same type. A component is selected by a vector selector or integer index.

A file structure is a sequence of components of the same type. A natural ordering of the components is defined through the sequence. At any instance, only one component is directly accessible. The other components are made accessible through execution of standard file positioning procedures. A file is at any time in one of the modes: READ, WRITE, READ/WRITE. According to the mode, a file can be read sequentially and/or it can be written by appending components to the existing sequence of components. The file type definition does not determine the number of components, and this number is variable during execution of the program.

A string structure is a sequence of components of the same type. Unlike vector and file structures, the components of a string are implicitly defined as the set of basic symbols (which includes the Snip character set). A component or consecutive sequence of components (called a substring) is selected by a string selector which specifies the initial basic symbol and the length of the substring. Strings are implicitly defined to be of dynamically varying length. It is, however possible to define an upper bound to the string length.

The most fundamental statement is the assignment statement. It specifies that a newly computed value be assigned to a variable (or component of a variable). The value is obtained by evaluating an expression. Snip defines a set of operators, each of which can be regarded as describing a mapping from the operand types into the result type. The set of operators is subdivided into groups as follows:

- (1) The Arithmetic operator group contains addition, subtraction, multiplication and division. The operand and result types are integer.
- (2) The Boolean operator group contains negation, union (or) and conjunction (and). The operand and result types are boolean.
- (3) The String operator group contains a concatenation operator and a match operator. In the former, the operand and result types are string; while in the latter, the operand types are string and pattern (cf. below), and the result type is boolean.
- (4) The Relation operator group contains equality, inequality and ordering. The result of relational operations is of type boolean. Ordering relations apply only to integers. Special forms of assignment statements, the insertion statement and the append statement are provided for string variables.

The assignment statement is a simple statement, since it does not contain any other statement within itself. Another kind of simple statement is the procedure statement which causes the execution of the designated procedure (cf. below). Simple statements are the components or building

blocks of structured statements which specify sequential, selected or repeated execution of their components.

Sequential execution of statements is specified by the compound statement, conditional or selective execution by the if-statement and the case-statement, and repeated execution by the loop-statement. The if-statement serves to make the execution of a statement dependent on the value of a boolean expression, and the case-statement allows for the selection among many statements, according to the value of a selector.

A statement can be given a name (identifier), and be referenced through that identifier. The statement is then called a procedure and its declaration a procedure declaration. Such a declaration may additionally contain a set of variable declarations. These variables can be referenced only within the procedure itself, and are therefore called local to the procedure. Their identifiers have significance only within the program text which constitutes the procedure declaration and which is called the scope of these identifiers.

A procedure may have a fixed number of parameters which are classified into constant- and variable-parameters. If the actual variable-parameter contains a (computable) selector, this selector is evaluated before the procedure is activated, in order to designate the selected component variable.

Functions are declared analogously to procedures. In order to eliminate side-effects, assignments to non-local (i.e. global) variables are not allowed to occur within the function body.

A pattern is a special purpose procedure which, in association with the match operation defines the recognition, within a given subject (sub)string, of a member of a particular set of strings. Statements, called action statements, may be associated with this recognition process. It is, in addition, possible to declare new variables or to define new procedures and functions incrementally, and to modify existing patterns (to the extent of defining new alternates only). These features permit the definition extensions to a language whose compiler is written in terms of Snip.

3.2.2 Snip Features

Within the framework of the characteristics of a base language considered in the previous section, there are a multiplicity of bases, each satisfying the design criteria to differing degrees. We have chosen to place special emphasis on security, but nevertheless, the design of a particular base language must still reflect to some degree the intuition, initial judgements and personal preferences of the designer.

In this section, we consider the design of a base for a string processing language. In view of the UNCOL experience (Stee 61) in which an attempt was made to design an abstract machine capable of representing translations from programs in any high level language (cf. section 0.1), it is not hard to predict that we will not achieve a single efficient and universal base language: and we expect therefore that we must allow the base language to be influenced by existing string processing languages.

In addition, we make use of the following empirical suggestions:

- (1) Simplicity and efficiency are essential for a good base language (Wir 74, Hoa 73).
- (2) In order to keep the size of a language small, it is important to keep the number of basic data types as low as possible (McK 74).

We consider design decisions of particular interest for the features as follows:

Control Structures

Data Structures and operations

Patterns

Procedures and Parameters

Program Structure

Language Extensions

We resolve conflicting design aims using the techniques described in section 3.1.1.2; we recall conclusions of the principal conflicts, namely generality versus efficiency and security cf. figure 3-1.

3.2.2.1 Control Structures

In section 2.1.3 we argued that the goto is, at best, partially overtransparent in a user-oriented language. The principal use of the goto is therefore to synthesise forms of control flow for which no aggregate instruction is provided. The dangers of including the goto are, however twofold:

- (a) the goto may be misused to synthesise aggregate control structures which are provided in the language (cf. programmer discipline), or
- (b) the programmer may synthesise constructs in an obscure fashion.

The arguments for and against the goto have been widely debated (Wul 71; Lea 72; Knut 74).

The programming language, Bliss (Wul 70) has been designed without the goto-statement, while in Pascal, the scope of the goto is restricted and its appearance made less attractive by the use of integer labels, thus discouraging use (Bos 73).

In view of the history of abuse of the goto (Dij 68), we choose not to provide it in Snip.

Since we have shown that security and efficiency are fairly well correlated, cf. figure 3-1, we would not expect the entailed loss of transparency to cause a significant loss of efficiency. This conclusion is substantiated by Petersen (Pet 73) who has proved theoretically that in most cases, programs can without loss of efficiency be written using suitable structured control statements in place of goto statements; and that in those cases where efficiency is lost, there exists an equivalent program which requires only a "little" more space or a "little" more time.

We seek therefore base control structures which are higher level and less transparent than typical real machine structures but sufficiently low level to allow extensibility cf. figure 3-3.

Initially, we considered the low level control structure, Bounce-and-Skip, developed by Nievergelt (Nie 70) as a result of the search for an alternative to the GOTO (Gut 75). However we reject this form of control structure on the following grounds.

- (a) Loss of transparency: Multiple selections (such as case statements) can be handled only by sequential and exhaustive testing. The lack of an "escape-from-

block" construct results in (non-useful) redundant evaluation of block entry and exit conditionals. We consider these particular instances of loss of transparency severe; the first since it appears a useful string processing facility cf. (Wai 73), and the second since it will be used in the definition of most control structure extensions.

- (b) Insecurity: The notation used allows the specification inconsistent, and hence undesirable block entry/exit conditions. Bounce-and-skip essentially allows various forms of primitive conditional and iterative statements. It is thus an aggregate or higher level form of statement. However, since the transparency of bounce-and-skip is no less than that of (combinations of) the primitive statements, the program structure is reflected better by the primitives themselves, particularly when the structure is deeply nested (cf. appropriate structuring, sections 2.1.3, 2.1.4).

As an alternative to bounce-and-skip, we consider the structures:

- (1) a simple loop-statement, in which exit is by way of an escape statement (cf. below)
- (2) a conditional statement,
- (3) a simple block structure,
- (4) an escape-from-structured statement.

These structures are essentially the primitives from which bounce-and-skip is synthesised. Individually, each of the structures (1)-(3) is less transparent (to the base machine) than bounce-and-skip. In combination, however, the 4 structures are more transparent since bounce-and-skip

restricts the escape-statement to occur at block end only.

These alternative control structures provide a simpler and more suitable abstraction which avoids one area of loss of transparency considered above. The introduction of a case-statement removes the remaining loss of transparency. A case-statement is introduced in preference to the poorly structured switch statement of Algol 60 (cf. section 2.1.4).

These control primitives are non-minimal. For example, the conditional-statement can be implemented in terms of the case-statement; in doing so, however, we lose the capability for optimising the conditional-statement, which is likely to be heavily used cf. CSC section 3.1.1.1.

Petersen (Pet 73) has proved theoretically that all "well-formed" programs (in which loops and if-statements are properly nested and can be entered only at their beginning) can without significant loss of efficiency be written in terms of:

- (1) if-statements
- (2) repeat-statements
- (3) multi-level exits

Petersen's structures are less transparent to the real machine than our own: it is not possible to handle multiple selections (e.g. case statements) efficiently. His structures are also higher level: the repeat statement is an aggregate form of the loop, conditional and escape statement; multiple exits are an aggregate of case and escape statements. Petersen's structures can therefore be defined in terms of the Snip control primitives described above and we are thus encouraged in the belief that these primitives are capable of representing a large

proportion of high level control structures. Furthermore, Snip control structures are primitive to Petersen's structures while allowing (with the exception of the case statement) no more transparency, thus improving extensibility cf. figure 3-3.

3.2.2.2 Data Structures and Operations

We consider the design of a set of data structures and operations capable of conveniently and securely reflecting the abstractions and abstraction-manipulations devised by the programmer.

Wirth (Wir 74) and Hoare (Hoa 73b; Hoa 75) propose that the following should be primitive data and constructors for a general core language:

- (a) Cartesian Product - variables X_1 to X_n of type t_1 to t_n .
- (b) Type Union - e.g. records whose components are variables of different types.
- (c) Type Replication - e.g. arrays whose components are variables of identical type.
- (d) Type Recursion - a new type may be defined in terms of itself and/or other types.

The basic data types and operations will vary for different applications.

We recall (cf. section 2.4) that we will not at present consider the method by which we propose to extend data structures, and hence we do not consider Type Recursion.

We gain some insight into the kinds of operations and the basic data types which we expect to be useful by considering existing string processing languages (Car 66,

Chr 65, Coh 65, Gris 71, Yng 63) and the discussion by Katzan (Kat 70).

It seems reasonable to expect that the basic data types integer and boolean will be required. To handle replication, we choose the simplest array structure (the vector), Pascal-type files for input and output, and strings of characters.

On the basis of involution, it might seem reasonable to introduce strings (as Standish proposes (Sch 71a)) as an array containing an indefinite (or dynamic) number of characters or as a more general replication type. However, in the first case it is hard to introduce appropriate and efficient substring accesses and operations; and in the second case strings of integers and boolean components would not appear to allow useful data structures or operations (cf. arbitrary features, section 3.1.1.1).

We choose therefore to define the type "character" as a string of length 1, rather than define a string in terms of characters.

Data Structures and Invariants

The data structures so far proposed are considerably abstracted from, and hence much more secure than, the corresponding structures at real machine level. However, they still have fairly transparent properties, and there are many ways in which we regard them as universally over-transparent. We therefore consider use of invariants to improve security at the expense of loss of generality (cf. figures 3-1, 3-3). We consider type checking, subranges and mode of access.

(a) Type Checking

We have already argued (cf. section 2.1.3) that weak typing (as found in the so-called typeless languages) is universally overtransparent for a large number of applications (cf. Wortman (Wor 74) and Stewart (Ste 74)). Wirth (Wir 74) makes the hypothesis that the most common reason for programmers desiring typless languages is to allow the packing of different kinds of data into a single word, which the available language regards as an indivisible entity. This ability to pack data can be provided without preventing valuable type checking (e.g. Pascal). We consider therefore, that a "strongly typed" language (Wor 74) with suitable data structures and type-transfer functions will allow sufficient transparency while maintaining maximum capability for detection of error. The variation of strong typing offered in Snobol 4 (Gris 71) in which variables are typed but whose type may change dynamically during program execution is less secure, since the type rule invariant degenerates to a series of assertions at points in the program (cf. section 2.1.3).

Since automatically invoked type-transfers have the effect of overriding type rules, we provide no such facility in Snip, and insist that all calls on type transfer functions be explicitly programmed. Hoare (Hoa 73) points out, in any case, that automatically invoked coercion in the base language has the effect of prejudicing extensions since base language coercions will apply also to the extended language.

(b) Subranges

The idea of subranges introduced in Pascal, is really

a strengthening of the notion of strong typing. Thus, for example, if it is possible to specify that a particular integer variable may assume only those values which lie within a particular subrange of the set of integers, the type rule invariant is strengthened.

Since variables which may assume any integer value will undoubtedly still be necessary, to omit this feature from the base language causes only partial overtransparency. However, because of the highly context-sensitive nature of this construct, we find it exceedingly difficult to handle implementation by extension. Since we do not expect integers to be heavily used in Snip, we propose to defer judgement on implementation of integer subranges until such time as we can consider the characteristic errors of Snip.

(c) Mode of Access

We consider protection of variables by defining invariants which restrict the mode of access. By comparison, access to procedures and functions is restricted to "execution only" in most high level programming languages. We consider two forms of access for variables: "read and write", "read only". We adopt the Pascal convention of defining such variables as CONSTANT or VARIABLE. In Pascal, this form of access can be defined for local variables and for procedure or function parameters. In Snip, we extend the notion, allowing the user to define similar invariants for non-local variables. As in the case of subranges, omission of this feature causes only partial overtransparency, but its context-sensitive nature prevents convenient implementation by extension.

We consider now the design of individual data

structures and operations. We expect that Snip may prove useful in other application areas such as the editing and manipulation of text tiles on backing store or the processing of job control language programs (cf. section 5.2), and temper our design decisions accordingly.

We consider (a) files and (b) strings.

(a) Files

In the spirit of Pascal, we propose that input and output be by way of files only. This abstraction improves security by shielding the programmer from hardware details concerning the handling of backing store and input/output devices. Since we expect text files to contain perhaps Snip or job control language programs, it is appropriate that such files consist of short strings (of the order of 1-2 hundred characters) of a fixed maximum length.

Operations on Files

We considered designing file operations transparent enough to allow flexible manipulation of file structures (for example, insert or delete a file component). However, since, to ensure recoverability, it is unusual to manipulate the master copy of a file, these operations might be better achieved while copying the master.

We therefore retain the set of standard procedures used in Pascal for file manipulation, introducing one further procedure to allow append operations. We do not allow sequencing backwards through a file as we consider this introduces confusion unnecessarily.

(b) Strings

We expect strings to grow frequently by appending

or concatenation of string expressions and to contract frequently by substring deletion; an upper bound for the length of the string will not, in general, be known.

String Operations

Many string operations in existing languages are too high level and restrictive to be considered for a base language. We feel that the transparency of these operations can be increased without reducing them to unmanageable and insecure levels:

The match operation in Snobol 4 is non-primitive in the sense that it combines primitive operations of matching, and substring-replacement: it is difficult therefore to handle substring-replacement on its own. A substring-fetch is equally difficult.

Algol W (Hoa 66) allows a simple means of access to substrings, but insists that the length of the substring be known at compile-time: while this allows a more efficient implementation of simple string handling, it severely prejudices the efficiency of more general substring accesses of the kind we expect to be useful in Snip.

In Snobol 4, it is not possible to handle substring insertion or append operations efficiently.

We discuss the more interesting operations:

Substrings

We encounter difficulties in attempting to develop a natural and consistent notation for specifying substrings. On the basis on involution, it seems reasonable to allow assignment to substrings in the same way as assignments to string variables are handled. This appears also to provide

a useful means of handling substring-replacement, at the same time.

In assignment, the length of the string (to which the assignment is made) is determined by the length of the assigned string expression. However, to allow a substring (whose length is specified in its denotation) to be replaced by a longer string expression, appears a contradiction of terms. Further, this prejudices the base against optimisation of the special case in which the length of the string expression is less than or equal to the length of the substring. It also disables the assertion concerning string lengths for this special case. From personal experience, we expect this special case to be fairly common. In contrast, the non-optimisable case can without prejudice to efficiency or security, be defined by extension, if it proves to be heavily used. We therefore demand that a substring be replaced only by an expression of equal or lesser length (cf. CSC, section 3.1.1.1).

We allow the deletion of a string by assigning the null constant to it. Since the null constant has zero length, we may similarly delete the substring without violating the above restriction.

Append and Insert Operations

We introduce an append operation to allow the appending of string expressions. This operation (depending on the implementation cf. section 4.4) allows updating of an existing string value (cf. CSC, section 3.1.1.1). We introduce an insert operation for similar reasons. It would be more elegant to include this in the substring

notation (cf. replacement and deletion) but we find it difficult to adapt the notation in any reasonable manner.

3.2.2.3 Patterns

Security

By the arguments of section 2.4, we have already determined that Snip patterns should have a structure isomorphic to the class of grammars known as LL(1). We consider that patterns presented on the level of Markov Algorithms (Weg 68) or Floyd's Production Language (Fel 64), for example, are universally overtransparent for our purposes. The control over the recognition process provided in Floyd's Production Language (PL) is analogous to the goto-construct of more general programming languages. The Markov Algorithm is a theoretical model of computable functions and was never intended as a practical programming language (indeed in practical terms, it specifies exceedingly inefficient algorithms). Carraciolo (Car 66) has however, designed a more practical "Generalised Markov Algorithm" which is used in the programming language Panon 1B. This turns out to be very similar to Floyd's PL.

We therefore present Snip patterns on a higher level: we find that control of recognition can be more securely provided by constructs very similar in purpose to those designed for more general control in section 3.2.2.1. Thus, the forms of control in Snip patterns are as follows:

- (1) Repetition, indicated by the pair brackets "{ " and " } ",
- (2) Conditionals or alternatives, indicated by " | ",

- (3) Compound structure, indicated by the pair brackets
"(" and ")",
- (4) Recursion, indicated by the pattern identifier itself
(cf. procedure or function call),
- (5) Escape-from-pattern, implicitly defined as a "match
fail".

We do not define a control structure analogous to the case statement since, in general, there is no implicit (or compile-time defined cf. Pascal) ordering of the labelling string literals and since pattern variables may be intermingled with these string literals. The possibility of simple optimisation of this structure is therefore lost.

The resulting structure is isomorphic to generalised BNF (Grif 74) or, equivalently to Chomsky Context Free Grammars (Hop 69). It is therefore necessary to check explicitly that pattern structures are in fact further restricted to LL(1). (cf. appendix C).

For the sake of involution it would seem reasonable to allow string variables as well as string literals to form components of pattern templates. To allow this, however, would be to allow the possibility of far reaching side-effects; the structure of several patterns could be altered by the assignment of a new value to a single string variable. Experience of the author, and others (Dah 72; Bec 75) with the debugging of programs suggests that programmers are liable to over-look similar side-effects while making modifications cf. instability. We therefore feel justified in explicitly disallowing string variables as components of pattern templates.

In most string processing languages, the order in which

alternation string sequences of a pattern are compared to the subject string is of considerable importance, since this order may affect the result. However, since the Snip pattern structure is deterministic (cf. section 2.4), this order has no effect on the result of the match.

Since a pattern defines a subset of the set of possible string values, it would seem reasonable to consider patterns as the definition of a variable type, rather than a variable itself (e.g. Snobol 4). In effect, this is a generalisation of the Axle notion (Coh 65) of defining patterns as tables of "assertions".

It would then be possible to treat simple strings as degenerate pattern types. This has the advantages of providing a more involuted structure, reducing the number of basic types (cf. McKeeman above) and providing a convenient means of referring to matched substrings.

However, this structure is in some ways unsatisfactory. In general, the manner in which we employ pattern types and simple string types is not at all similar. During pattern matching, we will frequently wish to fetch and manipulate matched substrings. These operations may be relatively securely defined if we introduce an aggregate form of pattern which allows the inclusion of suitable "action" statements (in the pattern template) to be executed at the appropriate point during the recognition process. Snip patterns have thus evolved in such a way that they correspond more closely to a special purpose boolean function.

We propose therefore that pattern definition be restricted to the declaration area at the head of a block, as is the case for function declarations.

We must allow some means of communication between the groups of statements (or action primitives) defined in pattern declarations. Communication through global variables alone is unsatisfactory as a stack mechanism for storage of current variable values must be explicitly programmed for recursive pattern call. Since we expect this facility to be of great value in Snip, security is improved if we allow communication through variables which are local to all the action primitives, or, equivalently, local to the pattern containing the action primitives.

We further improve the design of patterns (for our purposes) by associating a cursor with each string variable. Matching on the subject string proceeds from the current cursor position. The cursor position is updated on a successful match (e.g. Axle). Without the cursor, this form of matching can be effected only by deleting the string head (e.g. Snobol 4).

If no match is obtained, starting from the current cursor position, the match fails. This operation is primitive to the corresponding Snobol 4 operation in which the match continues, restarting from the next string position.

Snip patterns are thus presented at a very high level. We might expect, for example, that match-string assignments, recursion and action primitives could be implemented by extension, since the existing base is non-minimal. However, we justify inclusion of these features in the base language as follows:

- (1) We expect patterns in Snip to be heavily used cf.

section 4.3 in both the base and extended versions

of Snip. Hence, efficiency is of considerable importance.

- (2) In order to handle the implementation of patterns or to extend simple forms of patterns, very transparent base language features (for example code pointers and operations to allow manipulation of the local variable stack) are required.

Efficiency

We now consider the trade-off between generality of pattern constructs and the optimisation of frequently occurring special cases. As we have noted, we expect heavy use to be made of patterns, and hence there is opportunity for considerable gain in efficiency. In this respect, the avoidance of backup (because of the LL(1) structure of patterns) is already an important step.

Snobol 4 allows the construction of patterns at run-time (latent patterns (Wai 73)). This causes a considerable amount of effort, since it involves the copying of the components from which the new pattern is constructed and the adjustment of successor and alternate pointers within the thus-constructed pattern, by traversing the whole(run-time) representation of the pattern.

We consider that patterns used in Snip will, in general, be known at compile-time (manifest patterns). This is in any case, more in line with our view of patterns as a special kind of function. We expect to achieve a further considerable saving by holding a single copy of each pattern defined (cf. procedures and functions) along with a return address, rather than explicitly copying subpatterns. This is not advisable with latent patterns,

3-30

because of possible side-effects (cf. above). This implementation also circumvents the need to handle pattern recursion by dynamic reconstruction (cf. Snobol 4).

Snobol 4 is able in certain cases to speed up the matching process by using a length check. If the subject string is insufficiently long, failure can be signalled immediately. We do not consider this feature useful in Snip, because, in general we will require to scan the whole of the subject string in order to produce diagnostic messages.

3.2.2.4 Procedures, Functions and Parameters

We include functions and procedures in the base language because we expect these to be required in all versions of Snip. As we argued in section 2.1.3, we consider that the use of side-effects in functions is unstable and we therefore explicitly disallow assignments (in function bodies) to non-local variables or parameters cf. instability.

Parameters

It is theoretically a simple matter to define the meaning of procedure or function parameters by explicit copying (cf. Algol 60 copy rule (Nau 62)) and by declaration of local variables. This means of definition is, however, too inefficient to be used in practice.

Thus, in order to handle the definition of parameters by extension, we would have to include in the Snip base, a means of explicitly manipulating the stack used to implement local variables (cf. section 3.2.2.5), and a means of referring to variable addresses. Such constructs are very

low level and transparent for many Snip applications; we regard them as universally overtransparent.

In addition, the handling of parameters involves considerable context-sensitive action for checking and addressing purposes. Since Snip patterns are context-free, this context-sensitive action must take the form of table processing. If parameters are defined by extension, it is exceedingly difficult to organise the building and processing of such tables and to relate these to existing tables (for procedure identifiers and variables used as actual parameters).

Hoare (Hoa 73) points out that procedure interfaces are particularly susceptible to error. It is perhaps, therefore advisable that the base language itself should impose secure forms of parameter passing, flexibility being, at this point, undesirable.

We propose, therefore to include in the base language, a uniform and simple parameter passing mechanism. We consider that call-by-name for an expression, or use of Jensen's device (Nau 62) is universally overtransparent for most applications and propose to disallow this. The CONST-parameter facility of Pascal seems preferable to the Algol W VALUE-RESULT mechanism for its relative simplicity, and hence security cf. section 2.1.4. Characteristic errors of Algol W (cf. appendix G) suggest some difficulty with VALUE-RESULT. We adopt, therefore, the simple form of parameter passing found in Pascal: a variable as parameter may be called by name or value; an expression as parameter may be called by value only (cf. indirect addressing and value call, Hoare (Eng 71)).

Hoare (Hoa 73) has suggested that perhaps the most subtle defect of the Algol 60 parameter mechanism is that the user is permitted to pass the same variable twice as an actual parameter corresponding to two different formal parameters (cf. pointers, section 2.1.3). For example, if a procedure

```
matrix_multiply (A, B, C)
```

is intended to have the effect

```
A := B X C,
```

it would seem reasonable to square A by

```
matrix_multiply (A, A, A).
```

We consider therefore two further restrictions on parameters to improve security cf. Hoare (Eng 71). We cannot hope to implement these restrictions, however, and must rely on programmer cooperation (cf. section 2.1.3).

- (1) All actual parameters whose values may be changed by a procedure must be distinct from each other and from non-local variables which are referenced in the procedure.
- (2) None of the actual parameters described in (1) may be contained in any of the other parameters.

3.2.2.5 Program Structuring

Block Structure and Scopes

As we observed in section 2.1.3, block structuring and local variables increase security by increasing checkable redundancy. Neither can be simply or efficiently implemented in Snip by extension, and indeed, we regard a base language without some such properties as universally overtransparent. As Dijkstra (Dah 72) observes, while we

may have some misgivings about the specific scope rules as embodied in Algol 60 (cf. below), we should appreciate them as a very significant step in the right direction. We feel that this structure, as realised in Pascal is likely to provide a valuable simplification without compromising the applicability of Snip to string processing problems.

Scope Rules

While the number of undetected errors caused by confusion of scopes was relatively small in the sample programs studied in appendix G (cf. error type 3.4.2, figure G-1), we feel that this kind of error is liable to occur much more frequently in larger and more complex programs. This kind of error arises principally from the combination of two features in an Algol-like block structure:

- (a) The scope of a name is by default extended to inner blocks; and
- (b) Identical identifiers may be used in different scopes to denote different variables cf. (Wul 73) and section 2.1.3.

This results in three types of error: inadvertently interposing a redeclaration of a global name in an inner scope causes reference to the name in the inner scope to be bound to a new local variable, rather than to the global variable. Failing to redeclare a global name that was to be re-used to identify a local variable (either through omitting an entire declaration or through mis-spelling the name of the variable in the new declaration) causes names in the inner scope to be bound to the global variable, rather than to the new local variable. Confusion of the meanings associated with identical identifiers which denote

different quantities causes similar errors cf. (Gan 75).

Many errors of this type could be avoided by insisting on use of unique identifiers. We reject this solution, however as use of non-unique identifiers is both convenient, and, if procedures are to be independently coded, necessary.

We propose therefore a partial solution to the problem. We insist that reference to non-local, variables be explicitly defined, together with the type of access (read or read-write) permitted. We do not expect this mandatory declaration to be a burden on the programmer, but rather, an aid to clarifying his thought (cf. declaration of local variables). Gannon (Gan 75) imposes similar restrictions on the programming language TOPPS II.

3.2.2.6 Language Extensions

To a large extent, we considered the secure definition of extensions in section 2.4 (i.e. the capability for defining the syntax of extensions and for defining the semantics of extensions. In this section we consider suitable Snip constructs to handle these definitions. We recall from section 2.4 that we do not allow the syntax or semantics of existing extensions to be altered; we allow merely the addition of alternative constructs.

We introduce the statement:

`<statement> ::= DEFINE <pattern body> AS <context specifier>`

to define an alternate for the pattern specified by the <context specifier>. The <pattern body> defines the syntax of the extension, and also the action required to generate the appropriate substitution string. The context specifier defines the context in which the defined extension

5-43
may be used. We must ensure that the extended syntax remains LL(1) (cf. appendix C).

Context Specifier

The most obvious means of specifying context is merely to specify the syntactic class to which the extension belongs (as in most existing systems). There are many alternate methods of specifying context (e.g. use of predicates (Milg 71)), but none appears simpler.

Substitution String

In order to handle the generation of appropriate substitution strings (in expanding extension constructs appearing in a program), it is sometimes useful to have extension-time statements (for example, in handling context-sensitive semantics). For this reason, we allow the incremental declaration of new global variables and the definition of new procedures at extension-definition time. In certain circumstances, it may be useful to allow read access to global variables of the compiler, although we would expect use of this facility by sophisticated users only. To protect the base compiler and existing extensions from subversion cf. section 2.2, no assignments may be made to global variables. Similarly, we may allow compiler procedures to be executed, but not altered.

Extensions need not therefore be defined by pure substitution strings. We say therefore that the definition of extensions is procedural rather than declarative (Sch 71a).

Pattern Subdivision

We expect that by far the majority of extensions

defined will be relatively simple, and that the system so far defined will be well able to handle such extensions. However, for a smaller number of extensions, the system is too inflexible: this is because we have as yet no means of defining an alternate (extension) to a part of a pattern. We illustrate this problem by example:

Example: We assume the existence in the base language of a pattern defining a for-statement:

PATTERN FOR_ST:

```

BEGIN
    "FOR" . <VAR> . ":" = " . <EXPR> . "STEP" . <EXPR> .
                                     "UNTIL" .
    <EXPR> . "DO" . <ST>
END

```

where <VAR> and <EXPR> denote patterns defining variable and expression, respectively. We have no means of defining "TO <EXPR> " as shorthand notation for "STEP 1 UNTIL <EXPR>", as we are able to define alternatives only to FOR_ST itself. We propose to resolve this problem by allowing the pattern FOR_ST to be redefined in a restricted manner:

PATTERN FOR_ST:

```

BEGIN
    "FOR" . <VAR> . ":" = " . <EXPR> .
    ( "STEP" . <EXPR> . "UNTIL" . <EXPR>
      | <TO_EXPR> ) . "DO" . <ST>
END

```

where <TO_EXPR> denotes the pattern defined as above.

We must ensure, however, that the original pattern remains unchanged after this kind of modification. We note, in passing that a non-deterministic system can side-

step this difficulty, but would, in doing so lead to a less efficient and less compact definition. We proceed to consider this modification of existing patterns more carefully:

We propose the following construct to allow this kind of extension definition:

TAKE <modified pattern template> WHERE <extension declaration> AS <context specifier>

The <extension declaration> is the declaration of a pattern defining the extension. The <modified pattern template> is the original pattern template modified to include the newly defined extension as the alternate of some <simple pattern string> of the original pattern.

As with the define-statement, we must ensure that the modified pattern is still LL(1). In addition, we must ensure that the syntactic form of the original pattern remains unchanged. Thus, if additional bracketing is introduced in defining the new alternate, we must ensure that it does not override the original syntactic or precedence structure.

There is no need to specify action primitives of the original template, as these must remain unchanged.

In defining optional extensions, we find it convenient to assume the existence of redundant null successors in the original pattern. Since null successors have no effect on the structure recognised by a pattern, this causes no problems. (cf. appendix C).

The take-statement is rather clumsy but is brought about by the decision that the base should be protected from subversion. We expect, however, that we can organise

the base language so that the define-statement can be used for constructs which we expect will be frequently extended cf. CSC, section 3.1.1.1.

3.2.3 Syntax and Pragmatics of Snip

Having considered the design of the features of Snip, we proceed now to consider the design of syntax and pragmatics.

We consider the effect of the aims of translatability and security individually.

3.2.3.1 Translatability

Since we wish to be able to describe a Snip compiler in terms of Snip itself, and since we have chosen an LL(1) structure for Snip patterns, we restrict the syntax of Snip to LL(1) also. Anderson (And 71) and Griffiths (Grif 74) observe that reasonably efficient recognition is possible.

In order to allow the translator to take advantage of possible optimisation in an append string or insert string operation, we introduce a unique special symbol to denote these operations.

The labelling of the beginning of structured statements rather than the end permits simple compilation (at the expense of object code) in a single pass. While this practice might appear unnatural and obtuse, we consider that it may benefit security by encouraging the user to define exits at the most appropriate point during structured programming development.

Wirth (Wir 75) observes that for simplicity and translatability, Pascal design aimed at a reasonably small

number of operator precedence levels in contrast to Algol 60. Precedence is similarly chosen in Snip.

3.2.3.2 Security

Research into the study of characteristic errors of programming languages has, as yet received little attention and our conclusions in this area are necessarily tentative. By examining the evolution of programming languages, it is, however, possible to discern certain forms of notation which have been accepted as insecure and the measures taken to combat this insecurity.

In addition, we make use of error statistics for Algol 60 and Algol W programs collected by Pirie (Pir 75) to obtain a quantitative analysis of characteristic errors for Algol 60 and Algol W. This analysis is useful because of the similarities of many features of Snip (for example block and control structure) to Algol-like languages. From these figures also, we are able to predict errors which are liable to occur in certain other Snip features; thus suggesting ways in which the syntax and pragmatics should be re-designed. The studies of characteristic errors by Gannon (Gan 75), Ichbiah and Rissen (Ich 74) are also of value.

It would have been useful to consider also, characteristic errors for a string processing language such as Snobol 4, but neither programs nor statistics were available. We have, in any case, indicated that Snobol 4 has many insecure features (cf. section 2.5).

In appendix G, we consider the material available. Figure G-1 summarises this material and is used in considering the design of suitable syntax and pragmatics

for

(1) stability, and

(2) notation.

(1) Stability

Structured Statements and Bracketing

The characteristic errors of Algol W (error type 1.4.1, figure G-1) suggest that mismatch of bracketing is a fairly common error. While most of these errors are detected at compile-time, it is conceivable that some logic errors are caused by confusion of highly nested structured statements. We consider that error is both less likely to occur, and more likely to be detected when it does occur, if a unique non-terminal is used to terminate this kind of structured statement, (cf. Algol 68 (Wij 69)), rather than a semi-colon or end symbol. For example, we use the following forms of if-statement:

```
IF <boolean expression> THEN <statement> ELSE  
                                <statement> FI  
IF <boolean expression> THEN <statement> FI
```

This form of structured statement improves readability by providing a more readily recognised bracketing structure (cf. section 2.1.4).

Declaration of Variables

A significant number of identifiers were misspelled (error type 1.1.2, figure G-1) in the sample Algol W and Algol 60 programs examined. This shows the value of declaration of variables in assisting error detection and in the avoidance of potentially severe errors. All Snip

variables must be explicitly declared.

Commentary

Commentary is a valuable aid to secure programming. The tables of characteristic errors (error types 3.2.3, 3.2.4, figure G-1), show that the form of commentary provided in Algol 60 and Algol W causes a significant number of serious errors by preventing detection of a missing semi-colon (Hoa 73).

Example (1) COMMENT COMMENT WITH SEMI-COLON MISSING

X := 3;

(2) END X := 3

Algol W prevents the second, but not the first kind of error: an end-comment consists of a single identifier only.

The form of comment introduced in Pascal e.g. {This is a comment} may cause undetected error, since the effect of omitting a closing bracket may be cancelled by a subsequent comment. Similarly, in Algol 68 where opening- and closing-comment symbols are identical, two similar errors may have a cancelling effect (Sco 73). In the less serious situation, error is detected, but often results in the annoying effect of treating the rest of program text as comment.

We therefore adopt the proposals of Hoare (Hoa 73) and Scowen (Sco 73) that comments be terminated by the end-of-line symbol.

String Literals

The characteristic errors found for the sample Algol W programs lead us to expect that quote-marks will

frequently be omitted from string literals (error types 1.3.4, 3.2.5 figure G-1). This kind of error is likely to occur much more frequently in a string processing language. There is therefore some risk of a string literal which is inadvertently left unquoted, being confused with a variable identifier consisting of the same sequence of letters.

We consider introducing the following restrictions:

- (a) all string literals must be declared, and
- (b) string literals and identifiers declared in the same scope must be unique.

Thus, the literal "X1" and the variable identifier X1 would not both be permitted in the same scope. A similar restriction is placed on defined scalars and variable identifiers in Pascal, although for different reasons.

It is not clear whether the above solution will be an excessive burden on the user; nor is it clear how serious the problem with omission of string literal quotes will be. We choose therefore not to implement this restriction until we have some feed-back on the characteristic errors of Snip.

(2) Notation

Language Structure

We have already chosen an LL(1) structure for the language because of its simplicity and because of ease of extension. It is however, also important that the syntactic structure of the language be sufficiently flexible to allow choice of a natural notation. For example, the notation used for simple macro calls is too rigid and inflexible. However, the description of the (syntax of

5-51

the) Pascal language in terms of an LL(1) grammar (Wir 71) would seem to allay fears that an LL(1) syntax is too restricted.

Choice of Identifiers

Confusion of identifiers (cf. figure G-1) seems to be a common problem, (error type 3.3.3) at least with student programs. We expect, that in some cases, this is due to logic errors (for example, confusion of control variables in nested for-statements), but that in many cases, it is due to the use of poor (1-letter) mnemonics for identifiers. We would expect that the frequency of occurrence of this error would be reduced by encouraging the use of longer and better chosen identifiers cf. (Weis 74). This would also allow detection of misspelled 1-letter identifiers (error type 3.6.1).

Ordering

A significant number of undetected errors in the sample programs were caused by incorrect sequencing of ordered lists (cf. Ichbiah (Ich 74)) of parameters (in procedure calls), subscripts (in array designators) or case statement components (error type 3.1, figure G-1). We hope to reduce this problem by introducing a better mnemonic notation.

We propose to label case statements, as in Pascal. This also allows detection of omitted case components (error type 3.2.1, figure G-1).

We might label parameters in procedure and function calls (cf. Algol 60), but this form of mnemonic seems excessively clumsy for use with array subscripts: we have

been unable to devise any suitable alternative.

Procedures and Parameters

Hoare (Hoa 73) observes that a high proportion of program errors occur at procedure interfaces. By contrast, we observe comparatively few such errors in our sample programs (error type 3.5, figure G-1). We make the hypothesis that this is due to the relatively simple nature of the sample programs. Hoare expects that the rate of error can be reduced by choosing a notation so that the effect of a procedure on its parameters is obvious from its syntactic form. Most existing languages adopt this rule to some extent for procedure declarations, but not for procedure calls. Since adoption of a similar rule for procedure calls adds to program verbosity, we propose to defer decision to a Gannon-type experiment on Snip in some future research cf. section 2.1.2.

Control Structures

(a) The control structures proposed to reduce over-transparency (cf. section 3.2.2.1), also have the effect of improving the notation and readability of programs cf. section 2.1.4.

(b) Escape Statement

We propose to use identifiers to label scopes to which an escape is to be made rather than, as in Bliss, (Wul 70) requiring the user to indicate the number of scopes from which an escape is to be made. By eliciting a parallel between Bliss escape-statements and Algol W-type case-statements, and between Snip escape-statements and Pascal-type labelled case-statements, we deduce that the

latter form of escape mechanism is more secure.

Operations

From the frequency of error of omission (error type 1.3), we deduce that the use of a space to denote string concatenation causes instability. For example the omission of the alternate operator " | " in "A B" is undetected. The use of a space to denote the null string is similarly unstable.

In Snip, we introduce the symbols "." for concatenation and "NULL" for the null string.

CHAPTER 4

IMPLEMENTATION

4.0 Introduction

In this chapter, we consider the implementation of the Snip language. The method of implementation proposed is to design an abstract machine which is well-suited to modelling the data structures and operations encountered in Snip; and to implement a translator to translate programs written in Snip into equivalent programs for this abstract machine.

In the following sections, we consider and justify the proposed implementation scheme: we consider the design of abstract machines in general, and, in particular, an abstract machine well-suited to modelling the data structures and operations of Snip; we describe briefly the Snip Abstract Machine (SAM), and show how its design has been influenced by, and how it has deviated from the abstract machines AWAM and SIL designed for the implementation of the Algol W and Snobol 4 languages, respectively. We describe aspects of implementation of SAM on the IBM 370/158 computer.

The translator for Snip has not been fully implemented, but we do not envisage any new problem areas other than those mapped out for the incremental section (cf. section 3.2.2.6). Some small sections of the translator have been implemented (cf. Appendix E).

4.1 Implementation Strategy

We propose to implement the Snip language by designing an abstract machine which is well-suited to

modelling the data structures and operations of Snip. This abstract machine may be realised on a real machine by such methods as microprogramming or macro-expansion or interpretation. In conjunction with a translator (which translates Snip programs to equivalent abstract machine programs) it forms the basis of our implementation scheme. To aid portability, we would expect the translator to be written in Snip and (initially) hand-translated to an equivalent abstract machine program. It would then be possible to transport the language to a new receptor machine, simply by implementing the abstract machine on the receptor. Figure 4-1 illustrates this process using the T-diagram notation. (Ear 70). A similar implementation scheme is used in the implementation of Snobol 4 (Gris 72).

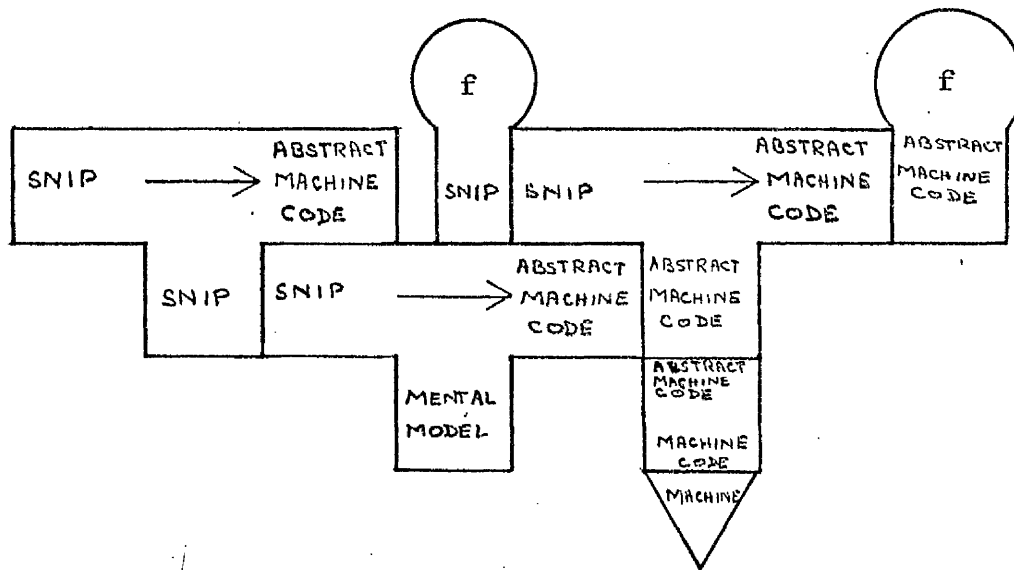
The principle advantages of this scheme are the portability offered (assuming that the abstract machine is readily realisable on a variety of real machines) and the relative ease of translation of Snip programs to equivalent programs for a problem-oriented abstract machine.

4.2 Abstract Machine Modelling

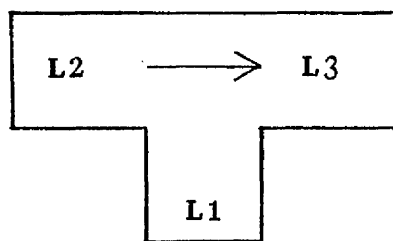
4.2.0

In this section, we consider the principles of abstract machine modelling: we draw heavily on the material of Poole and Waite (Poo 73), in particular, in this brief overview.

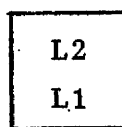
The basic aim of abstract machine modelling is the design of the conceptual structure, memory organisation, registers and operations for a machine which is ideally suited to modelling data structures and operations of a



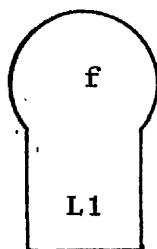
KEY



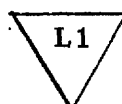
represents a translator written in language L1 to translate source language L2 to target language L3.



represents an interpreter written in language L1 which interprets programs written in language L2.



represents a program written in language L1 which computes function *f*. (We treat translators and interpreters as special case functions)



represents a machine which executes language L1.

FIGURE 4.1

SNIP Implementation Scheme

given programming language.

The design criteria are

- (a) portability of the abstract machine, and
- (b) efficient realisation of the abstract machine on a variety of real machines.

We consider the implications of these design criteria:

4.2.1 Design of Abstract Machines

We show that the portability and efficiency of an abstract machine are determined by

- (a) the relationship between the abstract machine and the language to be modelled, and
- (b) the relationship between the abstract machine and the real machine upon which it is implemented.

An extremely simple model results in high portability, since the model is easily realised on both simple and sophisticated real machines; however, if the language being modelled requires complex operations, then these have to be encoded in terms of the simple model. It is frequently the case that certain real machines have hardware capable of realising these complex operations directly. For example, some real machines are capable of manipulating strings and substrings directly through character or field selection operations, while less sophisticated machines have to simulate these operations by splicing and masking of words, a task involving considerable effort. In order to take advantage of sophisticated hardware, when present, the abstract machine itself must be designed with a higher degree of sophistication or at a "higher level"; however, in this case, portability suffers because of the difficulty of

realising the complex operations of a high level abstract machine on simple hardware.

Poole and Waite (Poo 73) propose that this conflict of design aims be resolved by introducing a hierarchy of abstract machines: at the top of the hierarchy is a high level abstract machine which closely models the language to be implemented; this machine is realised in terms of successively simpler abstract machines which model successively simpler hardware structures cf. figure 4-2A. The abstract machine at the top of the hierarchy is realised when the abstract machine closest to the particular real machine is realised on that real machine. We can thus retain portability and still achieve efficient implementation on sophisticated hardware.

It is important that each abstract machine in the hierarchy be sufficiently transparent to allow efficient implementation of the operations and data structures required at the topmost level, i.e. in the programming language itself (cf. section 3.2.2.2; (Par 72)). In order to keep the size of the abstract machines small and thus aid portability we may be prepared to compromise transparency (and hence efficiency) for operations which are infrequently used (cf. section 3.1.1).

In this thesis, we concern ourselves only with the design of a language-oriented (high level) abstract machine for Snip. We regard the development of a hierarchy of abstract machines for portability as beyond the scope of this thesis (cf. section 0.3). We refer the reader to the development of an abstract machine hierarchy for Algol W and similar languages (Ibr 74). We would hope, neverthe-

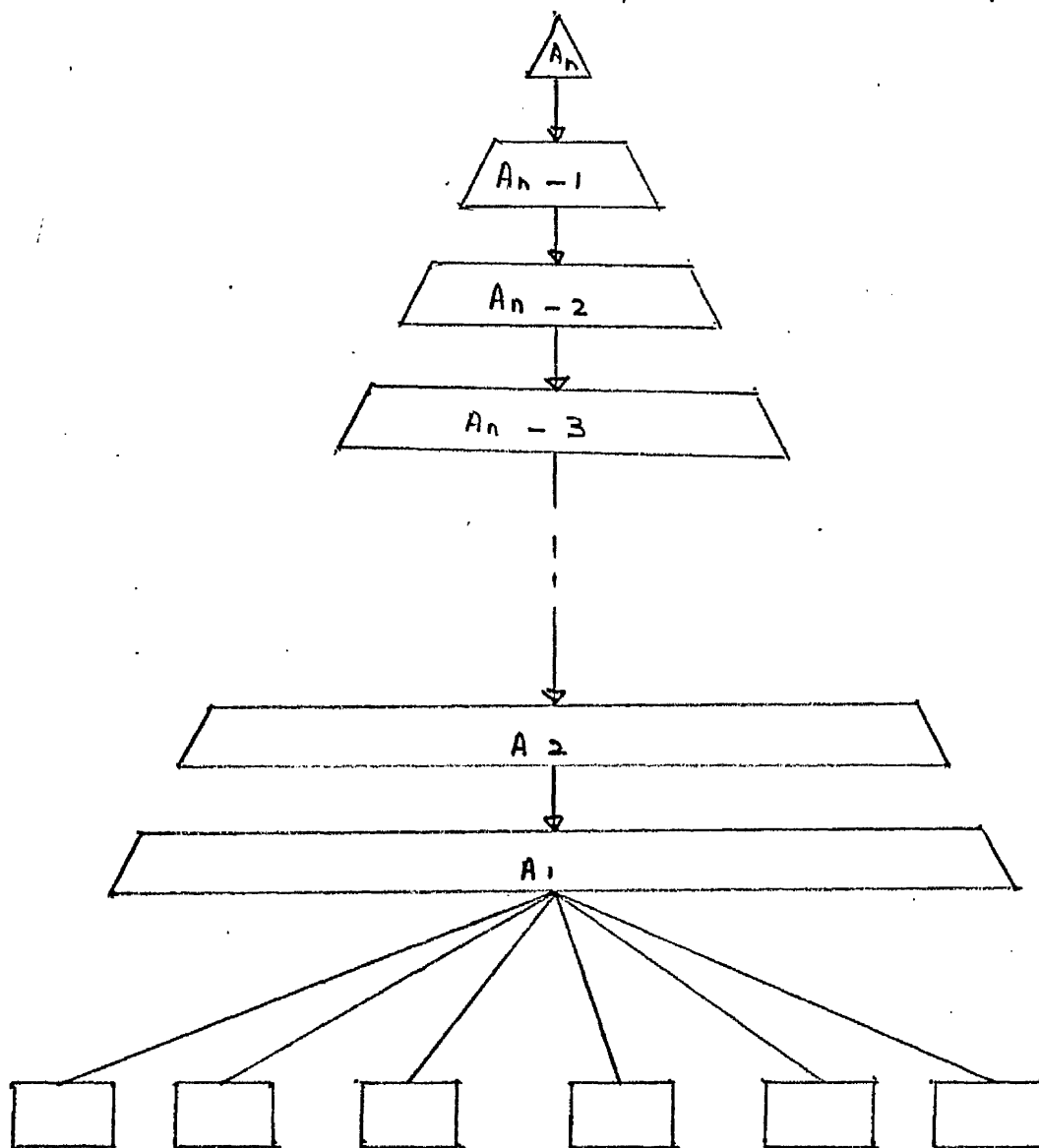


FIGURE 4-2A

ABSTRACT MACHINE HIERARCHY

(FROM POOLE, WAITE [POO73])

4-5

less, that the language-oriented abstract machine designed in this thesis will provide a suitable top level in the development of an abstract machine hierarchy for Snip.

4.3 Design of the Snip Abstract Machine (SAM)

4.3.0

Before considering the decisions involved in designing Snip, we present an overview of the Snip Abstract Machine (SAM), with particular reference to string- and pattern-operations and data structures.

4.3.1 The Snip Abstract Machine

The Snip Abstract Machine (SAM) is considered to have a conceptual structure and a physical realisation of this conceptual structure. This physical realisation is called the architecture. Our approach to conceptual structures and associated terminology closely follows that used in the Abstract Pascal Machine (APM) (Pat 75).

4.3.1.1 Conceptual Structure

The SAM abstract data structures may be conveniently and precisely described by a sub-classification of the well-known "stack" data structure.

In data structure theory, a stack is described as a linear list of records to which further records may be added or deleted at one end only, namely the top of the stack. Any record in the stack may be accessed for reading or writing of information. In the sub-classification described here, this form of stack is known as a Free Stack (FS). The sub-classification is:

- (1) Free Stack (FS)
- (2) Read Only Stack (ROS) in which records may be accessed

for reading only, and in which only the top cell may be written to.

- (3) Push-Down-Store (PDS) in which only the top cell of the stack may be used for reading or writing.

Two qualifiers may be associated with each of these structures:

- (a) A stack is said to be bounded if there is an upper limit to the number of cells which may be occupied at any time.
- (b) An expanding stack is never popped.

The term stack of stacks is used to refer to a stack each cell of which contains a stack.

Activation Records

In the SAM abstract data structures, attention is centred on the activation records and their enclosing structures. When a procedure or pattern is called, an activation record is set up. This record contains all the simple and structured variables local to the scope of the procedure or pattern body, together with the record holding the return address code. An activation record is normally only accessible when the procedure or pattern body is being evaluated.

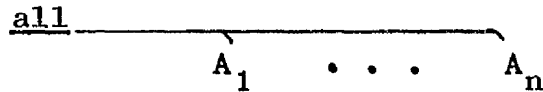
When a local activation record is accessible, the only other activation record which is simultaneously accessible is the one associated with the main program. Thus, if the local activation record contains a pointer to the global activation record, we may regard the stack of activation records as a PDS.

We describe the structure of an activation record using component diagrams. We explain the use of these

diagrams by example:

EXAMPLE 4-1

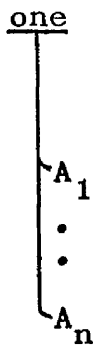
(a) The diagram



specifies the string $A_{i_1} \dots A_{i_n}$ where $i_j \in \{1, \dots, n\}$
and if $i_j = i_k$ then $j = k$.

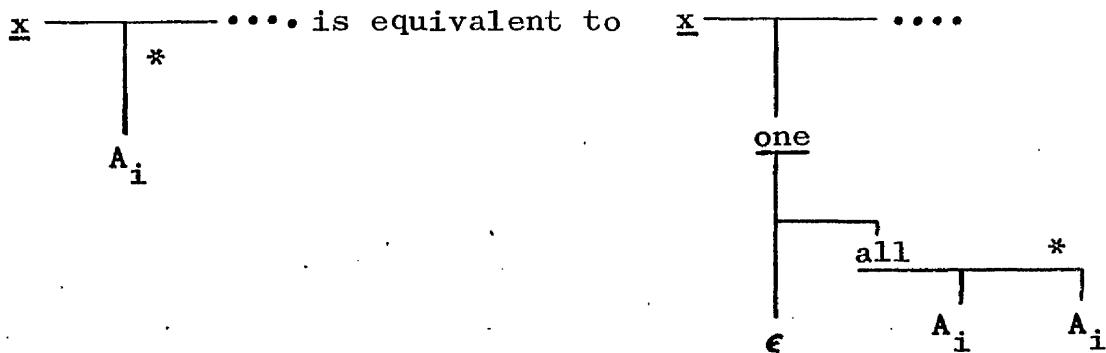
Thus, no ordering of the components is implied.

(b)



specifies the string A_i where $1 \leq i \leq n$

(c)



where ϵ denotes the empty symbol and x is an operator.

The structure of the SAM activation record is shown in figure 4-2B. Those groups of components, for example the temporary result records (TRS's), which form structures which are themselves stack subclasses are indicated by labelling the adjoining arc with a bracketed * symbol, and indicating the stack subclass immediately beneath the entry.

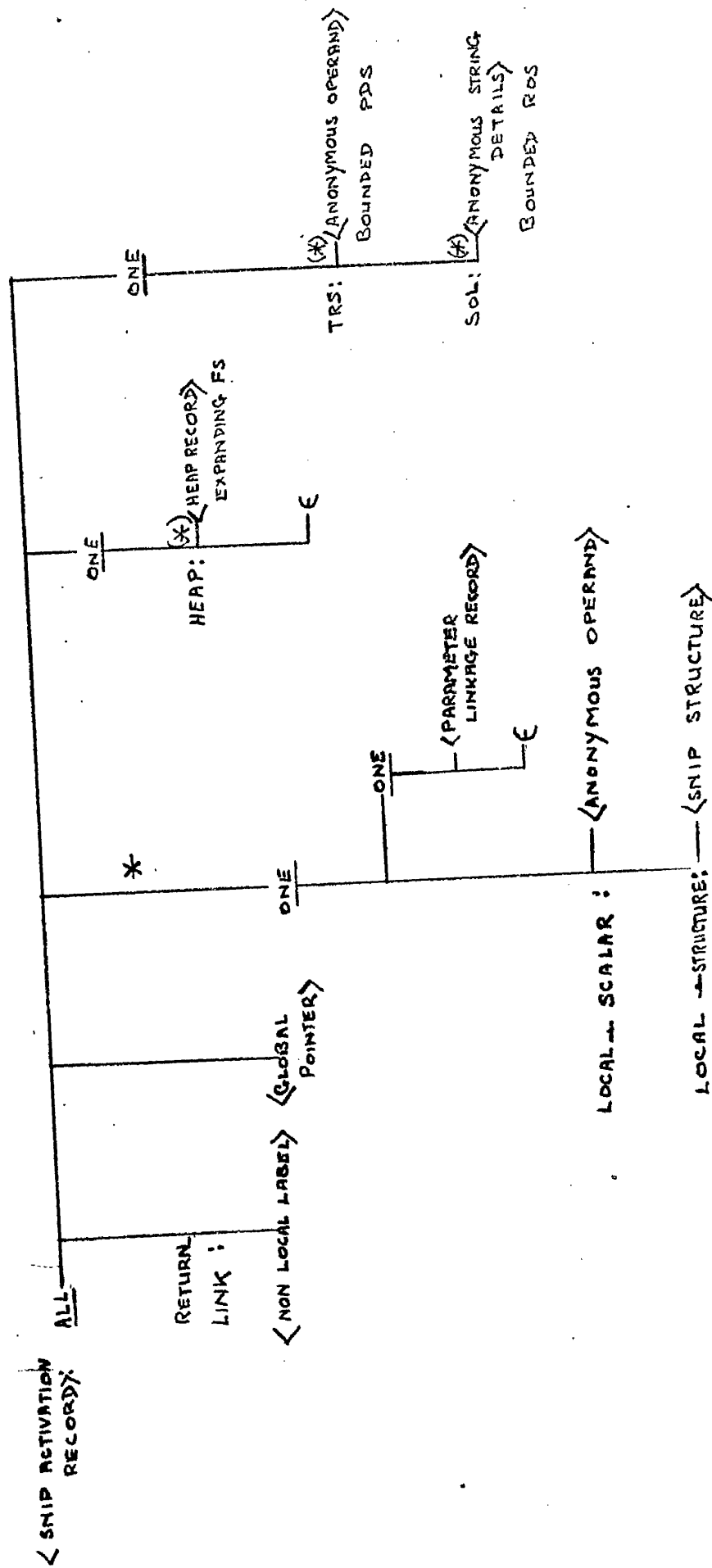


FIGURE 4-2B SNIP ACTIVATION RECORD.

The bounded PDS of <anonymous operands> is the temporary result stack (TRS). The bounded ROS of <anonymous string details> is the string offset and length stack (SOL).

Since the activation records are organised in a stack, it follows that the components of an activation record may be regarded as "separately" stack organised, although these stacks keep in step with the activation record stack, pushing when the activation record pushes and popping when it pops. In fact, when the components are considered to form separate stacks, we may find that these stacks actually behave like one of the sub-classifications of the free stack. It is therefore useful to consider the activation record stack in two different ways:

- (1) as a PDS of activation records which we call the characteristic structure, and
- (2) as nine qualified stacks of distinct activation record components which we call orthogonal components.

The orthogonal components for the SAM orthogonal structure are

- (a) a PDS of return links,
- (b) a PDS of global pointers,
- (c) a PDS of parameter linkage record lists for scalars,
- (d) a PDS of parameter linkage record lists for structures,
- (e) an FS of local scalar lists,
- (f) an FS of local structure lists,
- (g) a PDS of bounded PDS's of temporary results,
- (h) a PDS of bounded ROS's of string offsets and lengths,
- (i) a PDS of expanding FS's of heap variables.

We say that an orthogonal component is characterised by its characteristic structure if it is pushed and popped only when the structure pushes and pops. We say that the component is weakly characterised by its structure if

- (1) it pushes and pops whenever the structure pushes and pops, but may push and pop in between pushes and pops of the structure, or
- (2) it may not push when the structure pushes, and will not pop when the structure pops the record to which the component did not react, or
- (3) it pushes characteristically, but does not pop.

The heap, SOL and TRS are therefore weakly characterised by the activation record structure, while all other SAM components are characterised by the characteristic structure.

SAM Conceptual Structure

The SAM machine consists of a controller, a special register called the program counter (PC), an accumulator (ACC), an activation record stack, a program space, a table space, a pattern template space and a special register called the pattern node counter (SSA). Each activation record contains three (possibly null) stacks: the Temporary Result PDS, the String Offset and Length ROS, the heap FS. ACC forms the head cell of the Temporary Result PDS (TRS).

These components, together with the permitted data pathways amongst them are shown in figure 4-3. We elaborate the purpose of each SAM component:

- (1) The SAM program is a linear list of SAM orders selected from the SAM order codes (cf. below).

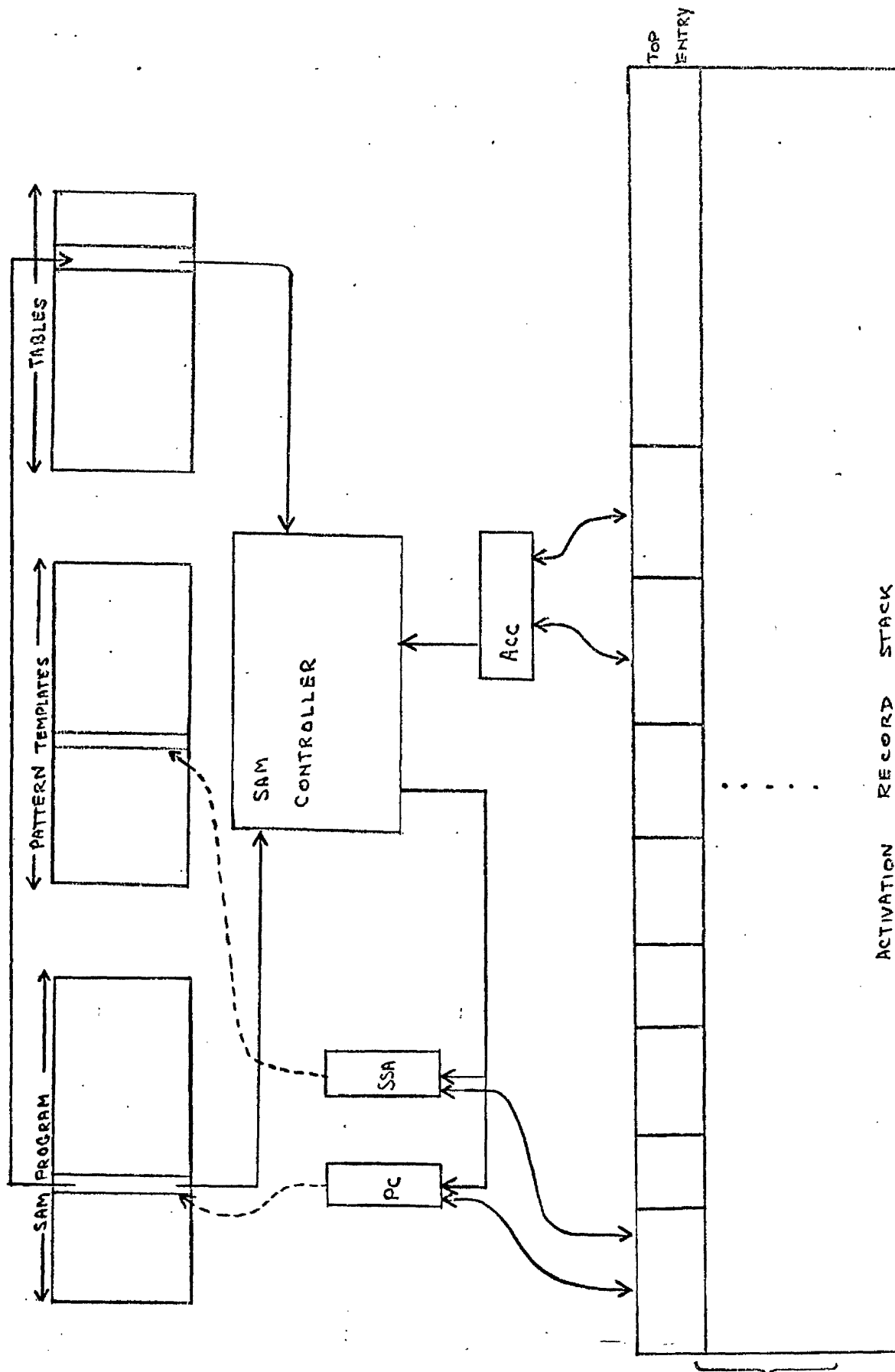


FIGURE 4-3

SAM CONCEPTUAL STRUCTURE

- (2) The program counter PC is connected to the Return-Link PDS and is itself, effectively the top cell of this PDS. PC points to the next instruction to be taken from SAM program space and obeyed. To effect subroutine linkages PC is pushed on initiation and popped on termination.
- (3) The pattern template space is a linear list of pattern nodes.
- (4) The pattern node counter SSA is also connected to the Return-Link PDS and is itself effectively the top cell of this PDS during pattern matching. SSA points to the next pattern node to be interpreted. To effect pattern linkages, SSA is pushed on initiation of a subpattern and popped on termination.
- (5) ACC acts as the top cell of the Temporary Result PDS (TRS) and is therefore connected to it.
- (6) During evaluation of string expressions, ACC acts as the top cell of the String Offset and Length ROS (SOL) which contains substring offsets and lengths.
- (7) The heap is a free stack of strings.
- (8) Tables consist of a linear list of identifier-defining and constant-specifying descriptors. All SAM orders which need to determine variable addresses or constant values contain pointers to table descriptors.

Addresses of variables in the activation record stack consist of a pair (n , m) where \underline{n} indicates whether the variable lies in the topmost activation record (i.e. variable is local) or in the lowest activation record (i.e. variable is global), and \underline{m} identifies the offset within the record.

Arithmetic operators take their operands from ACC and, optionally, the top cell of the temporary result PDS (TRS); they return the result to ACC.

Any string initialised is pushed on to the heap stack.

4.3.1.2 Architecture

This section comprises a physical realisation of the structures of the conceptual machine cf. figure 4-4. This physical realisation is peculiar to implementation on the IBM 370/158 machine or, more precisely, the IBM 370/158 as seen through Algol W and arises out of one possible interpretation of the conceptual structure. A one-level store and an unlimited number of special registers may therefore be assumed.

4.3.1.2.1 Memory

The structures of the conceptual machine are mapped on to a single contiguous store which is divided into three segments:-

- (1) the read/execute segment (RXS) containing the SAM program,
- (2) the read/write segment (RWS) on to which is mapped the return address PDS, the data stack, the temporary result stack and the heap, and
- (3) the read segment (RS) on to which is mapped the tables and the pattern templates.

All three segments are directly addressed from 1 and have a maximum size M (say) of SAM locations. The term static address, or simply address refers to the position of a location in one of these segments.

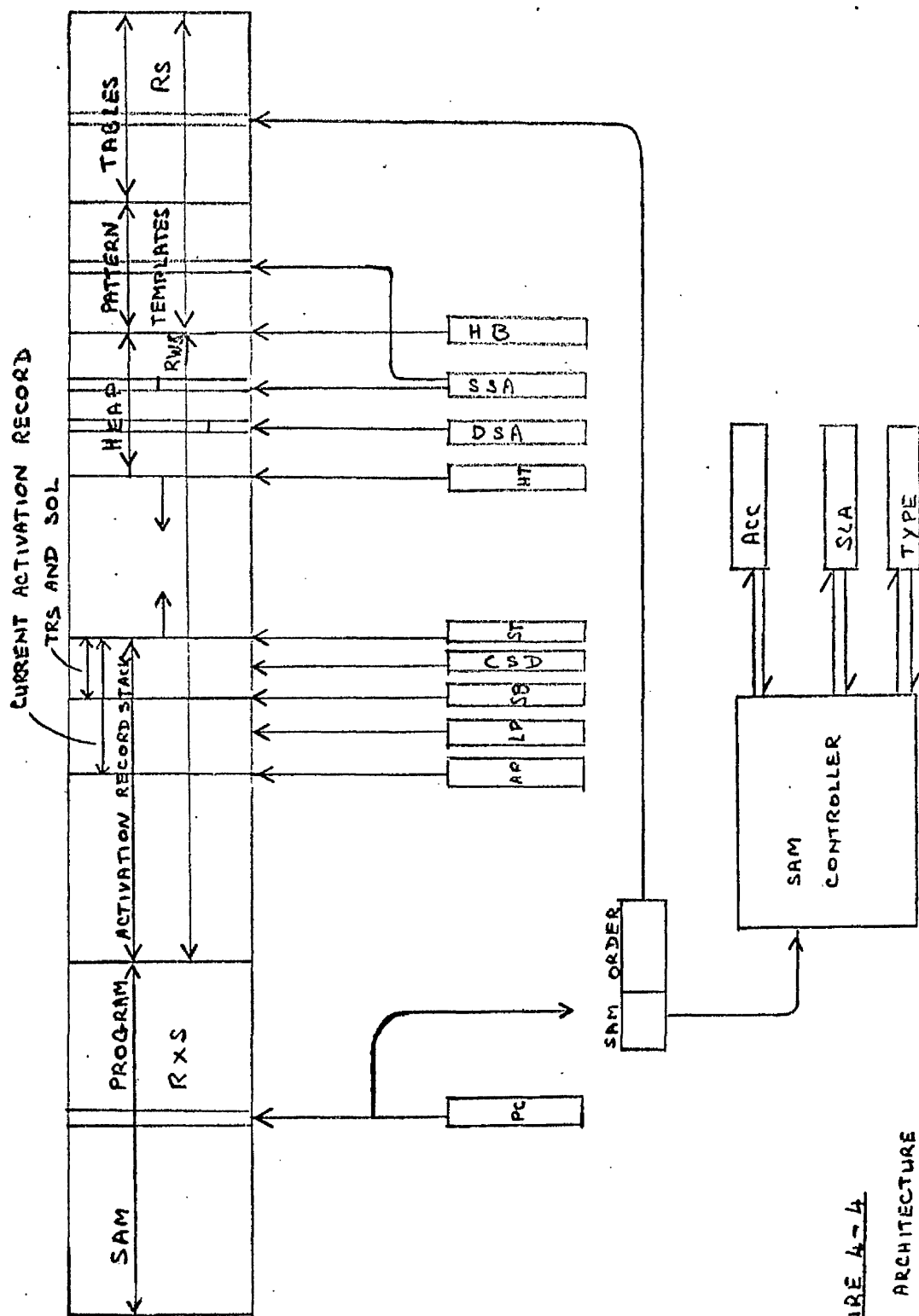


FIGURE 4-4
SAM ARCHITECTURE

4.3.1.2.2 Addressing

Five addressing formats are used:

- (1) Static addresses in the range 1 to M for code addresses, identifier- and constant-descriptor base addresses in the tables, data addresses compiled at run time and heap string base addresses;
- (2) an address of value zero which may refer either to ACC or to the top cell of the TRS (depending on the context);
- (3) SOL addresses s, where s is the displacement from the base address of the SOL;
- (4) local addresses l, where l is the displacement from the base address of the current activation record; and
- (5) Heap addresses of strings which incorporate a static word address together with a character offset within the word.

4.3.1.2.3 Registers

Thirteen special registers are available to the SAM controller, in addition to RXS, RWS, RS:

- (1) The active (local) pointer AP points to the current local activation record;
- (2) the local pointer LP points to the activation record for a procedure or pattern which is about to be, or just has been current;
- (3) the stack base pointer SB holds the static address of the first location in the TRS, following the local variable space of the current incarnation of the current scope;
- (4) the stack top pointer ST holds the static address of the highest address (occupied by the TRS) in the RWS segment;

- (5) The heap top HT holds the static address of the highest address in the RWS not occupied by the heap i.e. the location preceding the lowest location occupied by the string allocated space at the top of the heap;
- (6) The heap base HB holds the static address of the highest RWS location occupied by the heap;
- (7) The program counter PC holds the static address of the next instruction in the RXS to be fetched and interpreted;
- (8) The accumulator ACC holds the operand of monadic operators, one of the operands of dyadic operators, and the result of (integer) function calls;
- (9) The source string address register SSA contains either the address (byte or word and character offset) and length of the substring which is to be copied (or compared) or, the address of the node (of a pattern template) next to be interpreted;
- (10) The destination string address register DSA contains the address and length of the destination string area (or the descriptor of the first operand of a dyadic (string) operator, and (optionally) the address of the descriptor);
- (11) The string length accumulator SLA holds the length of a string expression during assignment;
- (12) A pointer CSD to the current-string details in the SOL stack; and
- (13) The type field which holds the type number of the operand in ACC.

4.3.1.2.4 Representation of Snip Data Structures within SAM

The simple data types in Snip, integer and boolean are represented in SAM by core locations allocated within

the activation record stack.

The structured data types in Snip, string, vector and file are represented in SAM by a descriptor, allocated within the activation record stack, which points to and describes the representation of the string, vector or file on the heap or in another part of the record activation stack.

The form of a descriptor depends on the variable type:

String Descriptor:

BH	CU	LS
----	----	----

String File Descriptor:

B	CU	LS	P
---	----	----	---

Integer File
Descriptor:

B	P
---	---

The descriptor for a vector of string elements consists of a series of string descriptors, one for each string element, where

BH is a pointer to the string area on the heap,

CU is the string cursor (or file buffer cursor),

LS is the string length (or file buffer length),

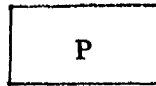
B is a pointer to the file data area,

P indicates whether the file is in core or on backing store,

UB is the (normalised) upper bound of the vector.

The run-time representation of procedure parameters is distinguished by the call attribute and the parameter type. Different kinds of parameters are stored in the parameter space in different ways:

var-parameters; and
const-parameters of
structured types (but
excluding string type):



Const-parameters of
simple data types :



where

P is a pointer to the parameter or its descriptor in the
activation record to which the parameter is local;

N is the amount of space required to store the value of the
formal parameter type or its descriptor.

4.3.1.2.5 Stack Frame

Each active scope (procedure, main program or
pattern) has at least one activation record allocated to it.
The structure of this record is known as the stack frame.

Figure 4-5 shows the layout of an activation record
for

- (a) a procedure or the main program block,
- (b) a pattern.

In the link space,

- (1) RA is the return address for a procedure or subpattern
call,
- (2) DP is the dynamic pointer,
- (3) SBS is the location in which the current SB is stored
if an activation record is laid on top of the current
record in the stack.

For pattern stack frames only,

- (4) SSAS, DSAS are the locations in which DSA and SSA are
(respectively) stored when an action primitive of the
pattern is called, and

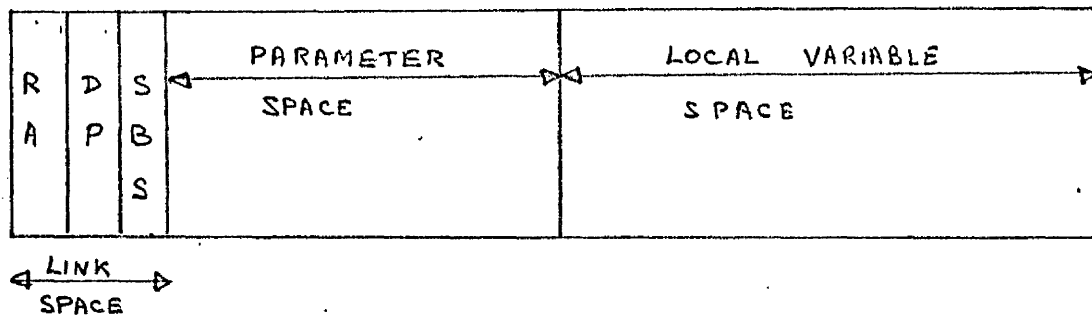


FIGURE 4-5(a) STACKFRAME (PROCEDURE OR MAIN PROGRAM)

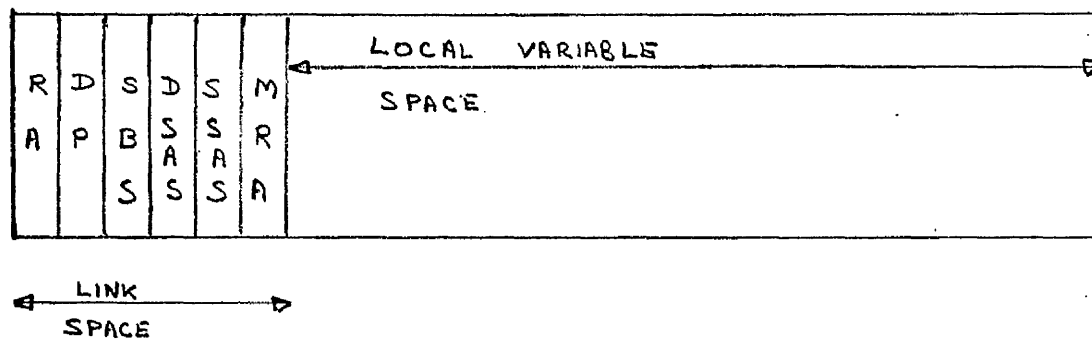


FIGURE 4-5(b) STACK FRAME (PATTERN)

(5) MRA is the location in which the return address for an action primitive call is stored.

The parameter space is that part of the activation record occupied by the actual items stored for the procedure parameters.

4.3.1.2.6 Instruction Groups

We find it convenient to classify SAM orders into the instruction groups listed below. We describe briefly the action taken by instructions in each group.

Monadic Load Group: load an integer or boolean value into ACC.

Subscripting Group: (optionally) load a subscript into ACC; check that the value in ACC lies within the subscript bounds of the given vector variable.

Load and Operate Group: (a) load an integer or boolean value into ACC and execute the specified operation using ACC and the top cell of the TRS as operands, or
(b) load a string descriptor into register SSA and (optionally) execute the specified operation using the strings described by (descriptors in) DSA and SSA as operands.

Store Group: store a value or string descriptor held in a specified register to the location whose address is given as parameter.
In the case of string variables the heap area occupied by an unrequired

	string value may be garbaged.
String Operation	(optionally) allocate area for a new
Initialisation	string value on the heap; set up a
Group:	string descriptor in register DSA to
	describe the area to which a new string
	value is to be copied.
Load and Store	load the descriptor of the string
String Group:	given as parameter into register SSA
	(in the case of a substring, this
	descriptor is modified according to
	the offset and length contained in
	the SOL stack); copy the string so
	described, to the area described by
	register DSA.
Jump Group:	effect control transfers.
Block and Pattern	effect the initialisation and
Control Group:	termination of use of an activation
	record.
Procedure Control	effect the initialisation and termin-
Group:	ation of procedures and their
	associated activation records.
Parameter Passing	implement parameter passing (for
Group:	those cases not covered by previous
	order groups).

The individual instructions, their mnemonics, parameters and associated meanings are described in detail in appendix B. The discussion of register organisation and usage contained in the following section is sufficient for our purposes here.

4.3.1.2.7 Register Organisation and Usage for Snip Operations

All arithmetic and boolean operations use register ACC and the temporary result stack (TRS) to store intermediate results.

String operations, however, record intermediate results in main memory and use registers to hold only the string descriptors of operands and intermediate results. There may be up to two "current" string descriptors loaded, one in each of registers DSA and SSA. The offsets and lengths of substrings are held in the SOL stack. During string comparisons, DSA holds the string descriptor of the first operand, and SSA the descriptor of the second. During pattern matching operations, DSA holds the descriptor of the second. During pattern matching operations, DSA holds the descriptor of the subject string, and SSA the descriptor of the pattern template.

During string assignment, it is usually necessary to copy the components of the assigned (string) expression to a new area on the heap. An instruction "INITVS" allocates an appropriate amount of space on the heap for the string expression (whose length is calculated in register SLA) and sets up a descriptor in register DSA to describe this area. Instruction "LSTSTR" loads register SSA with the descriptor of the (next) component of the string expression, and copies this component to the area indicated by DSA. When the string expression has been evaluated (in this manner) in the designated heap area, an instruction "STDESCR" marks as garbage, the area on the heap previously occupied by the assignment string variable, and replaces the

descriptor for this variable by a descriptor for this new heap area. Under special circumstances in substring replacement or insertion or deletion or append operations, it is possible to alter the value of a string without copying its value to a new position on the heap.

4.3.2 Design of SAM

4.3.2.0

In this section, we identify the principal decisions taken in the design of SAM, paying particular attention to string and pattern matching operations and data structures. Poole and Waite (Poo 73) note that there is a common core of data types and operators applicable to most abstract machines; in designing SAM, we find that we can build on the abstract machines SIL (Gris 72) and AWAM (Pat 74) designed for the implementation of the Snobol 4 and Algol W languages, respectively.

The representation of strings and patterns and the design of SAM orders to effect string and pattern matching operations have been influenced by SIL. However, since the Snobol 4 philosophy is to allow as much flexibility at run time as possible (cf. section 2.5), we find we can considerably optimise the design of SAM for the less flexible Snip constructs. The concept of registers, not present in SIL, is also introduced in SAM.

With the introduction of registers, and given the simple block structure and less flexible nature of Snip at run time, we find that the AWAM arithmetic operations, record activation stack and stack maintenance operations are more akin to the modelling of the corresponding Snip constructs. Here again, the structures can be considerably

simplified in SAM.

We now consider

- (a) how Snip data structures may be suitably represented in SAM, and hence, the decisions involved in the design of SAM data structures, and
- (b) the design of orders and registers well suited to the manipulation of these data structures in SAM.

4.3.2.1 Representation of Snip Data Structures

We consider strings, patterns, local variables and parameters.

(a) Strings

We consider the expected usage of (combinations of) operations on strings and thus determine an appropriate machine representation.

(1) Expected Usage

We have indicated (cf. section 3.2), that we expect strings to contract and expand frequently and that in general no upper limit on string length will be known at compile-time.

In application to amending or processing of text files, the deletion, replacement and insertion of substrings in file-component strings will be frequent. It is probable that a combination of these operations may be applied to a single file component. However, since we expect file-component strings to be relatively short and of known maximum length, the consequences of some loss of transparency at this level are probably not severe.

In a string processing language we should expect heavy usage of pattern matching, and large overheads (Wai 73). Since pattern matching essentially involves string

comparison and substring selection efficiency of these operations will be of considerable importance.

(2) Machine Representation of Strings

We consider string organisation and storage allocation on the basis of expected usage of operations considered above.

String Organisation

The most common forms of organisation are the linked list and the packed string. A packed organisation is more efficient for string concatenation and comparison if the string is long, as it results in fewer memory accesses (Wai 73). This is particularly true on byte-oriented machines allowing character selection (Poo 73). A list organisation is more efficient for insertion and deletion of substrings. In view of the expected usage (cf. above), a packed string organisation is more appropriate in SAM.

Storage Allocation

Two forms of storage allocation are commonly used for packed strings (Wai 73):

- (i) A certain fixed area is allocated to each string at compile-time. Strings thus have a fixed-maximum length (FML): they can grow or contract subject to this upper bound, or
- (ii) At run-time, each string is allocated an area sufficient for its current length. If a string grows, it is copied to a (different) larger area. Strings thus have a dynamic-maximum length (DML), with upper bound related to the capacity of the real machine.

FML strings allow clear advantages for append and

substring-replacement operations as updating operations may be used (cf. section 3.1.1.1); the capacity to grow is, however, severely limited. This form of string is appropriate for file-component strings (cf. above).

DML strings allow much greater flexibility, but lose the FML string advantages for append and substring-replacement operations. Overheads may be considerably increased by the need for garbage collection and compaction to deal with storage fragmentation. This form of string is appropriate to general Snip strings.

Within either system, we still have the option of sharing identical strings and substrings. From the point of view of efficiency, this is worthwhile for multiple assignments (we have none in Snip) and substring assignments (Wai 73). Substring assignments will frequently occur during or after pattern matching (cf. above). If shared substrings are allowed then it is more difficult to allow append or deletion operations or substring replacement by updating the existing string value.

In conclusion, we have rather unsatisfactorily proposed the inclusion of two forms of string representation, namely fixed-maximum length or FML strings and dynamic-maximum length or DML strings. We might consider combining the two systems, attempting to retain the advantage of each. Intermediate systems of this form do not, however, appear to have any clear advantages (Wai 73).

FML strings can be allocated a fixed-length region in the appropriate activation record. This is not, however, the case for DML strings. We consider these separately.

DML Strings

Since DML strings may expand or contract dynamically, space cannot be conveniently allocated to them on a stack structure (such as the activation record stack); a heap structure in which an appropriate amount of space can be dynamically allocated to a string and for which some form of storage regeneration exists, would provide a suitable means of representing DML strings. (McK 70; Grie 71).

Space on the heap is allocated according to string size. If a new value is assigned to the string variable, an appropriate amount of space is allocated in a new heap area; and the old area is garbaged. If only part of the string value is altered, by deletion or replacement of a substring, it may not be necessary to copy new string value to a new heap area (its bounds are merely altered).

This organisation is similar to the strategy used in SIL to allocate storage space to strings and other constructs in Snobol 4.

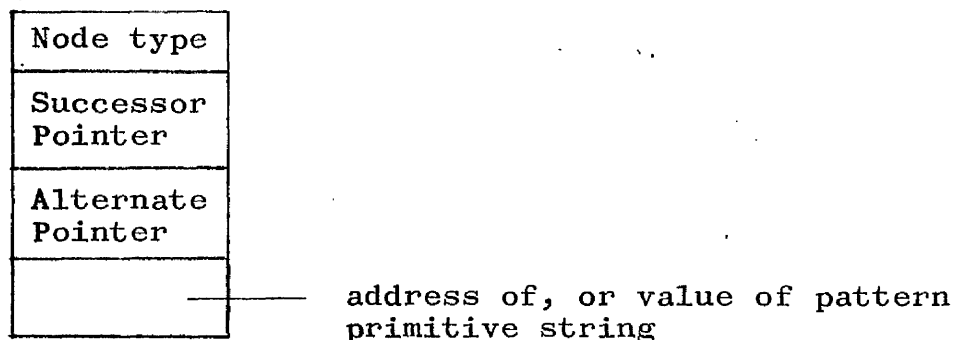
String Descriptors

Since DML strings may occupy different positions on the heap at different points during the execution of a program (as described above), it is necessary to maintain, in the appropriate activation record, a pointer to the heap area currently occupied by a string, and an indication of the length of the string. We call this pointer-and-length pair, the string descriptor. Each (incarnation of an) activation record contains string descriptors, one for each string variable which is local to that activation record. We find it convenient to associate the string cursor with the corresponding string descriptor.

(b) Patterns

Patterns may be altered only at extension definition time, prior to execution of the translated SAM object program (cf. section 3.2.2.3). Thus, pattern templates are fixed at (abstract machine) run time, and can therefore be allocated a fixed area in SAM memory.

We can represent a pattern template by a series of nodes (one corresponding to each pattern primitive string (cf. section A.11) of the template). These nodes are chained by "alternate-node" and "successor-node" pointer corresponding to alternate and successor sequences of the template they represent. Each node consists of the 4-tuple:



The number of alternate and successor pointers required is reduced to a maximum of one each by the chaining of alternates and successors (cf. SIL (Gris 72)).

(c) Local Variables and Parameters

The design of the record activation stack to represent local variables and parameters is influenced by AWAM. The linkage for activation of procedures, functions and patterns is similarly influenced.

A similar construct does, in fact exist in SIL (for procedures and functions only), but it is cumbersome and, conceptually, less well organised: an activation

record is effectively created on the heap, and pushed on to a stack only when the function or procedure is called recursively.

The linkage space of an activation record (cf. figure 4-5(a)) requires a dynamic pointer if the activation records are of different lengths. However, since no non-local variables, other than global variables are accessible, no static pointer is required.

The linkage space of an activation record for a pattern (cf. figure 4-5(b)) must record, in addition, the contents of registers DSA and SSA and the address of the current (match) instruction during the call of an action primitive of a pattern: DSA and SSA are recorded as they may be required in string or pattern operations within the action primitive; the match return address (MRA) is recorded because action primitives use the activation record corresponding to the pattern in which they are defined (they do not initialise a new activation record).

By the introduction of an activation record associated with each call of a pattern, we can implement the (compile time) construction of new patterns defined in terms of existing patterns, as well as recursive patterns without the need for explicit copying (contrast SIL (Gris 72)).

Since backup is not permitted during pattern matching in Snip (cf. section 3.2.2.3), we do not require to create a stack of "untried" alternates as backup points (contrast SIL).

4.3.2.2 SAM Registers and Operations

The SIL machine is a low level machine and has no

concept of registers: instructions refer directly to memory locations for their operands. For this reason, SIL is readily implemented on even simple machines cf. section 4.2; by the same token, however, SIL introduces inefficiency in terms of redundant code (unnecessary transfers to/from memory from/to registers) on most real machines. In this section, we consider the design of suitable registers and operations for SAM. We consider first the general form of SAM orders and then the orders and registers associated with certain data types:

(a) SAM Orders

General

(1) Polymorphism SAM orders are polymorphous in the sense, that, as far as possible, a single order is used to take a particular action irrespective of the type(s) of the operand(s). The abstract machine orders are thus designed at a high level, and are divorced from the idiosyncrasies of particular real machines.

(2) Postfix Code

Griswold (Gris 72) finds it expedient to express SIL orders in prefix form, because of the flexibility of Snobol 4. However, orders in prefix code define a highly nested and complex evaluation structure which is not particularly suited to realisation on most existing real machines. We find it possible in SAM to use the more natural postfix (or reverse polish) notation.

Arithmetic Operations and Registers

Arithmetic operations include an (optional) "load"

operation, since we expect the sequence of operations "load" followed by an arithmetic operation to be relatively common (cf. AWAM (Pat 74)). On some real machines these orders may be implemented by a single instruction. However, since we do not expect arithmetic operations to be heavily used in Snip, we do not provide "reverse" instructions (for non-commutative operations) to minimise TRS usage nor do we provide instructions to optimise special cases.

String Operations and Registers

It is not, in general, possible to load string values into registers. (Indeed, for certain string operations, concatenation for example, the use of registers to hold intermediate results is quite inappropriate.) For this reason, string orders refer to string descriptors to access their operands. In SIL, descriptors for the (current) string operands are stored in locations in the data area; in SAM, they are held in registers SSA and DSA. We consider the operations of string comparison and string assignment individually:

(1) String Comparison For string comparison operations, the descriptor of the first operand is held in register DSA; the descriptor of the second in register SSA. We keep these operations at a high level, by avoiding explicit definition of precisely how strings are compared. (This is possible because any register or workspace used in comparison is local to these instructions and is not required by subsequent instructions). We also avoid specifying the form of a string address (byte or word) held in register DSA or SSA.

(2) String Assignment and Concatenation Before a string assignment, register DSA is set up to describe the new/destination heap area. During assignment or concatenation, the descriptor for the (next) component of the assignment expression is loaded into SSA since this register is free; the component is then copied to the area indicated by DSA, register DSA being updated to receive the next component (if any). Since the loading of register SSA and the copying of a component always occur in sequence, we use a single "high level" instruction "LSTSTR" to carry out this action. Register DSA is required if there is more than one component (i.e. concatenation in the assignment expression), to point to the (next) free section in new heap area.

Considering the expected Snip usage, we felt it likely that large unallocated areas or "holes" will frequently develop in heap between the pointers HB and HT (cf. figure 4-4). We attempt to reduce this problem (and thereby, hopefully, to reduce the frequency of storage compaction) by attempting the infilling of holes on string assignments. This infilling is most easily organised if the length of the assignment expression is calculated before the assignment takes place. Since the expression may contain substrings, a stack of intermediate results (substring offsets and lengths) may be built up on the TRS during calculation of the expression length. The offset and length pairs are accessed in reverse order (hence the need for register CSD to point to the offset and length for the "current" string) and so we rename the TRS, when it is used in this manner, as the SOL.

Pattern Matching and Registers

In order to handle pattern matching, we require to maintain a pointer to the next node to be interpreted in the pattern template, and a pointer to the current position in the string being matched. At the start of a pattern match, the descriptor of the subject string is loaded into DSA. Register SSA is used to point to the next pattern node. During the pattern match, the cursor in DSA and the node address in SSA are appropriately updated.

Local Variable, Parameter and Activation Record Stack

Pointers

Pointer registers AP and LP are introduced to allow convenient access to local variable, parameter and linkage space in the current activation record and in the activation record which is about to be (or just has been) current, respectively (cf. figure 4-4).

Since no non-local variables (other than globals) may be accessed, it is not necessary to maintain a display register pointing to the base addresses of activation records for textually enclosing scopes.

4.4 Implementation of SAM

4.4.0 Introduction

An interpretive version of the Snip Abstract Machine was implemented on the IBM 370/158 computer. The high level language Algol W (Bau 71) was chosen as a suitable implementation medium; its constructs for bit and string handling were appealing, although it proved impossible to make use of this latter facility because of restrictions imposed by the language.

In this section, we describe and discuss the major aspects of this implementation. We consider first the run-time store and interpreter organisation, and subsequently the representation and organisation of SAM data structures and orders.

4.4.1 Run-Time Store and Interpreter Organisation

Store is laid out in a single Algol W array CORE, and registers are defined as individual global variables identified by the register name (cf. figure 4-6). Storage allocation is controlled by an Algol W subroutine, ALLOCATE, which controls the limits of the activation record stack and the heap. In addition to space for SAM data structures, the array CORE includes space for the FREE LIST which is used in heap organisation (cf. section 4.4.3).

The basic interpretation process is handled by a routine CONTROLLER which simulates the action of the SAM controller. CONTROLLER fetches and decodes the next SAM instruction from memory (pointed to by PC); this instruction and its parameters are decoded and passed to the appropriate "instruction-interpreting routine" which simulates the action of the instruction; there is an instruction-interpreting routine corresponding to each SAM instruction. On completion, instruction routines relinquish control to CONTROLLER, which continues the processing of SAM orders, as above.

4.4.2 Representation of Snip Data Structures and Statements

SAM Orders

Each order occupies one or two (32 bit) Algol W

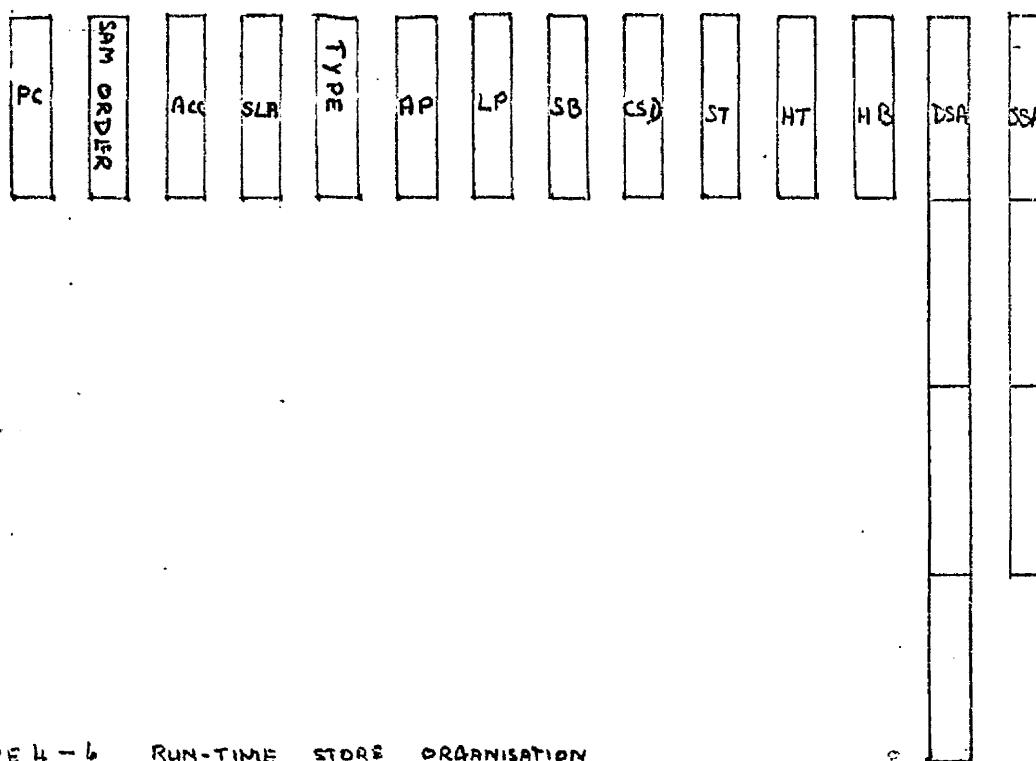
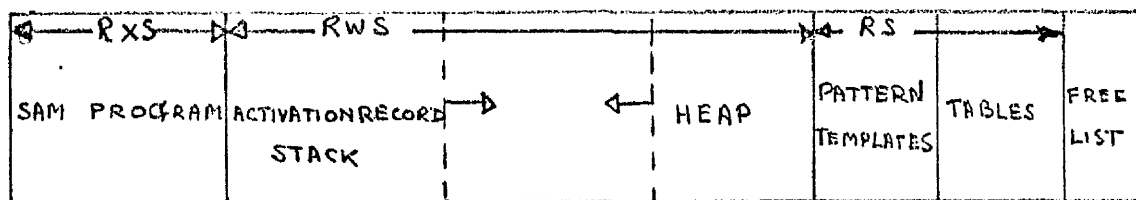
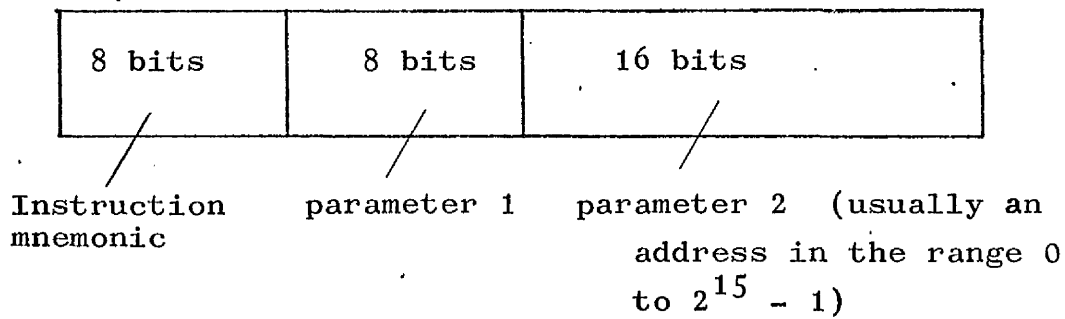
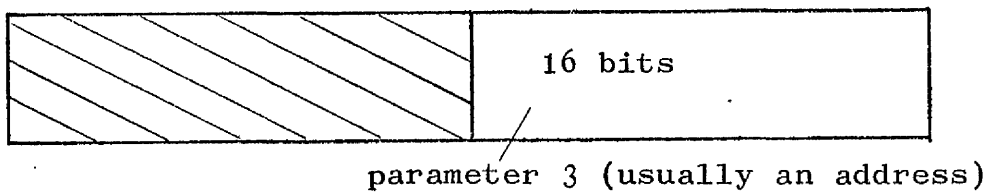


FIGURE 4-6 RUN-TIME STORE ORGANISATION

words. The first word of a SAM order is always of the form:



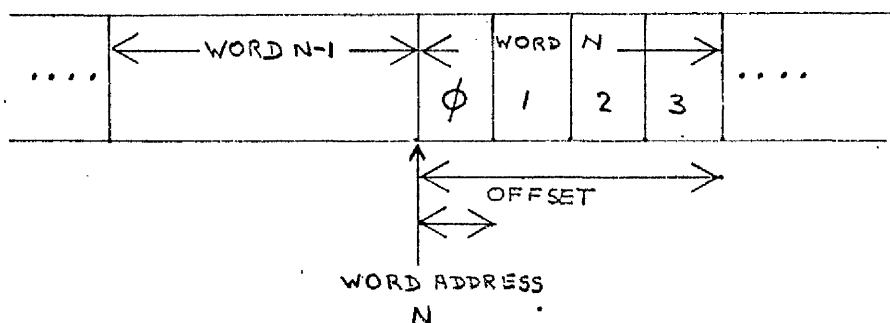
The second word of an instruction (if present) is always of the form:



Instructions are decoded (unpacked) by using bit masks and operations.

Strings

Because of the restrictions on the size of Algol W strings, it is not possible to use byte addressing. Strings are thus represented as a series of encoded integers (four 8-bit characters to a word) in the heap area of array CORE. String addresses, therefore, consist of a word address followed by the character offset within that word. The character offset lies in the range 0 .. 3 as shown in the figure:



Substrings are fetched/stored using appropriate bit masks and operations.

Registers DSA and SSA are therefore of the form:

DSA	WORD ADDRESS	OFFSET	STRING LENGTH	DESCRIPTOR ADDRESS
SSA	WORD ADDRESS	OFFSET	STRING LENGTH	

Files

In this implementation, files are treated as in-core files and allocated space in region RS (cf. figure 4-6). Backing store files can be handled by the inclusion of externally-defined Fortran routines.

The standard files INPUT and OUTPUT are simulated by the Algol W input and output streams, only the file buffer being allocated space in SAM memory. Since strings are stored as sequences of encoded integers (cf. foregoing) conversion to/from Algol W strings from/to integers is required during input and output. Although this type change does not in fact alter the form of the data, it does, in (the Algol W) implementation involve explicit manipulation of data, since there is no simple means of effecting this type change in Algol W. This obstacle can be circumvented by rewriting input/output handling routines as externally-defined Fortran procedures.

Patterns

Pattern templates are laid out from high core address to low core address, each node of the template

being represented by a block of 2 or more words. The size of the block is dependent on the node type (cf. figure 4-7). Action primitives and the null string have no alternates since they are always successful. A null successor or alternate pointer is represented by the address "-1". The set of chained nodes representing a single template is mapped on to a contiguous area of core.

We illustrate this representation of pattern templates by considering the Snip patterns declared as follows:

Example 4-2

PATTERN E;

```

BEGIN
    <:-T> . { "+" . <:-T> }
END;

```

PATTERN T;

```

BEGIN
    <:-F> . { "*" . <:-F> }
END;

```

PATTERN F;

```

BEGIN
    "ID" | "(" . <:-E> . ")"
END;

```

Figure 4-8 shows the representation of the patterns E, T, F. (In practice, the nodes are mapped on to a contiguous area of memory). This representation differs from the corresponding representation of Snobol 4 patterns:

- (a) A pointer is maintained to each pattern template to allow shared and recursive use of patterns without

Node TypeNode Representation

String Constant :

Node Type = 1
Successor Address
Alternate Address
String length
String value

Null string :

Node Type = 2
Successor Address

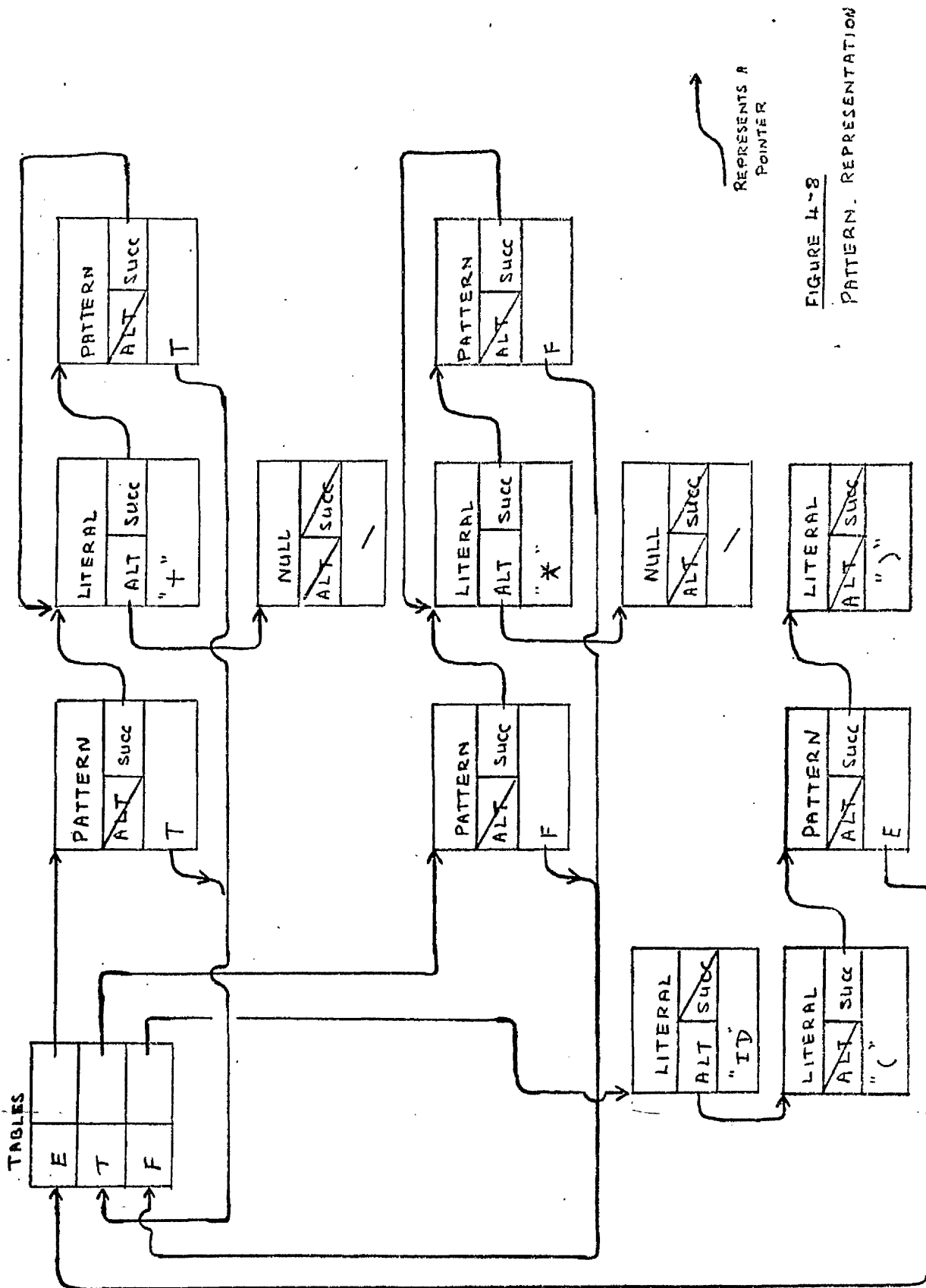
Subpattern :

Node Type = 3
Successor Address
Alternate Address
Table Pointer

Action Primitive :

Node Type = 4
Successor Address
Table Pointer

FIGURE 4-7Node Representation



copying.

- (b) Constructs enclosed in brackets { } include a "self pointer" as final successor (to allow repetition) and an initial NULL alternate.

Tables

Run time tables include an entry for each of the following occurrences in the source program:

declared variables	standard variables
implicitly declared cursors and buffers	constants
procedure or function declarations	pattern declarations
labels (for structured statements)	

As well as providing the SAM controller with the type and address (and other information) about variables, table information is useful for debugging purposes.

The form and size of a table entry depends on the type of the variable or construct which it describes. The information contained in table entries is summarised in figure 4-9. Each entry commences with an identification tag, a type and an address. Table entries for formal parameters are similar to those for local and global variables, but with the following changes:

- Variable-Parameters:
- (a) The type field is increased by 100, and
 - (b) CORE (PTR-2) contains the address of the address of the value or descriptor.
- Constant-Parameters:
- (a) The type field is increased by 200, and

FIGURE 4-9 RUN-TIME TABLES (FOR VARIABLES AND OTHER CONSTRUCTS)

Entry Variable Type	CORE (PTR)	CORE (PTR-1)	CORE (PTR-2)	CORE (PTR-3)	CORE (PTR-4)	CORE (PTR-5)	CORE (PTR-6)	CORE (PTR-7)	CORE (PTR-8)
Boolean or Integer	Tag	Type = $\begin{cases} 1 \\ 2 \end{cases}$	Address of value	-	-	-	-	-	-
Vector of Integer	Tag	Type = 3	Base address of values	normalised maximum bound	-	-	-	-	-
(Integer) File Buffer	Tag	Type = 4	Address of descriptor	-	-	-	-	-	-
(Vector) Cursor	Tag	Type = 5	Address of value	-	-	-	-	-	-
(String/Buffer) Cursor	Tag	Type = 6	address of value	-	-	-	-	-	-
Integer Constant	Tag	Type = 7	actual value	-	-	-	-	-	-
DML String	Tag	Type = 8	address of descriptor	-	-	-	-	-	-
Vector of String	Tag	Type = 9	address of descriptor	normalised maximum bound	-	-	-	-	-
FML String	Tag	Type = 10	address of descriptor	maximum length	-	-	-	-	-
String Constant	Tag	Type = 11	address of value	actual length	-	-	-	-	-
(Integer) File	Tag	Type = 12	address of descriptor	mode	first component address	last component address	last used component	-	-
(String) File	Tag	Type = 13	address of descriptor	mode	first component address	last component address	last used component	maximum component length	-
Pattern	Tag	Type = 14	address of template	local variable space	address of first string descriptor	number of string descriptors	size of template	-	-
Procedure	Tag	Type = 15	address of procedure body	local variable and parameter space	"	"	-	-	-
Function	Tag	Type = 16	address of function body	"	"	"	-	-	-
Structured Statement	Tag	Type = 17	address of end of statement	-	-	-	-	-	-
Main Program Block	Tag	Type = 18	first word of code	base of activation record stack	"	"	base address of heap	base address of free list	address of first table entry

(b) CORE (PTR-2) contains the address of the address of the descriptor for structured variables, and the address of the descriptor or value for string variables and scalar variables.

4.4.3 Instruction-Interpreting Routines

We do not intend to describe each instruction-interpreting routine in detail, since, in general, there is a one-to-one correspondence between these routines and the SAM orders; in most cases, they are adequately defined by the specification of the SAM order which they interpret. We consider instead, the following problems associated with string storage and string operations:

String Operations

(a) String Comparisons

In order to compare two strings S1 and S2 (say), we have to subdivide them into substrings of more manageable lengths. Since we cannot use byte operations (cf. section 4.4.2), comparison with substrings of length one word provides the most efficient implementation. However, this is only reasonably possible when the offsets of the first characters of S1 and S2 are equal. If the offsets are unequal, we compare the strings character by character.

For (in)equality, a comparison of the length of the two strings may yield a quick result.

(b) String Copying

During string concatenation and assignment, strings are frequently copied from one heap area to another. As

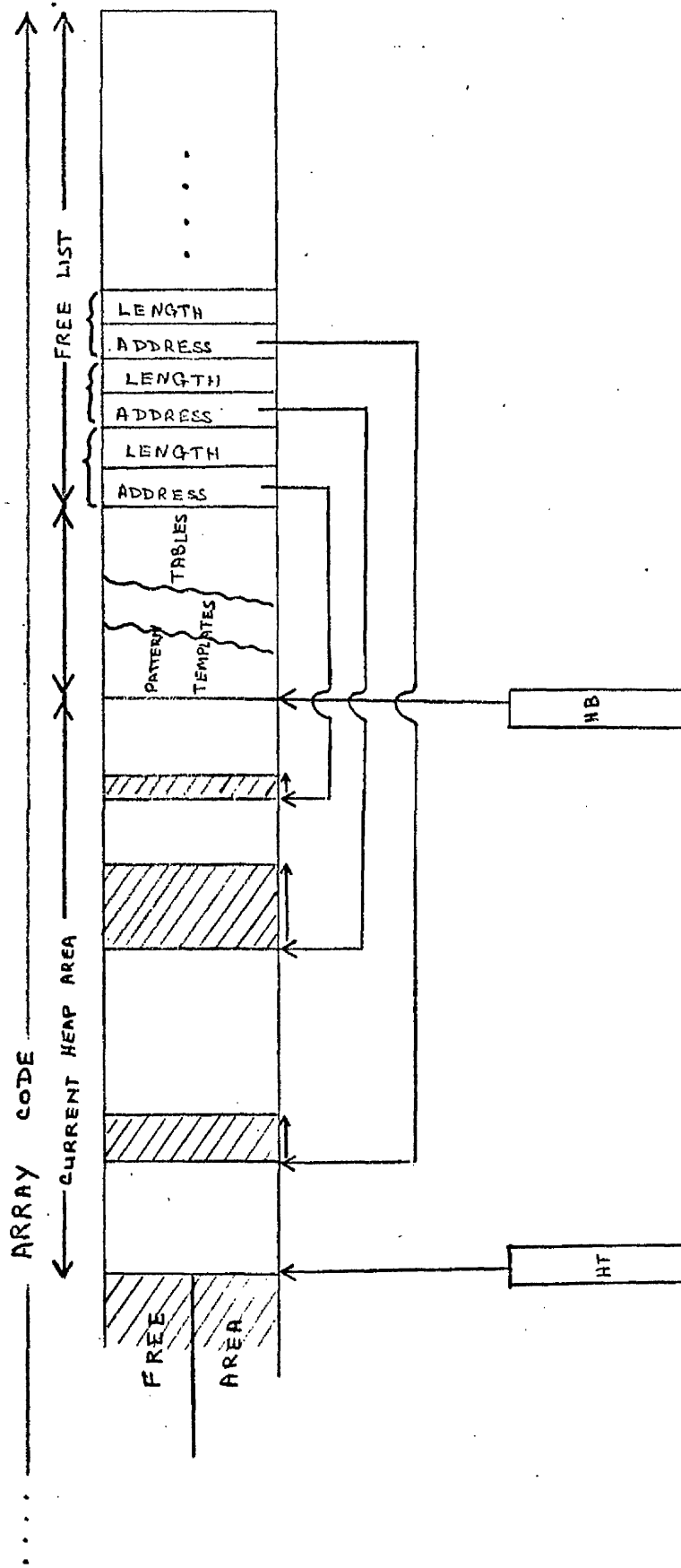
in the case of string comparisons, we find it necessary to subdivide strings during copying. Here again, this process is relatively easy to carry out at word level; but this is again only possible when the offsets of the first characters of the string source and destination areas are equal. If these offsets are unequal, we find it necessary to subdivide and re-merge partial words during copying. This is clearly a very time-consuming process.

Heap Organisation

Heap organisation is summarised in figure 4-10. HT and HB point to the highest-allocated heap address and the heap base address, respectively. Strings are allocated an appropriate amount of space on the heap by routine ALLOCATE. Conceptually, at least, we find it simpler to record strings, running from high heap (low physical) address to low heap (high physical) address. We recall (cf. section 4.3.2.2) that we attempt to infill "holes" appearing on the heap. We organise this by maintaining a FREE LIST to describe these holes: each cell of the list contains a pointer to a hole and the length of the hole. When routine ALLOCATE is called, the FREELIST is scanned for a suitable hole. If there is no suitable hole, space is allocated at the top of the heap. If space available to the heap for expansion becomes short and the available holes are small fragments of store, then storage compaction may be worth while.

Compaction Process:

During compaction, strings above the LOW_GARBAGE_POINT (which marks the lowest garbage area on the heap) are copied "downwards" in the heap to fill holes (cf.



REPRESENTS
A "HOLE"

FIGURE 4-10 HEAP ORGANISATION

figure 4-11). Strings are copied strictly in order, from the one with the lowest to the one with the highest heap address, in order to prevent overwriting. An ordered list must therefore be set up, each cell of which contains the address of a string descriptor and the base address of the heap area occupied by the string. String descriptors are updated during the compaction process. Similar compaction schemes are used in XPL (McK 70) and Snobol 4 (Gris 72).

4.4.4 Conclusions from SAM Implementation

The implementation highlighted some errors in the conception of the abstract machine, causing a certain amount of re-design; it demonstrated, however, that the revised abstract machine is feasible. Some small sections of the translator were hand coded in SAM and successfully executed (cf. appendix E).

4.5 The Translator

4.5.0

We do not intend to present a manual for the implementation of a Snip translator, but merely to illustrate (in a simple-minded fashion) how the more unusual features of Snip might be handled. In particular, we refer to the processing of metalanguage and augment texts.

Transition diagrams for Snip syntax may be useful at this stage and are presented in appendix D.

4.5.1 Processing of Metalanguage Text

We consider first the processing of metalanguage or definition text. By allowing the definition of extensions to a language L, say, we are, in effect, allowing

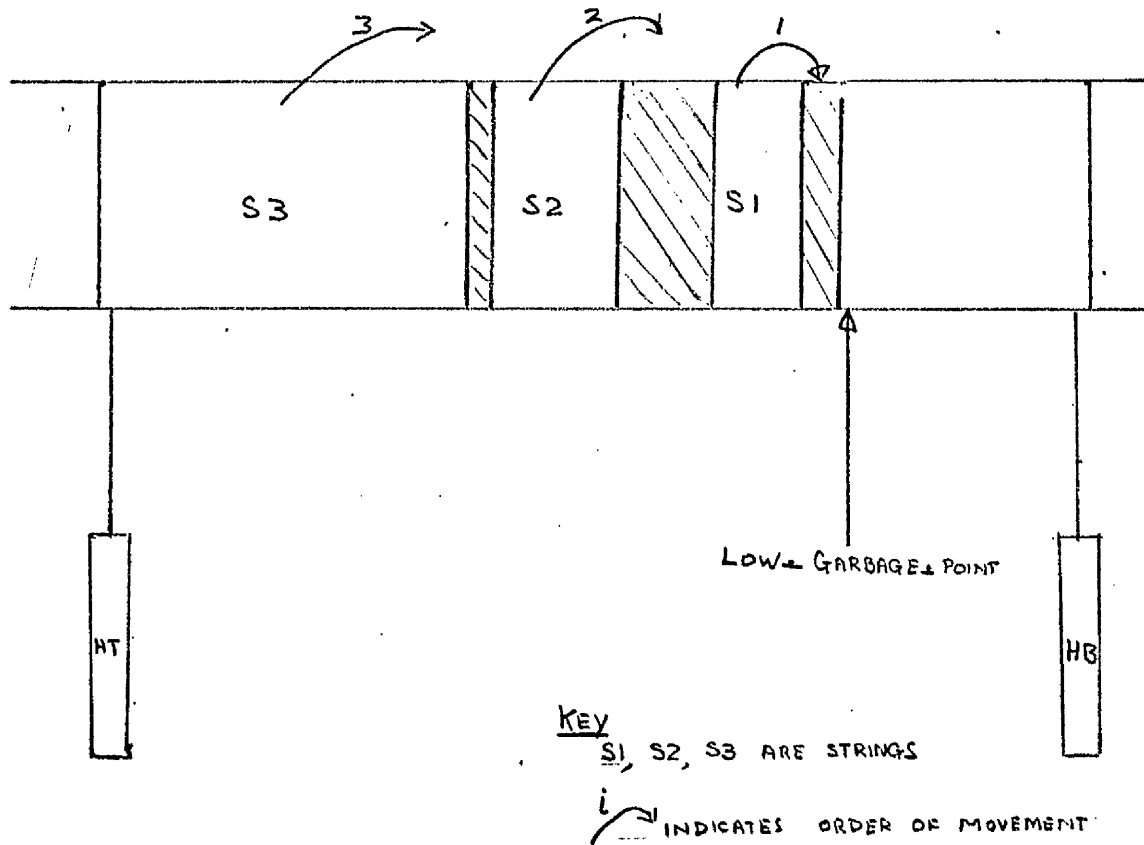


FIGURE 4-11 STORAGE COMPACTION

some form of "incremental" modification of the compiler for L, albeit in a highly restricted and abstracted form. We thus transform the compiler for L into a compiler for an extended version of L, L1, say.

We envisage that this compiler modification will be carried out by

- (a) writing the compiler for L in Snip, and
- (b) by allowing the Snip compiler to incrementally modify programs it compiles.

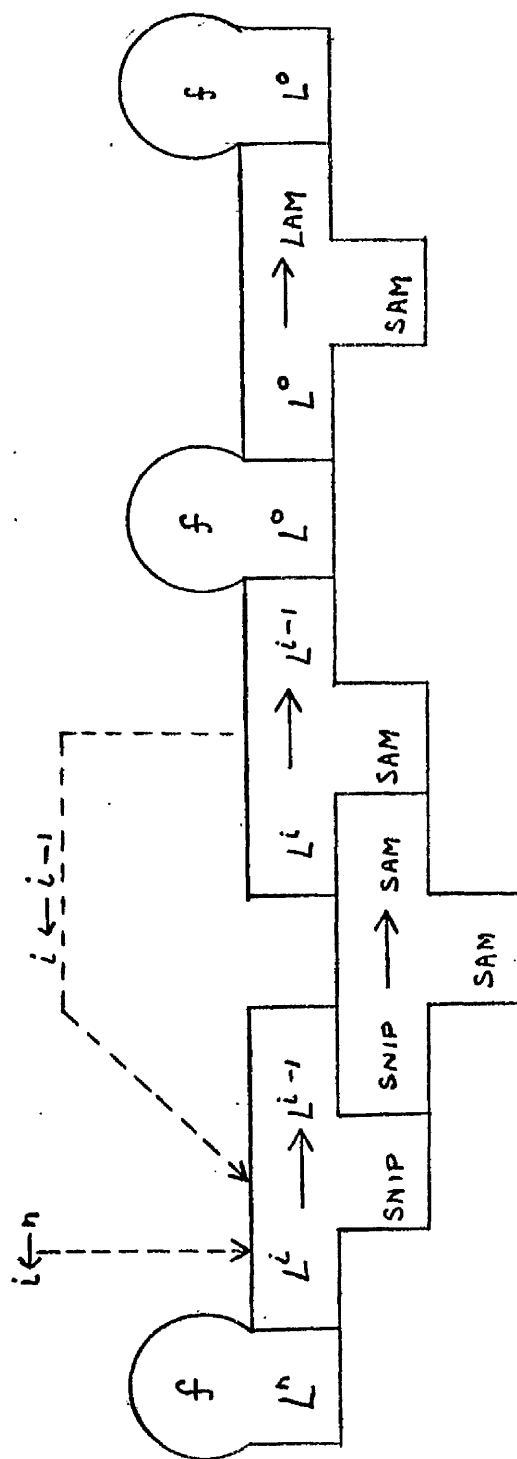
The extension mechanism does not then have to be re-written for each new language which is to be extended.

Figure 4-12 illustrates this process of extension.

Language L may be Snip itself.

As noted in section 3.2.2.6, we allow incremental definition of global variables and procedures, as well as certain controlled modifications to existing structure strings. We do not, in contrast to relatively powerful incremental compiler systems (cf. Mitchell (Mit 70)) allow incremental definition or modification of individual statements of the program being compiled. The principal distinction between an incremental compilation system and the process described here, is, however, the restriction that modification of the program (or compiler) being compiled may be effected only after compilation is completed, and not during the compilation process. Such modification is, in addition, constrained to occur only in the global context of the program being compiled (cf. section 3.2.2.3): it may not be related to the block structure of the program.

We consider the various incremental constructs



KEY

LAM \equiv ABSTRACT MACHINE FOR LANGUAGE L

$L^0 \equiv L$

$L^i, 1 \leq i \leq n \equiv$ DIFFERENT EXTENDED VERSIONS OF L

-----> INDICATES POSSIBLE REPETITIONS

FIGURE 4-12 EXTENSION OF LANGUAGE L

individually:

Global Variables

Incremental declaration of global variables is a simple matter if organised before the data structures of the corresponding abstract machine program are set up. Nor should it be a complex matter to flag and prevent corruption of (the values of) existing global variables.

Procedures

Incremental compilation of procedures may also be simply handled, provided we delay the binding of the abstract machine data structures so that code for the procedure may be placed above the code for the main program segment (cf. figure 4-4).

Existing context-sensitive syntax prevents the overwriting of any existing procedure.

Extension Definitions

The processing of extension definitions poses more complex problems, because in this case, we must actually modify existing patterns. In order to avoid the overheads of modifying the abstract machine structures representing patterns (cf. section 3.2.2.3), it is advisable to modify patterns before they are bound to the abstract machine. We might modify a compile-time representation of patterns, or perhaps even the source text itself. Both the define- and the take-statement are fairly readily implemented by this means. In each case, we must determine that the modified structure remains LL(1) in form (cf. appendix C). In the latter case, we must additionally ensure that the original precedence structure is retained intact, after

modification of the pattern (cf. section 3.2.2.6).

4.5.2 Processing of Augment Text

In this section, we consider the processing of augment text to produce semantically equivalent base or extended text. We consider, in other words, the generation of a semantically equivalent substitution string.

Example 4-3

Statements of the form:

"REPEAT <statement> UNTIL <boolean expression> "

might be defined by the Snip substitution string:

```
"  DO : BL
    |
    |   <statement> ;
    |   IF <boolean expression> THEN LEAVE BL FI
    |
    OD "
```

In normal circumstances, the substitution string will belong to the same syntactic class as the extension itself. Thus, in the example above, both extension and substitution string belong to the syntactic class, <statement> .

When a section of augment text corresponding to an extension is recognised during compilation, the appropriate substitution string is (usually) generated. The compilation process must be so organised that it continues recognition, by scanning this newly substituted string; the recogniser should continue as though it had failed (so far) to recognise the current syntactic class goal. Thus, in example 4-2, the recogniser should continue searching for the goal, <statement> , and scanning should

re-commence on the substitution string "D0".

For a recogniser which allows back-up (e.g. Imp (Iro 70)), this re-scanning of the substitution string might easily be absorbed into the existing mechanism, merely by indicating a "match failure" to the recogniser. However, the LL(1) recogniser used in Snip does not allow back-up (cf. section 2.4) and would not tolerate a failure signal of this form.

We illustrate one possible solution to this problem by way of example:

Example 4-4

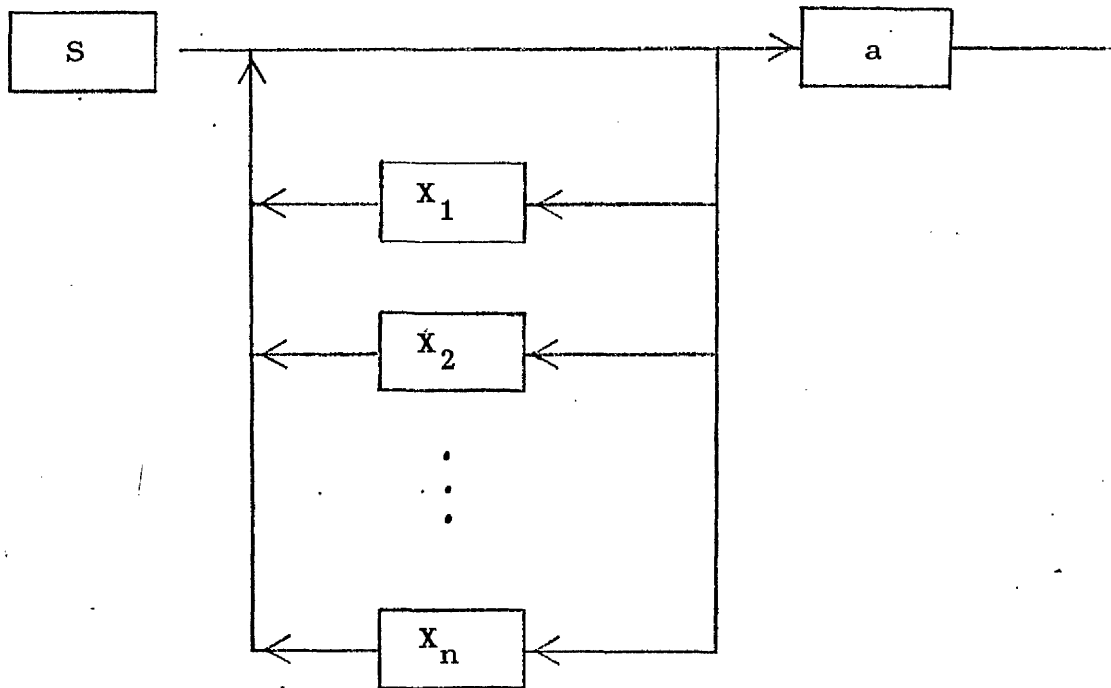
Suppose a particular rule of the LL(1) grammar describing some base language is of the form

$$S ::= a$$

where S is a non-terminal and a is a sequence of terminals and non-terminals. Suppose also that we wish to define extensions of the form $X_1, X_2 \dots X_n$ (where X_i , $1 \leq i \leq n$, is any sequence of terminals and non-terminals) to the syntactic class S . In order to ensure correct scanning of substitution strings, we might modify the rule as follows:

$$S ::= a \mid X_1 \{ X_1 \mid X_2 \dots \mid X_n \}^* a \mid X_2 \{ X_1 \mid X_2 \dots \mid X_n \}^* a \mid \dots \mid X_n \{ X_1 \mid X_2 \dots \mid X_n \}^* a$$

It is perhaps more obvious from the corresponding transition diagram that this rule does in fact specify the correct recognition sequence.



Example 4-5

We consider the definition of a repeat-statement as an extension to the syntactic class, $\langle \text{statement} \rangle$ cf. example 4-3. The transition diagrams for $\langle \text{statement} \rangle$ before and after extension are shown in figures 4-13(a) and (b), respectively.

We recall (cf. section 2.4) that we expect parameters (other than base language constructs) of extensions to be called by value. In order to handle value substitutions, it will in certain cases, be necessary to maintain a hierarchy of buffers, each containing (distinct) generated or partially generated substitution strings. We illustrate this once again by example.

Example 4-6

Consider the statement

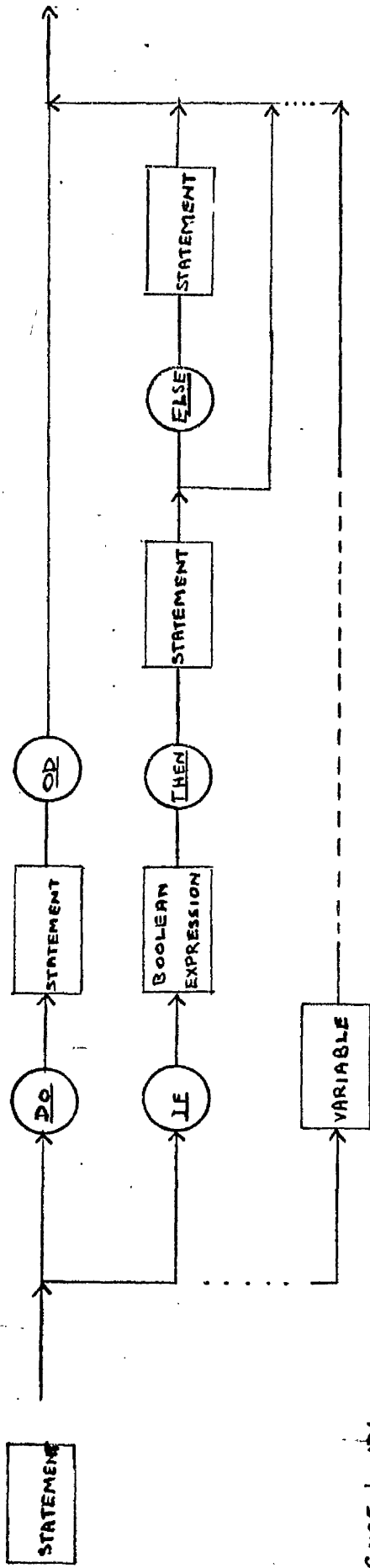


FIGURE 4-13(a)

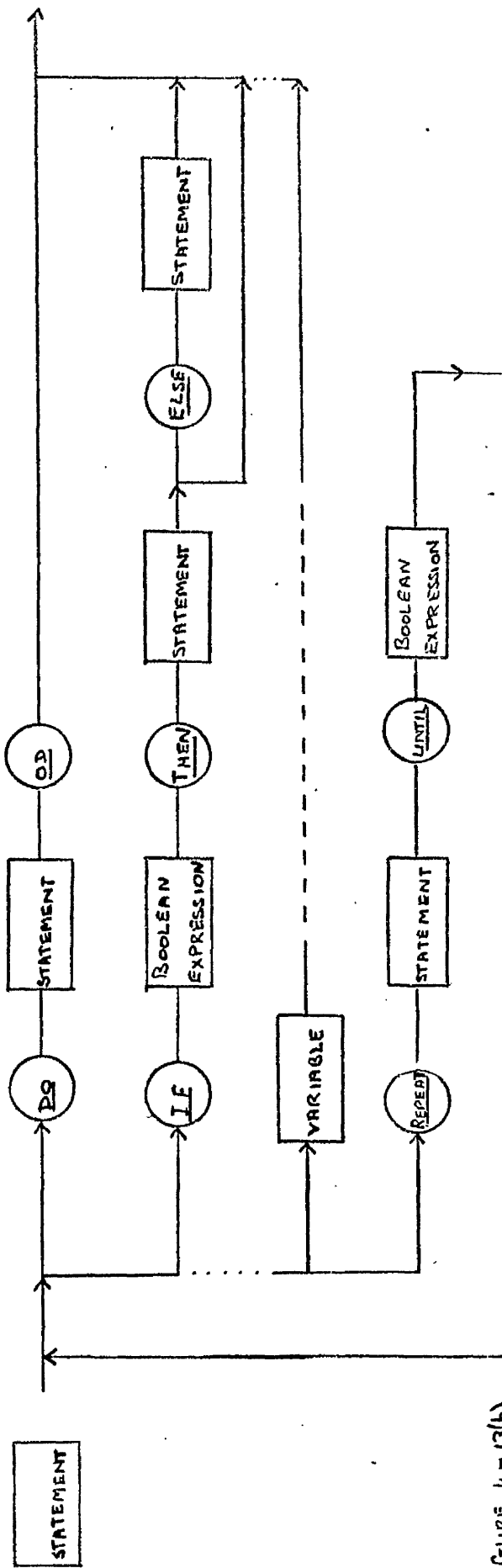


FIGURE 4-13(b)

```

"REPEAT
  A INC B
UNTIL A > B  "

```

where repeat-statement is defined as in example 4-3 and
inc-statement is defined to have the form

```
" <variable> INC <expression> "
```

and the meaning

```
" <variable> := <variable> + <expression> "
```

During recognition of the above statement,
substitution strings for the repeat-statement and for
the inc-statement must be generated (at least conceptually)
in separate buffers.

We now consider a skeleton program to illustrate
a simple-minded translator organisation. We must first,
however, consider two associated problems:

(a) Scanning the substitution string

Example 4-7

We re-consider example 4-4. As noted above, different
substitution strings will be generated in different buffers.
There is however, no means of indicating in Snip that
pattern matching is to continue on some other buffer string;
indeed, such a facility is perhaps inappropriate to a
secure string processing language.

However, the rule

$$S ::= a \mid x_1 \{ x_1 \mid x_2 \dots \mid x_n \}^* a \mid \dots \mid x_n \{ x_1 \mid x_2 \dots \mid x_n \}^* a$$

is equivalent to the rule

$$S ::= a \mid x_1 S \mid \dots \mid x_n S$$

We find that we can thus circumvent the above-noted difficulty by organising extended syntactic classes in the form

$$S ::= a \mid X_1 \mid \dots \mid X_n$$

provided we invoke the match operation recursively on rule S after recognition of any extension.

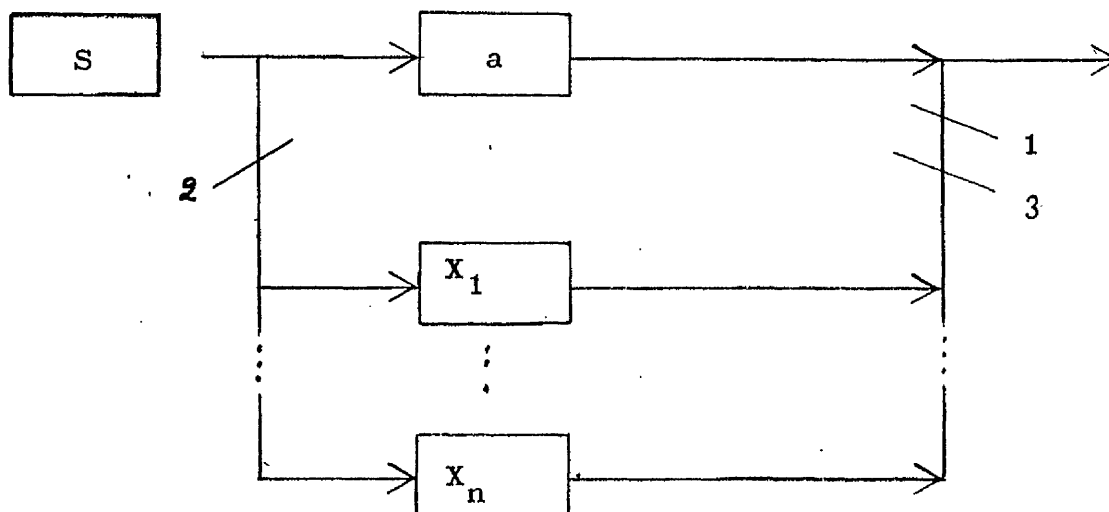
This may be automatically handled by the define-statement or take-statement, without the need of user intervention.

(b) Generation of Abstract Machine Code

When constructs of the base language appear as parameters to some extension, no code is generated for these constructs. However, when base text is recognised in its own right, appropriate (abstract) machine code should be generated. This problem is fairly readily solved by the setting of an "inhibit code" flag during the recognition of an extension.

Example 4-8

We consider once again example 4-4. The extended transition diagram is as follows:



At point (2) on this diagram, the `code_inhibit_flag` is set; at point (3), it is cleared.

At point (1) a recursive call on S is effected (cf. example 4-7).

Once again, the flag can be automatically set and cleared by the `define-statement` or `take-statement`, thus freeing the user from unnecessary effort.

We are now in a position to proceed with the illustration of a simple-minded translator organisation.

Example 4-9

This example defines recognition of a simple compound statement which consists of combinations of `if-statements` and `assignment statements`. A `while-statement` is introduced by extension. The program is written in Snip-like form (cf. figure 4-14).

Code is generated for base language constructs unless the `code_inhibit_flag` is set. Routine "gencode" is assumed to handle this code generation.

Routine SUBST is assumed to generate a substitution string in appropriate buffers. We also assume the existence of routines to generate unique identifiers e.g. `NEW_LABEL` etc.

This example shows that many Snip constructs are too simplistic. However, we recall that Snip was designed as a base language; the difficulty may therefore be overcome by defining suitable extensions to this base.

We illustrate the functioning of this mechanism by considering evaluation of the augment text string:

"WHILE A < B DO

WHILE P < Q DO D := C "

FIGURE 4-14

```

GLOBAL CONST TYPE = 1; ADDR = 2; STATE = 3; /* attribute
                                record indices */
        LOADED = 1; NOT_LOADED = 0; /* attribute states */
VAR   GATTR : VECTOR [ 1..3 ] OF INTEGER; /* global
                                attribute record */

PATTERN COMPOUND_STATEMENT;
    BEGIN
        "BEGIN" . <:-: STATEMENT>
                . { ";" . <:-:STATEMENT> } .
        "END"
    END CST;

PATTERN STATEMENT;
    EXTERNAL VAR GATTR;
    LOCAL    VAR LATTR : VECTOR [ 1..3 ] OF INTEGER;
    BEGIN
        ( "IF" . <:-: EXPR>
            ACT IF ¬inhibit_code THEN gencode
                    FI TCA .
        "THEN" . <:-: STATEMENT> .
        ( "ELSE" ACT IF ¬inhibit_code THEN
            gencode FI TCA .
            <:-: STATEMENT> . "FI"
        | "FI" ))
        | ( <:-: VARIABLE> . ACT LATTR := GATTR /* local
            copy */ TCA .

        " := " .
        <:-: EXPR> ACT
            Check_types
            IF ¬inhibit code THEN
                IF GATTR[STATE] = NOT_LOADED
                    THEN gencode FI
                FI
            TCA
    END ST;

```

PATTERN EXPR;

EXTERNAL VAR GATTR;

LOCAL VAR LATTR: VECTOR [1..3] OF INTEGER;

BEGIN

<=: TERM> .

{ "+" ACT

IF \neg inhibit_code THEN

IF GATTR[STATE] = NOT_LOADED THEN gencode

FI /* code to load */

FI;

LATTR := GATTR /* Local copy */

TCA .

<=: TERM> ACT

Check_types;

IF \neg inhibit_code THEN gencode

FI /* code for addition */

TCA }

END EXPR;

PATTERN TERM;

EXTERNAL VAR GATTR;

LOCAL VAR LATTR: VECTOR [1..3] OF INTEGER;

BEGIN

<=: FACTOR> .

{ "*" ACT

IF \neg inhibit_code THEN

IF GATTR[STATE] = NOT_LOADED THEN

gencode FI /* code to load */

FI;

LATTR := GATTR /* local copy */

TCA .


```

        <=: FACTOR> ACT
        Check_types;
        IF  $\neg$  inhibit_code THEN gencode
            FI /* code for multiplication*/
        TCA }
END TERM;

```

PATTERN FACTOR;

```

BEGIN
    <=: VARIABLE>
    | "(" . <=: EXPR> . ")"
END FACTOR;

```

PATTERN VARIABLE;

EXTERNAL VAR GATTR;

```

BEGIN
    "ID" ACT
        Search_identifier_tables; set_up_attribute_
                                record;
    TCA
END VARIABLE;

```

PRE

DEFINE

PATTERN WHILE_STATEMENT;

LOCAL VAR E, S, L : STRING;

BEGIN

"WHILE" . < E : BOOLEAN_EXPRESSION > . "DO" .
< S : STATEMENT >

ACT

NEW_LABEL (L);

SUBST("DO :" . L .

"IF \neg ".E." THEN LEAVE ".L." FI;"

S

"OD")

TCA

END WHILE_STATEMENT

AS STATEMENT;

ERP

(cf. figure 4-15)

Numbered arcs indicate the order of evaluation. The expression " $A < B$ " is first recognised and stored as the first incarnation of (local variable) E, E/1, say; " $P < Q$ " is stored as the second incarnation of E, E/2, say, while " $D := C$ " is stored as the second incarnation S, S/2. A substitution string is then generated to represent the translation of the inner while-statement: E/2 and S/2 are parameters in this substitution string. The substitution string thus created is stored as the first incarnation of S, S/1. Finally, a substitution string is created for the complete statement, using E/1 and S/1 as parameters. Object code is generated from this final substitution string. In this example, only one buffer is required. However, had the while-statement been defined in a hierarchy of 2 or more levels, then 2 or more buffers would have been required.

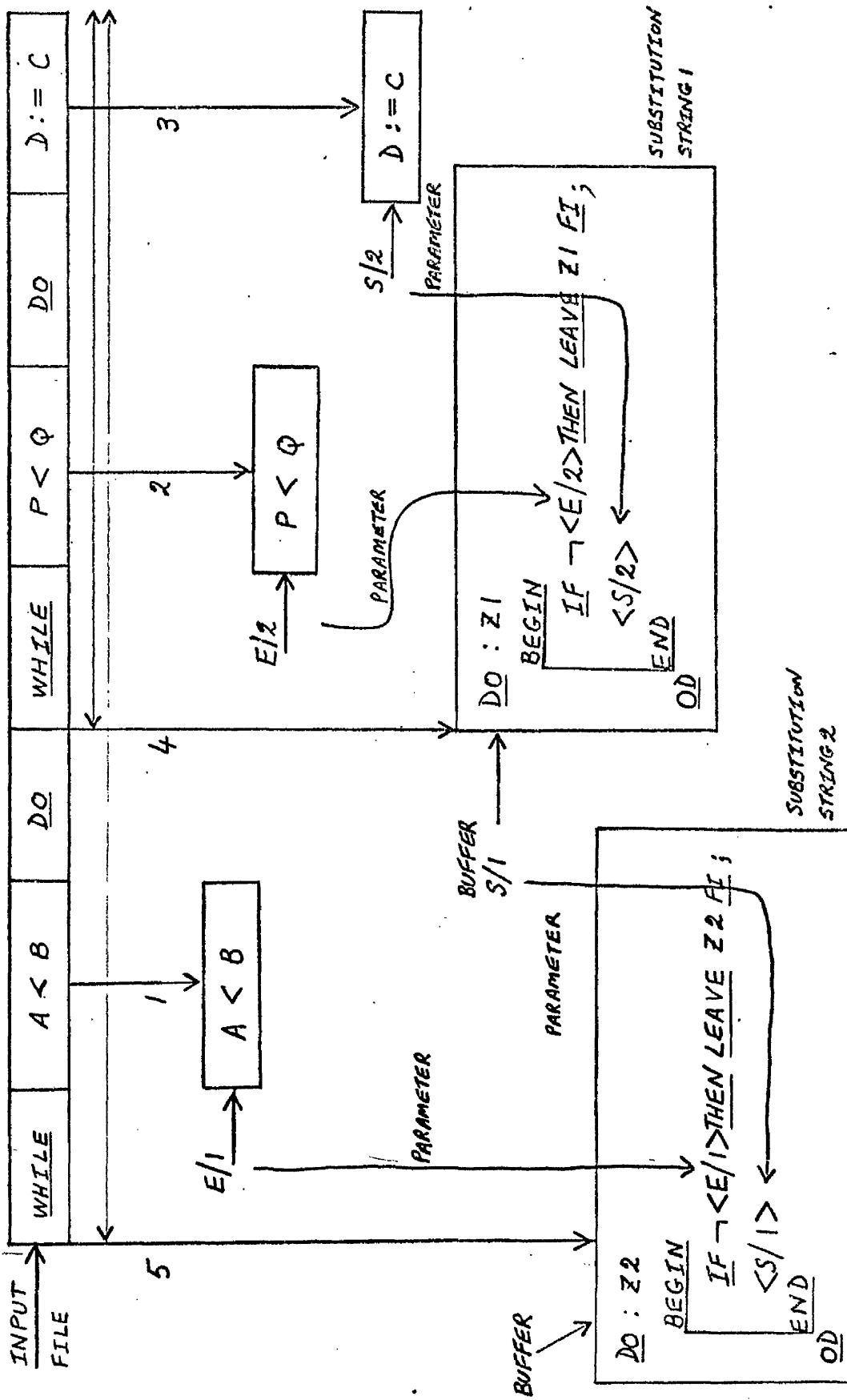


FIGURE 4-15

CHAPTER 5

CONCLUSIONS AND FURTHER RESEARCH

5.0 Introduction

In this chapter, we review the principal developments in this thesis and point out both weaknesses and possible improvements to the system. The underlying belief is that extensible languages ought to afford the user (whether systems programmer/language designer or not) as much protection from himself as possible and that this can be achieved to a large degree by careful choice of language abstractions. The success of this approach must ultimately rely on the willingness of the programmer to accept a "straight-jacket" providing relative security in place of a free language where the responsibility for security is entirely his own.

5.1 Review and Critique

In chapter 1, we described the background problem of language proliferation to which extensible languages provide a partial solution. We considered the evolution of extensible languages and introduced the Solntseff-Yezerski classification scheme to enable orderly discussion of existing extensible languages; we introduced also a diagrammatic representation to illustrate the (likely) translator organisation of each class of system. The theme of this dissertation, namely the security of extensible languages was then briefly discussed.

In chapter 2, we found it expedient to discuss first the notion of security in relation to (simple) programming languages before continuing to discussion of extensible

languages, since the base and extended versions of the base language are in any case (simple) programming languages. We introduced in particular the notions of unstable and overtransparent language features from an intuitive, human-oriented viewpoint. Using a machine-oriented view, we argued that relative security may be achieved (a) by increasing the number of distinct, non-trivial assertions concerning a program and (b) by minimising non-checkable redundancy, essentially suppressing irrelevant detail and defining higher level or aggregate data structures and operations. This second approach reduces the transparency of individual data structures and operations, although it may or may not reduce the transparency of the language as a whole. We argued therefore that improved security relies on the programmers ability and willingness to use appropriate language constructs and that this is more likely when structured programming techniques are used.

We used these notions to design two alternative models of secure extensible languages: the first was found to relate to bootstrapping or to a hierarchy of abstract machines designed for translator portability, and we therefore considered it no further; the second model was found to relate more closely to existing systems. Our discussion of security led us to believe that classification of extensible language systems according to the means of defining the semantics of extensions was more appropriate from the point of view of security. After re-classification of the principal existing extensible languages, we were able to show that none matched the ideals of the model. Perhaps surprisingly, however, we found data structure and operator extensions to have been much more securely developed in

contrast to more general syntax extensions. We therefore proposed a scheme for secure realisation of the model based on a string processing language with patterns which specify structures isomorphic to those described by LL(1) grammars.

Chapter 2 forms the most important part of this thesis: herein lies our claim to originality. The notion of security is not new, and has been introduced into most other contexts on an ad hoc basis. It is however, notably absent in extensible systems. We believe the application of security to extensible languages and to the design of a secure model for extensible languages to be original.

Chapters 3 and 4 are concerned with a possible realisation of the model in terms of a string processing language, Snip. Chapter 3 considered the design of the base language. We presented an informal discussion of the general design principles used and the method used to determine and resolve conflicting aims. This was followed by a brief introduction to Snip itself, and its design. Overtransparent and unstable features of this base language were determined by relating it to the experience of existing programming languages, and the notation was designed by considering brief surveys of the characteristic errors of Algol W, Algol 60 and other languages.

An implementation scheme for Snip was briefly considered in chapter 4. We designed an abstract machine, SAM, for this purpose, and although we did not specifically orient this towards machine independence and portability, we expect it to provide a useful starting point. A physical realisation of SAM for the IBM 370/158 machine as seen through Algol W was discussed. Finally, we briefly

discussed a simple translator architecture for the processing of constructs in the extended language (i.e. the augment text).

The principal goal of this thesis was the design of a secure extensible language; we discuss how far this objective has been achieved: we observed in chapter 2 that security of a programming language is to a large extent unquantifiable, since it is very much influenced by the user, the problem area, and indeed by the particular problem being solved. In the absence of ideal languages, therefore, while we can judge the relative security (overtransparency) of two candidate constructs or groups of constructs, the question of what is overtransparent and/or unstable must remain to some extent subjective - or, more precisely a matter of judgement of those base machine primitives and aggregates which are expected to be heavily used versus those which are not expected to be useful to a particular group of users working in a particular problem area. Thus Dijkstra, Hoare, Wulf, Wirth and Beckman (Dij 68; Hoa 73; Wul 73; Wir 74; Bec 75) have identified various features in many existing languages (goto, pointer, global variables, non-typed data objects, secondary effects) as, in their view, too undisciplined or too flexible. We do not therefore consider it either useful or necessary to produce any absolute measures of security.

In considering the security of Snip as an extensible language, we should start by considering our model. In view of the arguments presented above we do not by any means regard our model as uniquely secure; nor, indeed do we claim that it is finalised for the problem area in which

we expect it to be useful. We claim rather that our model is an initial estimate or starting point for iteration which must be adapted both for each new problem area and also in the light of practical experience which is yet to come. We justify our design decisions partly in terms of our own judgement as to the features likely to be useful and heavily used and partly eliciting parallel situations with (simple) programming languages. We would expect that a better understanding of the causes of human error will allow improved judgement at some later date.

We consider the realisation of the model: as far as the base language is concerned, we recall that we related overtransparency to the current ideas in language technology (e.g. control structures, data structures, data types, block structure etc.) and we justified the notation using simple studies of characteristic errors. We would expect improvements from further and more detailed studies of characteristic errors, and feedback from usage of Snip itself. It is important to remember that the version of Snip described in chapter 3 was intended as a low level base language; we would therefore expect suitable higher level abstractions to be defined in Snip, according to the particular application area. In the long term, then, we expect that both the model and its realisation will evolve considerably (compare for example the evolution of programming languages). We feel, however, that we may justly claim to have produced a useful starting point for further iteration.

As far as the general design of Snip is concerned, we have the following criticisms to make.

New evidence on the use of files has recently been presented in a paper by Wirth (Wir 75). Inefficient buffer handling may result essentially from the attempt to hide from the programmer the fact that files must be allocatable on secondary storage media. From the point of view of security, this is advantageous, provided the consequences on efficiency are fully understood and accepted.

For translatability and simplicity, Pascal was designed with a reasonably small number of operator precedence levels, in contrast to Algol 60; Snip operator precedences were similarly chosen. In retrospect, however, Wirth (Wir 75) considers that the decision to break from the widely traditional precedence seems ill-advised, particularly with the growing significance of complicated boolean expressions in connection with the use of structured programming and program verification. This often leads to the need for additional bracketing of boolean expressions e.g. $X < Y \wedge Y < Z$ vs $(X < Y) \wedge (Y < Z)$. This decision might well be left until the characteristic errors of Snip can be studied.

We consider Snip patterns: the introduction of a construct analogous to the case statement to allow lexically, driven pattern matching for characters might well prove useful for many applications. A means of determining which path of a pattern is traversed during the matching process would also be useful, although this could be introduced as an extension. Several difficulties stem from the fact that Snip was designed principally as an extensible language system, although it was hoped that it would be applicable also to more general string processing.

Some of these difficulties can be resolved as follows.

The restriction that string variables are disallowed as components of patterns is too severe for general string processing since it is useful to allow patterns to depend on input data. The restriction may be relaxed to permit this without re-introducing the side-effects which it was intended to outlaw.

In general string processing applications, it is often useful to be able to determine, within the action statements of a pattern, which string is currently being matched.

Built-in patterns e.g. Snobol 4 are appropriate to general string processing, but not to a secure extensible language system as they encourage violation of the LL(1) structure.

In chapter 2, we noted the subsidiary aim of efficiency for our extensible language. We have purposely avoided paying particular attention to this aim in this paper, but it is perhaps important to point out that "reasonably" efficient implementation is, in our opinion feasible.

Waite (Wai 73) has indicated that a high proportion of run-time in string processing is spent in pattern matching. In view therefore of the considerably reduced run-time flexibility of Snip patterns compared to those of Snobol 4 (cf. arguments of chapter 3) Snip programs should run faster than equivalent Snobol 4 programs.

However, since it is essential to share common substrings to allow efficient handling of substrings recognised during pattern matching, the operations for

updating string values look less attractive than was originally expected.

While SAM was implemented in order to assist in debugging of the abstract machine design, we have done little to consider timings or usage or tuning. The interpretive implementation of SAM will undoubtedly be slow. We might consider, therefore, generation of (real) machine code from the abstract machine code. Snobol 4, for example, achieved a 3-4 fold increase in speed by this method (Gris 72). Alternatively, an intermediate course of part interpretation, part compilation along the lines of Dawson or Mitchell (Daw 73; Mit 70), in an attempt to realise the advantages while avoiding the disadvantages of both systems might prove interesting. This becomes feasible because complex or highly abstracted operations such as pattern matching suffer less from the overheads of interpretation than do simple or low level operations (Gris 72).

With the recent advances in microprogramming, a further alternative implementation is offered. Several proposals have been made for the microprogrammed implementation of Snobol 4 (Gris 72). Recently, Rossman and Jones (Ross 74) have contended that the use of functional memory-based dynamic microprogramming is particularly well-suited to the implementation of string processing languages such as Snobol 4 because of the heavy use of pattern and string data structures and operators, these features being particularly foreign to the usual general purpose hardware.

One might reproach the inefficiency of the extension mechanism itself. Schuman (Sch 71b), for example,

observes that despite the advantages of cascaded (or pyramided) definitions, such extension methods are not without their accompanying drawbacks, which are all too often cited as sufficient reason for abandoning this sort of approach altogether. The most serious arguments are based on the question of efficiency.

The implementation scheme proposed in chapter 4 is a simple-minded scheme, intended for the purposes of illustration only. Several more realistic schemes have been proposed to overcome the accusations of slow compilation and poor object code.

As we observed in chapter 3, Woolley (Woo 71) has devised a system of measuring the effect of depth of definition pyramids on efficiency. We would thus expect to be able to tune any definition structure and place critical or heavily used features low in the definition hierarchy, or perhaps in the base language itself.

A similar but more far-reaching idea is that of the extensible interpreter proposed by Schuman (Sch 71b). In the traditional approach, every layer in the pyramid of definitions is faithfully preserved during translation of the augment text into the base language. Schuman proposes a mixed scheme whereby extensions may be either "interpreted" or compiled to machine code according to some strategy, information for which can be obtained during translation. The "strategy" is intended to identify and thus "flatten" critical sections of the definition structure. Additionally, the operations of the base language may be extended (i.e. semantic extension) according to the expected pattern of use of the language.

We regard realisation of such a scheme for the implementation of Snip as a research topic in its own right.

It would be interesting also to consider the feasibility of efficient implementation of Model M1.

There are several lessons to be derived from the extensible language herein designed.

We have been particularly concerned, as it turns out, with syntactic extensions. We noted that Snip should include also the well-established forms of data structure and operator extension and the more recently discussed control structure extension. Ideally we would like to discard compiler-compiler techniques of introducing semantic extensions because of the difficulties imposed by dependence on translator and real machine architecture. As we have indicated, there has, as yet, been little success in this area.

We were troubled also by the difficulty of handling context-sensitive syntax within a context-free system as this also results in translator-architecture dependency. As we indicated ad hoc solutions to this difficulty are used in most existing systems, but a more satisfactory solution would be to use some simplified form of affix grammar.

We remarked, in chapter 2, that perhaps the principal drawback of using LL(1) grammars is that it is, under certain conditions, easier to specify particular patterns non-deterministically. It would be interesting to consider the feasibility of a system to allow non-deterministic specification of a deterministic grammar rule

cf. non-deterministic Fortran (SPRINT).

5.2 Future Research

As we have already remarked in the preceding section, there are many possible refinements to the basic Snip system; we have indicated several directions in which future research might proceed. There is scope for further isolation of programming language components with the aim of allowing less machine- and translator-dependent semantic extensions. The notion of security might benefit from some attempt at formalisation, from research into human behaviour (the kinds, influences, e.g. complexity and causes of human error) or from more detailed study of characteristic errors. This would provide a more solid background on which to develop further ideas of security.

The notion of overtransparency, in particular, might usefully be applied to the design of any hierarchically structured system.

While we regard informal reports as the most suitable method of presenting and describing both Snip and SAM, formal specification will be necessary at some stage if we are to ensure correct implementation and interpretation.

We have already indicated how Snip may be applied to the extension of several different high level languages, starting from purpose-designed base languages.

There has recently been considerable activity in the field of job control languages (JCL) (Bcs 74). Barron and Jackson (Bar 72) have observed that modern JCL's are akin to programming languages and that we are likely to get better job control languages if we develop them as such, and use the same criteria of judgement. Barron (Bar 74)

has also remarked that the facilities provided by job control languages are often more general than is required. This might then prove an interesting area in which to use (secure) extensible languages: we might argue that security is of particular importance to job control languages because of the possible consequences in terms of machine resources, integrity of files and program development time. Snip might similarly be applied to the areas of query languages or information retrieval and database management languages.

The string processor itself might prove useful in other areas of operating systems which involve processing of text, such as context editing of files. It might also be usefully applied in language-to-language translation (perhaps even JCL-to-JCL translation cf. Dakin (Dak 72)). We would expect that the (convenient) capability of textual insertion offered by a string processor would be of value when the target language is a high level language as opposed to a low level abstract or real machine language.

We have presented in this dissertation a number of ideas, some borrowed, some new, concerning the design of secure extensible language systems. In our opinion, the most significant was the line of thought leading to the design of a secure model for extensible languages. We consider that Snip shows promise of helping to provide insight into the development of more secure such systems.

Whatever the relative merits of our particular system, it seems inevitable that with the ever-widening application of, and dependence upon computers, and the more devastating the consequences of failure, there will be increasing pressure for greater software reliability and

thus also for the greater use of some criterion of security in the design of programming and extensible programming languages.

A.0 We view the Report on the Pascal Language (Wir 70) as one of the most succinct and lucid informal descriptions of a programming language. In this appendix we are aiming at a descriptive documentation of the Snip language and not a precise and formal document oriented towards correct implementation. For this reason, we adhere closely to the form of the Pascal report.

A.1 Summary of the Language

A summary of the language appears in section 3.2.1.

A.2 Notation, Terminology and Vocabulary

We define the syntax of Snip using the Backus-Naur form. We find it convenient to use the meta-brackets { and } to indicate that the enclosed construct is to be repeated zero or more times.

The basic vocabulary consists of basic symbols classified into letters, digits and special symbols. In those cases where the special symbols coincide with meta-linguistic symbols, we underline the special symbol. (e.g. { is a meta-linguistic symbol, while { is a special symbol in Snip).

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
P | Q | R | S | T | U | V | W | X | Y | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special symbol> ::= <restricted special symbol> | " |

<restricted special symbol> ::= + | - | * | / | \ | ^ | ~ | = | ≠ | < | > | ≤ |
≥ | (|) | [|] | { | } | := | . | , | : | ↑ |
@ | _ | & | / | * | % | ; | # | | | ① | ② | € |
NULL | IF | THEN | ELSE | FI |

CASE | OF | ESAC | DO | OD | LEAVE | VAR | CONST | DIV |
FUNCTION | PROCEDURE | SIZE | ACT | TCA | PATTERN |
APPEND | FALSE | TRUE | INTEGER | BOOLEAN |
STRING | FILE | VECTOR | DEFINE | AS | TAKE |
WHERE | EOL | EXTERNAL | READ | WRITE | LOCAL |
GLOBAL | PRE | ERP

The construct `/*` <any sequence of basic symbols followed by `EOL`> `*/` may be inserted between any two identifiers, numbers (cf. section A.3) or restricted special symbols. This construct is called a comment and may be removed from the program text without altering its meaning. Program text which follows a comment of this form must appear on a new line of text. A single identifier immediately following one of the basic symbols `END`, `OD`, `FI` or `ESAC` is also regarded as a comment.

<any sequence of basic symbols followed by `EOL`>
 ::= <restricted special symbol> | " | <letter>
 | <digit>

A.3 Identifiers, Numbers and String Literals

Identifiers serve to denote constants, variables, statement labels, functions and patterns. Their association must be unique within their scope of validity i.e. within the procedure, function or pattern in which they are declared (cf. sections A.9, 10, 11).

`<identifier>` ::= `<letter>` { `<letter or digit or connector>` }
`<letter or digit or connector>` := `<letter>` | `<digit>` | _

Numbers are constants of the data type integer.

`<number>` ::= `<integer>`
`<integer>` ::= `<digit>` { `<digit>` }

String literals are constants of the data type string. A

string literal is an ordered sequence of basic symbols enclosed in quotes, or the symbol NULL.

$$\begin{aligned} \langle \text{string literal} \rangle &::= \langle \text{quote} \rangle \langle \text{string item} \rangle \\ &\quad \{ \langle \text{string item} \rangle \} \langle \text{quote} \rangle \mid \underline{\text{NULL}} \\ \langle \text{quote} \rangle &::= " \\ \langle \text{string item} \rangle &::= \langle \text{basic symbol other than} \rangle \mid "" \\ \langle \text{basic symbol other than} \rangle &::= \langle \text{restricted special symbol} \rangle \mid \\ &\quad \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \end{aligned}$$

A.4 Constant Definitions

A constant definition introduces an identifier as a synonym to a constant.

$$\begin{aligned} \langle \text{boolean constant} \rangle &::= \underline{\text{TRUE}} \mid \underline{\text{FALSE}} \\ \langle \text{unsigned constant} \rangle &::= \langle \text{number} \rangle \mid \langle \text{string literal} \rangle \mid \\ &\quad \langle \text{boolean constant} \rangle \\ \langle \text{constant} \rangle &::= \langle \text{unsigned constant} \rangle \mid \langle \text{sign} \rangle \langle \text{number} \rangle \\ \langle \text{constant definition} \rangle &::= \langle \text{identifier} \rangle = \langle \text{constant} \rangle \\ \langle \text{sign} \rangle &::= + \mid - \end{aligned}$$

The standard integer constant identifier WORD_LENGTH can be assumed to be predeclared. Its value is the number of string characters which may be packed into one word of the implementation machine.

A.5 Data Types

A data type determines the set of values which variables of that type may assume, and associates an identifier with the type. In the case of structured types it also defines their structuring method.

$$\langle \text{type} \rangle ::= \langle \text{scalar type} \rangle \mid \langle \text{structured type} \rangle$$

A.5.1 Scalar Types

A scalar type defines an ordered set of values

$\langle \text{scalar type} \rangle ::= \underline{\text{INTEGER}} \mid \underline{\text{BOOLEAN}}$

INTEGER . The values are the integers within a range depending on the particular implementation. The values are denoted by integers (cf. section A.3).

BOOLEAN The values are boolean values denoted by TRUE and FALSE.

A.5.2 Structured Types

Structured types are defined by describing the types of their components and by indicating a structuring method.

$\langle \text{structured type} \rangle ::= \langle \text{vector type} \rangle \mid \langle \text{file type} \rangle \mid \langle \text{string type} \rangle$

A.5.2.1 Vector Types

A vector type is a structure consisting of a fixed number of components which are all of the same type, called the component type. The elements of the vector are selected by indices of type integer. The vector type definition specifies the component type.

$\langle \text{vector type} \rangle ::= \underline{\text{VECTOR}} \left[\langle \text{lower bound} \rangle .. \langle \text{upper bound} \rangle \right] \underline{\text{OF}} \langle \text{component type} \rangle$

$\langle \text{lower bound} \rangle ::= \langle \text{integer constant} \rangle$

$\langle \text{upper bound} \rangle ::= \langle \text{integer constant} \rangle$

$\langle \text{integer constant} \rangle ::= \langle \text{constant} \rangle$

$\langle \text{component type} \rangle ::= \langle \text{scalar type} \rangle \mid \langle \text{string type} \rangle$

Example: VECTOR $\left[\emptyset .. 1\emptyset \right] \underline{\text{OF}} \underline{\text{INTEGER}}$

A.5.2.2 String Types

A string type is a structure consisting of a linear list of components which are all of the same type. The

components are in fact implicitly defined to be basic symbols (cf. section A.2). The values associated with the string type are thus the set of possible sequences of basic symbols. These values are denoted by string literals.

The number of components, called the length of the string, associated with the string type is not normally fixed by the type definition (i.e. each variable of that type may have a value with a different, varying length). It is possible, however, to specify an upper bound to this length. The elements, or consecutive sequences of elements of the string are designated by a pair of selectors (of type integer) which specify the position of the selected element sequence, and its length (cf. section A.6).

Associated with each variable of string type is a string position or string cursor denoting a specific element. The string cursor may be moved during a match operation (cf. section A.7.1.4) or by explicit manipulation (cf. section A.8.1.1).

```
<string type> ::= STRING <maximum string length>
<maximum string length> ::= <empty> | (<integer>)
<empty> ::=
```

Examples: STRING
 STRING (80)

A.5.2.3 File Types

A file type definition specifies a structure consisting of a sequence of components, all of the same type. The number of components, called the length of the file, is not fixed by the file type definition (i.e. each variable of that type may have a value with a different,

varying length).

Associated with each variable of file type is a file position or file pointer denoting a specific element. The file position or file pointer can be moved by certain standard procedures (cf. section A.9.1).

$$\langle \text{file type} \rangle ::= \underline{\text{FILE}} \left[\langle \text{mode} \rangle \right] \underline{\text{OF}} \langle \text{component type} \rangle$$
$$\langle \text{mode} \rangle ::= \underline{\text{READ}} \langle \text{mix mode} \rangle \mid \underline{\text{WRITE}}$$
$$\langle \text{mix mode} \rangle ::= \mid \underline{\text{WRITE}} \mid \langle \text{empty} \rangle$$
$$\langle \text{component type} \rangle ::= \langle \text{scalar type} \rangle \mid \langle \text{fixed maximum length string type} \rangle$$
$$\langle \text{fixed maximum length string type} \rangle ::= \langle \text{string type} \rangle$$

A.6 Declarations and Denotations of Variables

Variable declarations consist of a list of identifiers denoting the new variables followed by declaration of their types.

$$\begin{aligned} \langle \text{variable declaration} \rangle &::= \langle \text{identifier} \rangle \{ , \langle \text{identifier} \rangle \} \\ &\quad : \langle \text{type} \rangle \end{aligned}$$

Two standard file variables can be assumed to be predeclared as

$$\text{INPUT} : \underline{\text{FILE}} \left[\underline{\text{READ}} \right] \underline{\text{OF}} \underline{\text{STRING}} (81)$$
$$\text{OUTPUT} : \underline{\text{FILE}} \left[\underline{\text{WRITE}} \right] \underline{\text{OF}} \underline{\text{STRING}} (121)$$

The INPUT file is restricted to input mode (read only) and the OUTPUT file to output mode (write only). A Snip program should be regarded as a procedure with these two variables as formal parameters: the corresponding actual parameters are expected either to be the standard input and output media of the computer installation or to be specifiable in the system command activating the Snip system.

Examples:

I,J : INTEGER
B : BOOLEAN
V1,V2 : VECTOR [-10 .. +10] OF INTEGER
S1,S2 : STRING
F : FILE [READ | WRITE] OF STRING (81)

Denotations of variables either denote an entire variable or a component of a variable.

$\langle \text{variable} \rangle ::= \langle \text{entire variable} \rangle \mid \langle \text{component variable} \rangle$

A.6.1 Entire Variables

An entire variable is denoted by its identifier.

$\langle \text{entire variable} \rangle ::= \langle \text{variable identifier} \rangle$

$\langle \text{variable identifier} \rangle ::= \langle \text{identifier} \rangle$

A.6.2 Component Variables

A component variable is denoted by the denotation for the variable followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable.

$\langle \text{component variable} \rangle ::= \langle \text{indexed variable} \rangle \mid \langle \text{substring} \rangle \mid$
 $\langle \text{current file component} \rangle \mid \langle \text{string cursor} \rangle$

A.6.2.1 Indexed Variables

A component of a vector variable is denoted by the denotation for the variable followed by an index expression.

$\langle \text{indexed variable} \rangle ::= \langle \text{vector variable} \rangle \left[\langle \text{index expression} \rangle \right]$

$\langle \text{index expression} \rangle ::= \langle \text{integer expression} \rangle$

$\langle \text{integer expression} \rangle ::= \langle \text{expression} \rangle$

$\langle \text{vector variable} \rangle ::= \langle \text{variable} \rangle$

Examples:

A [12]
A [I + J]

A.6.2.2 Substrings

The selector for a string variable permits access to a contiguous sequence of elements (called a substring) of the string. A substring is denoted by the denotation for the variable followed by a substring selector which specifies the first element and the length of the substring. The components of a string are numbered consecutively from zero upwards. Thus, a string of length l has l components addressed \emptyset to l-1 (inclusive).

$\langle \text{substring} \rangle ::= \langle \text{string variable} \rangle \left\{ \begin{array}{l} \langle \text{first component} \\ \text{position} \rangle \mid \langle \text{substring length} \rangle \end{array} \right\}$
 $\langle \text{substring length} \rangle ::= \langle \text{non negative integer expression} \rangle$
 $\langle \text{first component position} \rangle ::= \langle \text{non negative integer} \\ \text{expression} \rangle$
 $\langle \text{non negative integer expression} \rangle ::= \langle \text{integer expression} \rangle$
 $\langle \text{string variable} \rangle ::= \langle \text{variable} \rangle$

A.6.2.3 String Cursors

Every string variable declared has a cursor variable of type integer associated with it (cf. section A.5.2.2). The cursor indicates a particular component of the string. The value of a cursor lies in the integer range \emptyset to l (inclusive) where l is the (current) length of the string with which the cursor is associated. A cursor variable is denoted by the denotation of the string variable with which it is associated, followed by the symbol "@".

$\langle \text{string cursor} \rangle ::= \langle \text{string variable} \rangle @$

Cursor variables are initially "undefined".

Examples:

S {0 | 2}

$$\begin{array}{l}
S \{ 10 | 5 \} \\
S \{ S@ | I+J \} \\
S \left[\begin{array}{c} I \\ I \end{array} \right] \{ 2 | 3 \} \\
S \left[\begin{array}{c} I \\ I \end{array} \right] @
\end{array}$$

A.6.2.4 Current File Components

At any one time, only the one component determined by the current file position (or file pointer) is directly accessible.

$$\begin{array}{l}
\langle \text{current file component} \rangle ::= \langle \text{file variable} \rangle \uparrow \\
\langle \text{file variable} \rangle ::= \langle \text{variable} \rangle
\end{array}$$

Examples:

$$\begin{array}{l}
F \uparrow \\
F \uparrow @ \\
F \uparrow \{ 2 | 3 \}
\end{array}$$

A.7 Expressions

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands i.e. variables and constants, operators, patterns and functions.

The rules of composition specify operator precedences according to four classes of operators. The operators \neg and SIZE have the highest precedence, followed by the multiplying operators, then the adding operators, and finally, with lowest precedence, the relational operators. Sequences of operators of the same precedence are executed from left to right. These rules of precedence are reflected by the syntax:-

$$\begin{array}{l}
\langle \text{factor} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{unsigned constant} \rangle \mid \\
\quad \langle \text{function designator} \rangle \mid (\langle \text{expression} \rangle) \mid \\
\quad \neg \langle \text{factor} \rangle \mid \underline{\text{SIZE}} \langle \text{factor} \rangle
\end{array}$$

$$\begin{aligned}
 \langle \text{term} \rangle &::= \langle \text{factor} \rangle \{ \langle \text{multop} \rangle \langle \text{factor} \rangle \} \\
 \langle \text{simple expression} \rangle &::= \langle \text{term} \rangle \{ \langle \text{addop} \rangle \langle \text{term} \rangle \} \mid \langle \text{addop} \rangle \\
 &\quad \langle \text{term} \rangle \{ \langle \text{addop} \rangle \langle \text{term} \rangle \} \\
 \langle \text{expression} \rangle &::= \langle \text{simple expression} \rangle \mid \langle \text{simple expression} \rangle \\
 &\quad \langle \text{relop} \rangle \langle \text{simple expression} \rangle
 \end{aligned}$$

Examples:

Factors:	X	15	<u>SIZE</u> S
	(X+Y+Z)	\neg B	EOS (F)
Terms:	X * Y	P \wedge Q	S.S1.S2
	I <u>DIV</u> (I-1)	B \wedge (X < Y)	

Simple Expressions:

X + Y	P \vee Q
-X	I * J + 1

Expressions:

X = 1.5	S \in P
A < B	

A.7.1 Operators

A.7.1.1 Operators \neg and SIZE

\neg The operator \neg applied to a boolean operand denotes negation.

SIZE The operator SIZE applied to a string operand denotes the length of the operand.

A.7.1.2 Multiplying Operators

$\langle \text{multop} \rangle ::= \text{DIV} \mid * \mid \wedge \mid .$

Operator	operation	type of operands	type of result
*	multiplication	integer	integer
<u>DIV</u>	division with truncation	integer	integer
\wedge	logical "and"	boolean	boolean
.	concatenation	string	string

Concatenation: The length of the string result is equal to the sum of the lengths of the two string operands.

A.7.1.3 Adding Operators

$\langle \text{addop} \rangle ::= + \mid - \mid \vee$

Operator	operation	type of operands	type of result
+	addition	integer	integer
-	subtraction	integer	integer
\vee	logical "or"	boolean	boolean

A.7.1.4 Relational Operators

$\langle \text{relop} \rangle ::= = \mid \neq \mid < \mid \leq \mid \geq \mid > \mid \epsilon$

Operator	operation	type of operands	type of result
= \neq	relation	integer or string	boolean
< \leq	relation	integer	boolean
\geq >	relation	integer	boolean
ϵ	pattern match	string (first operand) pattern (second operand)	boolean

Match Operation

A pattern describes a particular subset of the set of possible string values by defining an ordered sequence of successor substrings and substring alternatives from which the subset of values may be composed (cf. section A.11). A pattern thus essentially describes a tree-like structure in which successor substrings are represented as "sons" and substring alternatives as "brothers". The match operation takes two operands, one of string type, the other a pattern. The effect of this operation is to determine whether or not the value of the string (or one of its substrings) belongs to the subset of string values defined by the pattern. The match operation effectively interprets the pattern by systematically traversing the tree structure (defined by the pattern) comparing string and substring, until either the string is matched or there are no further alternative branches to attempt. This process may be compared to top-down syntactic analysis.

We consider the expression " $B \in A$ ", where B is a string, called the subject string, and A is a pattern. The match operation matches string B (commencing from the character position indicated by B@ i.e. B's current cursor position) to the pattern A, to determine whether a substring of B belongs to the set of string values described by A.

The matching process proceeds as follows:

- (1) The match pointer is set to point to one of the initial alternative substrings specified (in the ~~ordered sequence of substring successors~~) of A.
- (2) B is matched (from its current cursor position) to the substring pointed to by the match pointer, proceeding

from left to right.

- (3) If the match in (2) succeeds, the match pointer is updated to point to one of the subsequent successor substrings (if any). The subject string cursor points one symbol beyond the matched substring. The process continues at (2).
- (4) If the match in (2) fails, the match pointer is updated to point to one of the remaining untried alternative substrings (if any). The process continues at (2).
- (5) The process (as above) continues until either
 - (a) there are no successors; in this case, the result of the match operation is TRUE, and the subject string cursor points to the end of the matched substring, or
 - (b) there are no untried alternatives; in this case, the result of the match operation is false, and the subject string cursor position remains unchanged.

Since the structure of a Snip pattern is isomorphic to the LL(1) grammar structure (cf. section A.11), the match/recognition process is unambiguous, and the result is independent of the order in which substring alternates are attempted.

If the subject string is a file buffer, matching continues across successive file components, if necessary.

A.7.2 Function Designators

The precise interpretation of a function designator is as in Algol 60, Algol W or Pascal. A function designator specifies the activation of a function. It consists of the identifier designating the function and a

list of actual parameters. The parameters are variables or expressions (cf. sections A.9, 10).

$\langle \text{function designator} \rangle ::= \langle \text{function identifier} \rangle$
 $(\langle \text{actual parameter} \rangle \{, \langle \text{actual parameter} \rangle \})$
 $\langle \text{function identifier} \rangle ::= \langle \text{identifier} \rangle$
 $\langle \text{actual parameter} \rangle ::= \langle \text{expression} \rangle$

Examples: EOF(S)
 SUM(A + B)

A.8 Statements

Statements denote algorithmic actions and are said to be executable.

$\langle \text{statement} \rangle ::= \langle \text{simple statement} \rangle \mid \langle \text{structured statement} \rangle$

A.8.1 Simple Statements

A simple statement is a statement, no part of which constitutes another statement.

$\langle \text{simple statement} \rangle ::= \langle \text{assignment statement} \rangle \mid$
 $\qquad \qquad \qquad \langle \text{procedure statement} \rangle \mid$
 $\qquad \qquad \qquad \langle \text{insertion statement} \rangle \mid$
 $\qquad \qquad \qquad \langle \text{escape statement} \rangle \mid$
 $\qquad \qquad \qquad \langle \text{empty statement} \rangle \mid \langle \text{append statement} \rangle$

A.8.1.1 Assignment Statements

The assignment statement serves to replace the current value of a variable by a new value indicated by an expression. The assignment operator symbol is ":", pronounced "becomes".

$\langle \text{assignment statement} \rangle ::= \langle \text{assignment variable} \rangle :=$
 $\qquad \qquad \qquad \langle \text{expression} \rangle$

$\langle \text{assignment variable} \rangle ::= \langle \text{string variable} \rangle \mid \langle \text{integer variable} \rangle$
 $\qquad \qquad \qquad \mid \langle \text{boolean variable} \rangle \mid \langle \text{function identifier} \rangle$

The variable (or the function) and the expression must be of

identical type.

Examples: $X := Y + 2$
 $S := S1 \cdot S2 \{ 2 | 4 \}$
 $F := S \{ \emptyset | 8 \}$

Substring Assignments

When the assignment variable is an entire string variable, the length of the variable after assignment is determined by the length of the assigned expression.

However, when the assignment variable is a substring, the assignment corresponds to the "replacement" of the substring. In this case, therefore, the length of the substring after assignment must be unchanged i.e. the length of the substring must be greater than or equal to the length of the string expression (if greater, then the string expression is first extended to the right with blanks until the lengths are equal).

Examples:

$S \{ 2 | 4 \} := \text{"ABCD"}$
 $S \{ 1 | I \} := S2 \{ 3 | I \}$

The assignment $S \{ \emptyset | 4 \} := S2 \{ \emptyset | 8 \}$ is illegal.

Avoidance of Ambiguity

The possibility of semantic ambiguity arises in substring assignments in which the same string variable S (say) appears on both sides of an assignment statement.

Example:

Consider the sequence of statements:

$S := \text{"ABCD"}$;
 $S \{ 2 | 2 \} := S \{ 1 | 2 \}$

If the assignment is implemented by copying substring

$S \{1|2\}$ character by character, the resulting value of S is "ABBB"; while if the complete substring $S \{1|2\}$ is copied as a whole, the resulting value of S is "ABBC". This second interpretation is the one intended.

We avoid the ambiguity by defining

$$"S \{s_1|l_1\} := S \{s_2|l_2\}"$$

to be semantically equivalent to the text

$$"SW := S \{s_2|l_2\} ; S \{s_1|l_1\} := SW"$$

where

s_1, s_2 are character positions, $\emptyset \leq s_1, s_2 \leq \text{SIZE } S - 1$

l_1, l_2 are substring lengths, $\emptyset \leq s_1 + l_1, s_2 + l_2 \leq \text{SIZE } S$

SW is a string variable.

A.8.1.2 Append Statements

The append statement is a special form of assignment statement which is used to append the value of a string expression to the existing value of a string variable. The string variable is an entire variable (and not a substring).

$\langle \text{append statement} \rangle ::= \langle \text{string variable} \rangle \text{ APPEND}$

$\langle \text{string expression} \rangle$

$\langle \text{string expression} \rangle ::= \langle \text{expression} \rangle$

Examples:

$S \text{ APPEND } S1$

$S \text{ APPEND } S1 . S2 \{ \emptyset | 3 \}$

Semantically, these statements are equivalent to the statements

$S := S . S1$

$S := S . S1 . S2 \{ \emptyset | 3 \}$, respectively.

The append statement permits optimisation.

A.8.1.3 Insertion Statements

The insertion statement is a special form of assignment statement which serves to insert the value of a string expression at the current cursor position of the insertion variable. The string insertion variable is an entire string variable (and not a substring).

$\langle \text{insertion statement} \rangle ::= \langle \text{string variable} \rangle \text{ ① } \langle \text{string expression} \rangle$

Examples: S ① "ABC" S ① S1 { 0 | 2 }

Semantically, these statements are equivalent to the statements

$S := S \{ 0 | S@ \} . \text{"ABC"} . S \{ S@ | \underline{\text{SIZE}} S - S@ \}$
 $S := S \{ 0 | S@ \} . S1 \{ 0 | 2 \} . S \{ S@ | \underline{\text{SIZE}} S - S@ \}$
respectively.

The length of the string insertion variable after insertion of the expression is determined by its original length, plus the length of the inserted expression. The insertion statement permits optimisation.

A.8.1.4 Escape Statements

An escape statement permits control to leave its current environment. Further processing continues after the structured statement (cf. section A.8.2) whose label is specified in the escape statement. The scope of a label is the structured statement which it labels. It is not therefore possible to escape to the end of a structured statement which is not currently being executed.

$\langle \text{escape statement} \rangle ::= \underline{\text{LEAVE}} \langle \text{structured statement label} \rangle$
 $\langle \text{structured statement label} \rangle ::= \langle \text{identifier} \rangle$

Examples: Execution of the statement "LEAVE Z2" in the text:-

```

BEGIN
:
  BEGIN : Z2
  :
    BEGIN : Z3
    :
      LEAVE Z2
      :
    END ;
    :
  END ;
  :
  A := B;
  :
END

```

causes transfer of control to the statement "A := B".

A.8.1.5 Procedure Statements

A procedure statement serves to execute the procedure denoted by the procedure identifier. The procedure statement may contain a list of actual parameters which replace their corresponding formal parameters defined in the procedure declaration (cf. section A.9). The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. There are two kinds of parameters: variable-parameters whose values may be altered by the procedure body, and constant-parameters whose values are constant within the procedure body.

In the case of variable-parameters, the actual parameter must be a variable. If it is a variable denoting a component of a structured variable, the selector is evaluated when the substitution takes place (i.e. before the execution of the procedure). If the parameter is a constant parameter, then the corresponding actual parameter must be an expression.

Two further restraints are imposed:

- A-19
- (a) All actual parameters whose values may be altered by the procedure (i.e. the actual parameters corresponding to variable-parameters) must be distinct from each other and from the non-local variables defined in the external references declaration part of the called procedure (cf. section A.9).
- (b) None of the above-noted (variable-) actual parameters nor the specified non-local variables may be contained in any of the expressions corresponding to the (constant-) actual parameters.

$\langle \text{procedure statement} \rangle ::= \langle \text{procedure identifier} \rangle ($
 $\qquad \qquad \qquad \langle \text{procedure statement tail} \rangle$
 $\langle \text{procedure statement tail} \rangle ::= \langle \text{actual parameter} \rangle$
 $\qquad \qquad \qquad \{, \langle \text{actual parameter} \rangle \}) |)$
 $\langle \text{procedure identifier} \rangle ::= \langle \text{identifier} \rangle$
 $\langle \text{actual parameter} \rangle ::= \langle \text{expression} \rangle \mid \langle \text{variable} \rangle$

Examples: NEXT ()
 PUT (F1)
 TRANS (S1.S2)

A.8.1.6 Empty Statements

The empty statement consists of no symbols and denotes no actions.

$\langle \text{empty statement} \rangle ::= \langle \text{empty} \rangle$

A.8.2 Structured Statements

Structured statements are constructs composed of other statements which have to be executed in sequence (compound statement) or conditionally (conditional statements) or repeatedly (loop statement). Any loop- or compound-statement may have associated with it a label which may be referenced by an escape-statement (enclosed

by the loop- or compound-statement) cf. section A.8.1.4.

$$\langle \text{structured statement} \rangle ::= \langle \text{compound statement} \rangle \mid$$

$$\langle \text{conditional statement} \rangle \mid \langle \text{loop statement} \rangle$$

A.8.2.1 Compound Statements

The compound statement specifies that its components are to be executed in the same sequence as that in which they are written.

$$\langle \text{compound statement} \rangle ::= \text{BEGIN} \langle \text{label clause} \rangle$$

$$\langle \text{compound tail} \rangle$$

$$\langle \text{compound tail} \rangle ::= \langle \text{statement} \rangle \{ ; \langle \text{statement} \rangle \} \text{END}$$

$$\langle \text{label clause} \rangle ::= : \langle \text{label} \rangle \mid \langle \text{empty} \rangle$$

$$\langle \text{label} \rangle ::= \langle \text{identifier} \rangle$$

Example: BEGIN Z := X; X := Y; Y := Z END

A.8.2.2 Conditional Statements

A conditional statement selects for execution, a single one of its component statements.

$$\langle \text{conditional statement} \rangle ::= \langle \text{if statement} \rangle \mid \langle \text{case statement} \rangle$$

A.8.2.2.1 If-Statements

The if-statement specifies that a statement is to be executed only if a certain boolean expression is true. If it is false, then either no statement is to be executed, or the statement following the symbol ELSE is to be executed.

$$\langle \text{if statement} \rangle ::= \text{IF} \langle \text{boolean expression} \rangle \text{ THEN} \langle \text{statement} \rangle$$

$$\langle \text{else clause} \rangle$$

$$\langle \text{else clause} \rangle ::= \text{ELSE} \langle \text{statement} \rangle \text{ FI} \mid \text{FI}$$

$$\langle \text{boolean expression} \rangle ::= \langle \text{expression} \rangle$$

Examples:

IF I < 5 THEN V := V + 1 ELSE V := ∅ FI

IF S \in P THEN COUNT := \emptyset FI

A.8.2.2.2 Case Statements

A case statement consists of an expression (the selector) and a list of statements, each labelled by a constant of type integer. Only the statement whose label is equal to the current selector value is executed.

\langle case statement $\rangle ::=$ CASE \langle integer expression \rangle OF

\langle case body \rangle ESAC

\langle case body $\rangle ::=$ \langle case list element \rangle { ; \langle case list element \rangle }

\langle case list element $\rangle ::=$ \langle case label \rangle { | \langle case label \rangle } :
 \langle statement \rangle

\langle case label $\rangle ::=$ \langle integer \rangle

Examples:

```
CASE      I      OF
  1: X:= SIZE STR;
  2: X:=Y;
  3 | 4: X:=Z
ESAC
```

A.8.2.3 Loop Statements

The loop statement specifies that certain statements are to be executed repeatedly. In order that loop execution is of finite duration, at least one of the component statements of a loop statement must be an escape statement.

\langle loop statement $\rangle ::=$ DO \langle label clause \rangle \langle loop tail \rangle

\langle loop tail $\rangle ::=$ \langle statement \rangle { ; \langle statement \rangle } OD

Example:

```
DO : LP1
  GET(F);
  IF (F  $\uparrow$  = S)  $\vee$  EOF(F) THEN LEAVE LP1 FI
OD
```

A.9 Procedure Declarations

Procedure declarations serve to define parts of programs and to associate identifiers with them so that they can be activated by procedure statements.

```
<procedure declaration> ::= <procedure heading>
                               <external references declaration part>
                               <local declaration part>
                               <statement part>
```

The procedure heading specifies the identifier naming the procedure and the formal parameters (if any). The parameters are either constant- or variable-parameters (cf. section A.8.1.5).

```
<procedure heading> ::= PROCEDURE <identifier> <rest heading>
<rest heading> ::= ( <formal parameter section>
                     { ; <formal parameter section> } ) ; |
<formal parameter section> ::= CONST <parameter group> |
                               VAR <parameter group> | <empty>
<parameter group> ::= <identifier> { , <identifier> } : <type>
```

The effect of CONST is that of call by value, and the effect of VAR that of indirect addressing. The effect of indirect addressing is explained by the following rule which is applied to the procedure body before the procedure is invoked:

If the formal parameter section contains the symbol VAR, the selectors (if any) of actual parameters are first evaluated, formal parameters are then replaced throughout the procedure body by the corresponding actual parameter. Possible conflicts between the identifier inserted through this process and another local identifier already present within the procedure body will be avoided

A-23

by suitable systematic changes of the local identifier involved.

The external references declaration part must contain declarations of all non-locals (i.e. main program or global variables) referenced within the statement part. As noted in section A.8.1.5, non-local variables accessed in this way must be distinct from the actual parameters used in the call of the procedure.

`<external references declaration part> ::=`
`EXTERNAL <non local constant definition part>`
`<non local variable declaration part>`
`| <empty>`

The non-local constant definition part contains all constant synonym identifiers defined external to the procedure.

`<non-local constant definition part> ::=`
`CONST <global identifier>`
`{, <global identifier>}; | <empty>`

`<global identifier> ::= <identifier>`

The non-local variable declaration part contains all variable identifiers declared external to the procedure.

`<non-local variable declaration part> ::=`
`VAR <global identifier>`
`{, <global identifier>}; | <empty>`

`<local declaration part> ::= LOCAL <constant definition part>`
`<variable declaration part> |`
`<empty>`

The constant definition part contains all constant synonym identifiers local to the procedure.

`<constant definition part> ::= CONST <constant definition>;`
`{<constant definition>;} | <empty>`

The variable declaration part contains all variable

declarations local to the procedure.

$\langle \text{variable declaration part} \rangle ::= \text{VAR } \langle \text{variable declaration} \rangle ;$
 $\{ \langle \text{variable declaration} \rangle ; \} \mid \langle \text{empty} \rangle$

The statement part specifies the algorithmic actions to be executed upon an activation of the procedure by a procedure statement.

$\langle \text{statement part} \rangle ::= \langle \text{compound statement} \rangle$

All identifiers introduced in the formal parameter part, the constant definition part or in the variable declaration part are local to the procedure declaration which is called the scope of these identifiers. They are not known outside their scope. In the case of local variables, their values are undefined at the beginning of the statement part.

The use of a procedure identifier in a procedure statement within its declaration implies recursive execution of the procedure.

Examples of procedure declarations:

PROCEDURE FREQ (CONST I : INTEGER);

EXTERNAL VAR TOTAL, CHART;

BEGIN

TOTAL := TOTAL + 1;

CHART [I] := CHART [I] + 1

END FREQ

PROCEDURE STRSWOP (VAR S1, S2 : STRING);

LOCAL VAR W : STRING;

BEGIN

W := S1 ; S1 := S2 ; S2 := W

END STRSWOP

A.9.1 Standard Procedures

Standard procedures are predeclared in Snip. The standard procedures are listed and explained below:

PUT(F) advances the file pointer of file F to the next file component. It is only applicable if the file mode is WRITE or READ WRITE.

GET(F) advances the file pointer of file F to the next file component. It is only applicable if the file mode is READ or READ WRITE. If there does exist a "next file component", the end-of-file condition arises, and the value of $F \uparrow$ becomes undefined.

The effect of GET(F) is defined if EOF(F) (cf. section A.10.1) is false prior to its execution.

RESET(F) the file pointer of file F is reset to its beginning.

SETEND(F) the file pointer of file F is set to point to the end of the file.

A.10 Function Declarations

Function declarations serve to define parts of the program which compute a scalar value or a string value.

Functions are activated by a function designator (cf. section A.7.2) which is a constituent of an expression.

A function declaration consists of the following parts:

$\langle \text{function declaration} \rangle := \langle \text{function heading} \rangle$

$\quad \langle \text{restricted external references declaration part} \rangle$

$\quad \langle \text{local declaration part} \rangle \langle \text{statement part} \rangle$

The function heading specifies the identifier naming the function, the formal parameters of the function and the type of the (result of the) function.

<function heading> ::= FUNCTION <identifier>

(<function formal parameter section>
 { ; <function formal parameter section> }) :
 <result type>

<function formal parameter section> ::= CONST

<parameter group> { ; <parameter group> }

<result type> ::= <scalar type> | <string type>

<restricted external references declaration part> ::=

EXTERNAL CONST <global identifier>

{ , <global identifier> } ; | <empty>

The type of the function must be a scalar or string type.

Within the function declaration, there must be at least one assignment statement assigning a value to the function identifier. This assignment determines the result of the function. Occurrence of the function identifier in a function designator within its declaration implies recursive execution of the function. Within the statement part, no assignment is allowed to any variable which is not local to the function. This rule also excludes assignments to parameters. All parameters are therefore constant parameters, and all non-locals constant.

Examples:

FUNCTION MAX(CONST SV : VECTOR [1..10] OF INTEGER);
LOCAL VAR MS, I : INTEGER;

BEGIN

MS := SV [1] ; I := 2;

DO : LOOP

IF MS < SV [I] THEN MS := SV [I] FI;

I := I + 1;

IF I > 10 THEN LEAVE LOOP FI

OD;

MAX := MS

END MAX

FUNCTION TOP_TAIL (CONST S : STRING)

BEGIN

TOP_TAIL := S { \emptyset | 1 } . S { SIZE S - 1 | 1 }

END TOP_TAIL

A.10.1 Standard Functions

Standard functions are predeclared Snip functions.

The standard functions are listed and explained below:

EOF(X) If X is of type file, EOF indicates whether the file is in end-of-file status.

If X is of type string, EOF indicates whether the current cursor position is the end of string i.e. $X@ = \text{SIZE } X$.

INT(S) S must be of type STRING(1). The result, of type integer, is the ordinal number of the character $S \{ \emptyset | 1 \}$ in the defined character set.

CHR(I) I must be of type integer. The result, of type STRING(1), is the character whose ordinal number is I in the defined character set.

A.11 Pattern Declarations

A pattern declaration is a special purpose procedure which serves to define a particular subset of the set of possible string values, and to associate an identifier with this subset. This set of string values is defined by a pattern template: a pattern template is an ordered set of ~~successor~~ and alternative string components from which the defined set of string values may be composed.

~~A given, subject~~ string (so-called) may be matched to a pattern to determine whether the subject string itself (or one of its substrings) belongs to the set of string

values defined by the pattern template. We say that the match operator (cf. section A.7.1) interprets the pattern template.

Local variables, to which matched substrings may be assigned, can be associated with a pattern declaration. These local variables are also local to groups of statements (called action primitives) which may be defined within the pattern template.

The form of pattern template (excluding the action primitives) is restricted in such a way that it is isomorphic to the structure of an LL(1) grammar. The problem of ambiguity does not therefore arise (cf. appendix C).

A pattern declaration consists of the following parts:

$\langle \text{pattern declaration} \rangle ::= \langle \text{pattern heading} \rangle \langle \text{pattern body} \rangle$

$\langle \text{pattern body} \rangle ::= \langle \text{external references declaration part} \rangle$

$\langle \text{local declaration part} \rangle$

BEGIN $\langle \text{pattern template} \rangle$ END

$\langle \text{pattern heading} \rangle ::= \text{PATTERN } \langle \text{pattern identifier} \rangle ;$

$\langle \text{pattern identifier} \rangle ::= \langle \text{identifier} \rangle$

The pattern template specifies the primitive strings and primitive string combinations defining the pattern.

$\langle \text{pattern template} \rangle ::= \langle \text{simple pattern string} \rangle$

$\{ \langle \text{alternate operator} \rangle \langle \text{simple pattern string} \rangle$

$\langle \text{simple pattern string} \rangle ::= \langle \text{primitive pattern string} \rangle$

$\{ \langle \text{concatenation operator} \rangle$

$\langle \text{primitive pattern string} \rangle \}$

$\langle \text{primitive pattern string} \rangle ::= \langle \text{free primitive pattern string} \rangle$

$\langle \text{action primitive} \rangle$

$$\langle \text{free primitive pattern string} \rangle ::= \langle \text{string constant} \rangle \mid$$

$$\langle \text{sub-pattern} \rangle \mid (\langle \text{pattern template} \rangle)$$

$$\mid \{ \langle \text{pattern template} \rangle \}$$

$$\langle \text{subpattern} \rangle ::= \langle \text{match assignment variable} \rangle :$$

$$\langle \text{pattern identifier} \rangle$$

$$\langle \text{match assignment variable} \rangle ::= \langle \text{string variable} \rangle \mid -$$

An action primitive specifies statements to be executed during pattern interpretation.

$$\langle \text{action primitive} \rangle ::= \underline{\text{ACT}} \langle \text{statement} \rangle \{ ; \langle \text{statement} \rangle \}$$

$$\underline{\text{TCA}} \mid \langle \text{empty} \rangle$$

$$\langle \text{alternate operator} \rangle ::= \mid$$

$$\langle \text{concatenation operator} \rangle ::= .$$

In the pattern template,

- (1) the concatenation operator indicates that the two adjacent strings are to be taken as successor strings during the match operation,
- (2) the alternate operator indicates that the two adjacent strings are to be taken as alternative strings during the match operation,
- (3) the concatenation operator takes precedence over the alternate operator; the pair brackets "(" and ")" are used to override this precedence,
- (4) the pair brackets "{ " and " } " indicate that the enclosed template is to be repeated zero or more times,
- (5) the pair brackets ACT and TCA indicate that the enclosed statement(s) are to be executed during the match operation on matching of the $\langle \text{free primitive pattern string} \rangle$ which they succeed,
- (6) the pair brackets "<" and ">" indicate that the enclosed pattern identifier is a component pattern; the substring matched to this pattern component is assigned

to the corresponding match assignment variable (if any).
 The use of a pattern identifier in the pattern template within its declaration implies the recursive execution of the pattern (during a pattern match).

Examples:

PATTERN P;

LOCAL VAR I : INTEGER;

BEGIN

 ("MON" | "TUES" | "WED" | "THURS" | "FRID") ACT I := 1 TCA
 | ("SAT" | "SUN") ACT I := 2 TCA

END P

PATTERN E;

LOCAL VAR T1, T2 : STRING;

BEGIN

 <T1 : T> ACT TEXT (T1) TCA .
 { "+" . <T2 : T> ACT TEXT (T2) TCA

END E

A.12 Incremental Sections

The incremental section allows the incremental declaration of variables, procedures, functions and patterns associated with the definition of extensions to the language. This section consists of two parts:

- (a) the program declaration part consisting of the definition of variables, functions, procedures and patterns used in defining extensions. (The scope of these declarations is the incremental section itself.)

and

- (b) the extension sections which permit the connection of newly defined patterns (or pattern bodies) as alternates to existing patterns, thus extending existing patterns

in a controlled manner.

The compiled incremental section is executed before the execution of the main program.

```

<incremental section> ::= PRE <pre program declaration part>
                        { <extension section> ; } ERP | <empty>
<pre program declaration part> ::= <local declaration part>
                                <procedure or function or pattern
                                  declaration part>
<procedure or function or pattern declaration part> ::=
    { <procedure or function or pattern declaration>; }
<procedure or function or pattern declaration> ::=
    <procedure declaration> |
    <function declaration> | <pattern declaration>
<extension section> ::= <define statement> | <take statement>

```

Both the define- and the take-statement allow the definition of extensions by (restricted) modification of existing patterns. The define-statement allows the specification of a new alternate to an existing pattern template, while the take-statement allows the specification of a new alternate to some component part of an existing pattern template. The take-statement is thus the more flexible.

A.12.1 Define-Statements

The define-statement permits extensions by allowing the construction of a (new) pattern declaration as alternate to an existing pattern, called the context specifier.

```

<define statement> ::= DEFINE <pattern declaration> AS
                        <context specifier>

```

```

<context specifier> ::= <pattern identifier>

```

The pattern must conform to an LL(1) structure both before and after execution of the define-statement.

A.12.2 Take Statements

The take-statement provides for the definition of extensions by allowing the construction of a (new) pattern as the alternate of some component part of a pattern which already exists.

```

<take statement> ::= TAKE <modified pattern template>
                     WHERE <extension declaration> AS .
                     <context specifier>

```

$$\langle \text{extension declaration} \rangle ::= \langle \text{pattern declaration} \rangle$$
$$\langle \text{modified pattern template} \rangle ::= \langle \text{pattern template} \rangle$$

The `<extension declaration>` is the declaration of a pattern defining the extension. The `<modified pattern template>` denotes the original pattern template modified to include the newly defined extension as the alternate of some `<simple pattern string>` of the original pattern. The `<context specifier>` denotes the original pattern, and hence, specifies the context of the extension.

Restrictions:

- (a) As with the define-statement, the modified pattern must conform to an LL(1) structure.
- (b) The precedence structure of the original part of the template must remain unchanged; so also must the action part (indeed, the action part of the original template may therefore be omitted from the take-statement).

The language syntax is defined as a standard pattern set (with empty replaced by the symbol NULL) so that extensions may be readily transported from one installation to another.

Examples:

We assume that a compiler for Snip has been written

in terms of itself; and the compiler implemented by bootstrapping or hand translation. We assume also that the compiler defines a pattern named after and corresponding to each syntactic class defined in the report i.e. the language is defined as a standard pattern set. We introduce the following extensions:

(a) A repeat statement of the form

REPEAT <statement> UNTIL <boolean expression>

DEFINE PATTERN REPEAT_ST;

LOCAL VAR BEXPR, BL, S : STRING;

BEGIN

"REPEAT" . <S : STATEMENT> . "UNTIL".

<BEXPR : BEXPRESSION>

ACT

NEW_LABEL (BL);

SUBST ("BEGIN" . ":" . BL .

"DO" .

S . ";" .

"IF" . BEXPR . "THEN" . "LEAVE" . BL . "FI" .

"OD" .

"END")

TCA

END AS STATEMENT;

In this example, we assume that procedures NEW_LABEL and SUBST are defined in the compiler: NEW_LABEL generates a new and unique identifier; SUBST generates the substitution string constructed as the semantic equivalent of some section of extended text.

(b) We introduce an initial value construct in which variables may be assigned values prior to execution of the statement part of a main program. This construct will take the form:

$$\underline{\text{VALUE}} \langle \text{variable} \rangle = \langle \text{constant} \rangle \{ ; \langle \text{variable} \rangle = \langle \text{constant} \rangle \}$$

We propose to allow this construct to appear immediately before the statement part. We define the meaning of this construct by substituting appropriate assignment statements at the beginning of the statement part. In order to handle this extension, we must modify the pattern template corresponding to the production:

$$\langle \text{program} \rangle ::= \langle \text{incremental section} \rangle \langle \text{program declaration part} \rangle$$

$$\langle \text{statement part} \rangle .$$

to the rule:

```

<program> ::= <incremental section> <program declaration part>
              <initial value part> <statement part> .
              | <incremental section> <program declaration part>
              <statement part> .

```

We denote this as follows:

```

TAKE INCREMENTAL_SECTION. PROGRAM_DECLARATION_PART.
      (INITIAL_VALUE_PART | NULL). STATEMENT_PART. ". "

```

WHERE

```

PATTERN INITIAL_VALUE_PART;
LOCAL VAR V, C, VALUES : STRING;
BEGIN
    "VALUE" . <V:VARIABLE> . "=" . <C:CONSTANT>
        ACT VALUES := V . " :=" . C . ";" TCA
    . { ";" . <V:VARIABLE> . "=" . <C:CONSTANT>
        ACT VALUES APPEND V . " :=" . C . ";" TCA }
    . "BEGIN"
        ACT SUBST ("BEGIN" . VALUES) TCA
END

```

AS PROGRAM:

A.13 Programs

A program has the form of a procedure declaration without the heading. There are, however, no non-local variables, and hence no declaration of external references.

Procedures, functions and patterns may, however be declared.

```

<program> ::= <incremental section> <program declaration part>
                <statement part> .
<program declaration part> ::= GLOBAL
                <constant definition part>
                <variable declaration part>
                <procedure or function or pattern
                    declaration part>
                | <empty>

```

A.14 Program Examples

(a) The following program recognises the simple arithmetic expressions defined by the grammar

```

<expression> ::= <term> { + <term> }
<term> ::= <factor> { * <factor> }
<factor> ::= id | ( <expression> )

```

The recognised expression is printed in Reverse Polish form.

GLOBAL

PATTERN EXPR;

BEGIN

```

    <-: TERM> . { "+" . <-: TERM>
                ACT OUTPUT↑ APPEND "+" TCA }

```

END EXPR;

PATTERN TERM;

BEGIN

```

    <-: FACT> . { "*" . <-: FACT>
                ACT OUTPUT↑ APPEND "*" TCA }

```

END TERM;

PATTERN FACT;

BEGIN

 "ID" ACT OUTPUT↑ APPEND "ID" TCA
 | "(" . <-: EXPR> . ")"

END FACT;

BEGIN

 OUTPUT↑ := NULL;

IF ¬ (INPUT ∈ EXPR) THEN

 OUTPUT↑ := "INVALID EXPRESSION"

FI;

 PUT(OUTPUT)

END MAIN.

- (b) Example (b) is included in appendix F since it employs many of the language extensions defined therein.

APPENDIX B SNIP ABSTRACT MACHINE (SAM)

In this appendix, we describe in detail, the various SAM order codes and some brief examples of usage. We assume familiarity with the SAM architecture cf. section 4.3.

B.1 SAM Order Codes

SAM order codes are of the form

$$m \quad [p_1] \quad [p_2] \quad [p_3]$$

where m is the instruction mnemonic, p₁, p₂ and p₃ are parameters and the brackets "[]" indicate options.

All SAM orders include table pointers which "chase" the pointers to determine the location from which a variable is to be fetched, or to which it is to be stored. The pointer may, for example, point to a vector entry rather than a simple variable entry; in this case, the order expects the offset from the element with zero subscript to be loaded in ACC. This is an example of the pointer being "overloaded". We elaborate this notion of pointer overloading:

SAM is designed so that, as far as possible, a single instruction can carry out a particular operation, irrespective of the type of the operand(s). For particular operand types, some special action or special access information may be required. If the need for some such action or information is determined by chasing the table pointer, then we say that the pointer is overloaded. The permitted forms of overloading are covered along with the orders themselves.

Many orders which include table pointers permit a zero in place of this pointer; in this case, the operand is obtained from ACC or the top cell of the TRS, rather than being fetched as part of the elaboration of the order.

B.1.1 The Monadic Load Group (ML)

The ML or "Monadic Load" group is associated with monadic operators. Instructions in this group have the format:

m = p2

where m = instruction mnemonic

p2 = table pointer or 0

The instructions are:

ML MLP

ML \neg MLSLA

MLEFS MLSZE

All SAM orders are written using the convention that an underlined variable name or constant denotation indicates a pointer to the table entry for that variable or constant:

ML

If X is a variable of type integer or boolean, the effect of ML X is to load the contents of variable X into ACC.

If X is of type boolean, \neg X compiles as ML \neg X which effects ML X, followed by \neg to the contents of ACC.

The pointer field may be overloaded:

- (a) by an (integer) vector entry pointer, in which case the offset from the address of the vector element with zero subscript is in ACC,

- (b) by a string or file buffer cursor; in this case, the cursor is determined by accessing (in turn) the table entry and the string descriptor, or
- (c) by an entry pointer to the cursor of (an element of) a string vector; in this case the index is in ACC.

MLP

If X is a variable of type integer or boolean, then MLP X fetches to ACC a parameter value from the parameter space of an activation record which is no longer current. This record is pointed to by LP cf. section B.1.9.

MLSLA

The effect of MLSLA, which takes no parameters, is to load the contents of ACC into the string length accumulator (SLA). The contents of ACC is not destroyed.

MLSZE

The effect of MLSZE X is to load the size of a string (or file buffer) into ACC.

The pointer field may be overloaded by a vector of string; in this case, the index is loaded in ACC.

MLEFS

The effect of MLEFS is to load ACC with the boolean result of the operation END-OF-FILE (or -STRING) X.

The pointer field may be overloaded by a vector entry pointer, in which case the index is loaded in ACC.

B.1.2 The Subscripting Group (SC)

The subscripting group is associated with subscript calculations. A subscript instruction follows the placing

of an integer subscript value in ACC. Instructions in this group have the format:

$$m \quad p_2 \quad \left[p_3 \right] \quad \text{where} \quad p_2 = \text{table pointer}$$

$$p_3 = \text{table pointer or zero}$$

The instructions are:

SUB

SUBZ

SUBZ

If X is a vector, then SUBZ X checks that the integer Z (say) in ACC lies in the range of the subscript of X. This range is determined by accessing the record pointed to by X.

SUB

SUB X, Y \equiv MLY , SUBZ X

Examples:

The following declarations are assumed to have been made in this and later examples:

A, B, C, I, J, L, M, N : INTEGER ;

S, S1, S2, E, F, G : STRING

VS: VECTOR [1..10] OF STRING;

V, VINT : VECTOR [0 .. 99] OF INTEGER

FL: FILE [READ | WRITE] OF STRING(80);

(A) The integer expression "VINT [I] " compiles as

n + 0 SUB VINT, I

n + 1 ML VINT

B.1.3 The Load and Operate Group (LO)

The Load and Operate group combine the application of

n + 1 SUB V, M

n + 2 L+ V

where $n + i$, $0 \leq i \leq 2$, is the code address.

(C) The integer expression "VINT [I * J]" compiles as

n + 0 ML I

n + 1 LMULT J

n + 2 SUBZ VINT

n + 3 ML VINT

Relational Operators

The instructions $L \leq$ $L =$ $L \geq$ $L <$ $L \neq$ $L >$
effect integer comparisons.

Examples:

(D) The expression " $A \leq B$ OR $B \leq C$ " compiles as

n + 0 ML A

n + 1 $L \leq$ B

n + 2 ML B

n + 3 $L \leq$ C

n + 4 LOR 0

(E) The expression " $A \leq B$ AND $B \leq C$ " compiles as

n + 0 ML A

n + 1 $L \leq$ B

n + 2 ML B

n + 3 $L \leq$ C

n + 4 LAND 0

LSLA+

The effect of LSLA+ (which takes no operands) is to add the contents of ACC to SLA, leaving the contents of ACC unaltered.

LDSA op1, X

The effect of this operation is to load register DSA with the address and length of the string X. This instruction is used when string X appears on the left-hand side of a (sub) string assignment or insertion.

The particular actions involved in loading DSA depend on the operation type indicated by "op1".

The pointer field may be overloaded as follows:

- (a) by a string entry pointer; in this case, the length is in ACC; if the string offset is non-zero, it is loaded in the top cell of the TRS, or
- (b) by a (string) vector entry pointer; in this case, the string offset and length are loaded, as in the previous case, and the index is, additionally, loaded in the top cell of the TRS.

Examples: cf. G, 0 below

LMATCH X

The effect of LMATCH on a pattern X is as follows:

- (1) The address of the pattern template is fetched to register SSA,
- (2) The match instruction interprets the pattern template:-
 - (a) the first node of the template specifies execution of the appropriate code section to lay out an activation record and initialise local variable space,
 - (b) subsequent template nodes may specify:-
 - (i) the matching of a substring to the subject string described by register DSA,

- (ii) the call of a subpattern (to be interpreted in the same manner), and laying out of a new activation record, storing the contents of SSA as the return address in the link space (cf. figure 4-5 (b)),
 - (iii) the execution of a section of code corresponding to an action primitive; in this case the existing activation record remains current, but the contents of registers SSA, DSA and PC are stored in areas SSAS, DSAS and MRA (respectively) of the link space (cf. figure 4-5(b)).
- (c) On completion of interpretation of a pattern template, the topmost activation record is popped and interpretation resumes at the node address (if any) specified in the return address of the link space.
- (3) When there are no further return links the result of the match operation is left in ACC. If successful, the matched string is assigned to the match-assignment variable (if any), and the current cursor position of the subject string (described by register DSA) is updated, appropriately.

If the subject string is a file buffer (indicated by "op1"), a new file component is fetched, as required, during the matching process, and both the file buffer cursor and the current component pointer are updated on a successful match.

Examples:

- (G) The match expression " $S \in PAT$ " where PAT is some

pattern variable compiles as follows:

n + 0 LDSA PATTERN, S

B.1.4 Store Group (ST)

```

m  p1  p2      where  p1  =  c or 0
                        p2  =  table pointer
                        c   =  DSA/VSNUL/FSNUL

```

ST . STDESCRP

The normal store operations are ST, STP and the string store operations are STDESCR, STDESCRP.

The store operation determines the address to be assigned to by chasing the pointer X, and storing the contents of ACC in that address. After execution, ACC and the TRS should be empty.

ST completes the address calculation before making the assignment.

(H) The assignment "I := M" compiles as

n + 0 ML M

n + 1 ST I

(I) The assignment "VINT [I*J] := N" compiles as

n + 0 ML I

n + 1 L* J

n + 2 SUBZ VINT

n + 3 ML N

n + 4 ST VINT

(J) The assignment "VINT [J] @ := VINT [K] @" compiles as

n + 0 SUB VINT, J

n + 1 SUB VINT, K

n + 2 ML VINT@

n + 3 ST VINT@

STP X

This operation bears the same relation to ST as MLP to ML. If X is a pointer to an integer or boolean entry, then STP X stores the parameter value in ACC in the parameter space X of an activation record which is about to become current. This record is pointed to by LP. (cf. section B.1.9).

STDESCR c, X

This instruction stores a string value for string X, by updating the descriptor for X. The exact manner in which this occurs is dependent on the value of c:

(a) c = DSA

In this case, the heap area which previously held the (string) value of X is garbaged, and the descriptor of

the new string value (in register DSA) is copied to the descriptor for X.

The pointer field may be overloaded by a vector entry pointer; in this case, the index is found in the SOL, relative to SB.

(b) c = VSNULL

String X is deleted. The heap area previously occupied by the (string) value of X is garbaged, and the descriptor set undefined. The pointer may be overloaded as in case (a).

(c) c = FSNULL

The (string) file buffer X is deleted. The action is the same as in case (b) except that the string area is not garbaged, and overloading by a vector pointer cannot occur.

The operation is invalid if X is a pointer to a table entry for a constant variable or formal parameter (cf. section A.9).

Example: The string assignment "S := NULL" compiles
as n + 0 STDESCR VSNULL, S

STDESCRP c', X (where c' = DSA only)

This instruction bears the same relation to STDESCR DSA, X as STP to ST.

The string descriptor of the parameter value is copied from DSA to the parameter space X of the activation record which is about to become current. This record is pointed to by LP.

$$m \quad p_1 \quad p_2 \quad \text{where} \quad p_1 = op2$$

p_2 = table pointer

The instructions are: INITVS INITVSP

(1) ACC is stacked into SOL.

(2) The size of the new string value is calculated from SLA. (The precise calculations vary, according to op1).

(3) The SOL pointer register, CSD, is set to point to the SOL base.

(4) The required amount of space is reserved on the heap; register DSA is set to describe this area.

The pointer field may be overloaded as follows:

(a) by a (string) vector entry pointer; in this case the index is loaded in the bottom cell of the SOL, or

(b) by a file buffer entry pointer; in this case step (4), above, is omitted.

INITVSP op1', X (where op1' = STRASSIGN only)

INITVSP bears the same relation to INITVS STRASSIGN, X
as STDESCRP to STDESCR.

Initialisation is effected for the parameter space X in the activation record which is about to become current. This record is point to by LP (cf. section B.1.9).

B.1.6 Load-and-Store String Group (LSS)

This group is used to carry out the combined operations of "loading and storing" a (sub)string i.e. a copy action.

Instructions in this group have the format:

m p₁ p₂ where p₁ = s / op₃
p₂ = table pointer
op₃ = STRINSERT/SSTRREPLACE/SSTRDELETE
s = 0/LOADED

The instructions are: LSTSTR LSTMTL
 LSTHD LSTETL

LSTSTR s, X

The effect of this instruction is defined as follows:

- (1) Register SSA is loaded with the address and length of (sub)string called the source (sub)string. If s = LOADED, then the character offset is found in the SOL (pointed to by CSD), otherwise the offset is zero. The string length is found above the offset (if loaded), in the SOL.
- (2) If a substring is specified, the instruction checks that its offset and length lie within the source string bounds.
- (3) The source (sub)string specified by SSA is copied to the destination area on the heap described by register

DSA.

(4) Registers DSA, SSA and CSD are appropriately updated.

The pointer field may be overloaded by a (string) vector entry pointer; in this case the index is also found in the SOL below the offset and length (accessed via CSD).

Examples:

(K) The string concatenation "S := S1 . S2 . E" compiles as

```
n + 0    MLSZE  S1
n + 1    MLSLA
n + 2    MLSZE  S2
n + 3    LSLA+
n + 4    MLSZE  E
n + 5    LSLA+
n + 6    INITVS STRASSIGN, S
n + 7    LSTSTR 0, S1
n + 8    LSTSTR 0, S2
n + 9    LSTSTR 0, E
n + 10   STDESCR DSA, S
```

(L) The substring assignment "E := F { F@|6 }" compiles as

```
n + 0    ML F@
n + 1    ML 6
n + 2    MLSLA
n + 3    INITVS STRASSIGN, E
n + 4    LSTSTR LOADED, F
n + 5    STDESCR DSA, E
```

(M) The append statement "FL \uparrow APPEND S" compiles as

```
n + 0  MSLZE FL  $\uparrow$ 
n + 1  MLSLA
n + 2  MLSZE S
n + 3  LSLA+
n + 4  INITVS STRAPPEND, FL  $\uparrow$ 
N + 5  LSTSTR 0, S
```

LSTHD op3, X

For the purposes of the next three instructions, we consider that the selection of a substring divides the original string into three parts: the head, the middle and the tail.

Consider string S and the substring S { a | b }.

We define:

head (S) = S { 0 | a }

middle (S) = S { a | b }

tail (S) = S { a + b | size (S) - (a + b) }

We consider that a string cursor similarly subdivides a string S:

head (S) = S { 0 | S@ }

middle (S) = S { S@ | 1 }

tail (S) = S { S@ + 1 | size (S) - (S@ + 1) }

The effect of LSTHD op 3, X is similar to that of LSTSTR. It is used when only the head (X) is to be copied. The parameter op3 indicates whether the string is subdivided by the cursor or by a substring. In the substring case, the offset and length are loaded in the SOL and accessed via CSD.

The pointer field may be overloaded by a string

vector entry pointer; in this case the index is loaded in ACC or the SOL.

LSTMTL op3, X

This instruction behaves in a manner similar to LSTHD, but copies the middle and tail of string X. It is used with op3 = STRINSERT only i.e. X is subdivided by the cursor X@.

LSTETL op3, X

This instruction behaves in a manner similar to LSTHD, but copies only the tail of string S. It is used with op3 = STRREPLACE/STRDELETE only i.e. X is subdivided by some substring.

Examples:

(N) The substring replacement "E { E@|J } := F { F@|6 } .G"

compiles as

```
n + 0  ML E@
n + 1  ML J
n + 2  ML F@
n + 3  ML 6
n + 4  MLSLA
n + 5  MLSZE G
n + 6  LSLA+
n + 7  INITVS SSTRASSIGN, E
n + 8  LSTHD SSTRREPLACE, E
n + 9  LSTSTR LOADED, F
n + 10 LSTSTR 0 , G
n + 11 LSTETL SSTRREPLACE, E
n + 12 STDESCR DSA, E
```

(O) The file buffer substring replacement

"FL \uparrow {10|16} := E{0|7} . S"

compiles as

```
n + 0  ML 10
n + 1  ML 16
n + 2  ML 0
n + 3  ML 7
n + 4  MLSLA
n + 5  MLSZE S
n + 6  LSLA+
n + 7  INITVS SSTRREPLACE, FL $\uparrow$ 
n + 8  LDSA SSTRREPLACE, FL $\uparrow$ 
n + 9  LSTSTR LOADED, E
n + 10 LSTSTR 0, S
```

(P) The substring deletion "S {S@|I} := NULL" compiles

as

```
n + 0  ML S@
n + 1  ML I
n + 2  INITVS *FSTRDELETE, S
n + 3  LSTETL SSTRDELETE, S
```

*The dynamic-maximum-length (DML) string S is treated as a fixed-maximum-length (FML) string, in this case, to avoid unnecessary copying.

(Q) The insertion statement "S \textcircled{i} S1 {N|M} . S2" compiles

as

```
n + 0  ML N
n + 1  ML M
n + 2  MLSLA
n + 3  MLSZE S2
```

n + 4 LSLA+
 n + 5 INITVS STRINSERT, S
 n + 6 LSTHD STRINSERT, S
 n + 7 LSTSTR LOADED, S1
 n + 8 LSTSTR 0, S2
 n + 9 LSTMTL STRINSERT, S
 n + 10 STDESCR DSA, S

B.1.7 Jump Group (J)

The Jump group is responsible for all simple control transfers.

Instructions in this group have the format:

m p₁ p₂ where p₁ = 0 or k
 p₂ = ca or table pointer
 k = number of case alternatives
 ca = code address

The instructions are IFJ UJ
 CASEJ GOTO

UJ ca

UJ transfers control unconditionally to the code address ca.

IFJ ca

IFJ transfers control to the code address ca if the value of ACC is false.

Example:

(R) The loop statement

"DO

I := 1

OD"

compiles as

$n + 0$ ML 1
 $n + 1$ ST I
 $n + 2$ UJ $n + 0$

CASEJ k, ca

This instruction

- (a) checks that the value in ACC is less than or equal to the number of case alternatives k, and
- (b) transfers control to the location $ca + (ACC)$ in RXS (where "()" means "contents of").

Following the code for the case statement are compiled, in successive locations, the addresses of each case alternative. The code address parameter ca points to the location preceding the first of these.

Example:

- (S) The case statement

" CASE N OF

3: I := 0 ;

1: J := 1 ;

2: K := 1

ESAC " compiles as

$n + 0$ ML N

$n + 1$ CASEJ 3, $n + 10$

$n + 2$ ML 0

$n + 3$ ST I

$n + 4$ UJ $n + 14$

$n + 5$ ML 1

$n + 6$ ST J

$n + 7$ UJ $n + 14$

$n + 8$ ML 1


```

n + 9   ST K
n + 10  UJ n + 14
n + 11  UJ n + 5
n + 12  UJ n + 8
n + 13  UJ n + 2
n + 14

```

GOTO X

This instruction is used to handle exits from structured statements. X is a pointer to a table entry for a structured statement. GOTO transfers control to the address (the end of the structured statement) obtained from this table entry.

Example:

(T) The compound statement

```

" BEGIN : BL1
    I := J ;
    IF I < 2 THEN LEAVE BL1 FI ;
    M := N

```

END " compiles as

```

n + 0   ML J
n + 1   ST I
n + 2   ML I
n + 3   L< 2
n + 4   IFJ n + 6
n + 5   GOTO BL1      where BL1 is a pointer to the
n + 6   ML N           table entry for the compound
n + 7   ST M           statement labelled BL1.
n + 8

```

B.1.8 The Block and Pattern Control Group (BPC)

The Block and Pattern Control group effects the initialisation and termination of (use of) an activation record (for a pattern procedure or the main program block). Instructions in this group have the format:

m p₂ where p₂ = table pointer or 0

The instructions are: IB EXIT
 IV RETURN

IB, IV are involved in laying out the activation record for the new block; EXIT and RETURN collapse the activation record.

IB X

The action taken by this order depends on the overloading of X:-

- (a) If X points to the table entry for the main program block, then IB sets up the link and local variable space for the block. Registers AP, SB, ST, HB, HT are initialised.
- (b) If X points to a table entry for a procedure, IB sets up link, parameter and local variable space for the activation record of the procedure about to be entered. DP is extracted from AP. AP is not updated, but LP is. SB is stored in SBS.
- (c) If X points to a table entry for a pattern, IB sets up link and local variable space for the activation record of the pattern template about to be interpreted. DP is extracted from AP. SB is stored in SBS. AP is updated.

IV X

IV initialises the descriptor of (a structured) variable X in the local space of the activation record about to become current.

EXIT

Exit terminates the main program.

Example:

(U) The program skeleton "I,J: INTEGER: S1, S2: STRING

BEGIN

I := 3

END " compiles as

n + 0	IB <u>MP</u>	where <u>MP</u> points to the table
n + 1	IV <u>S1</u>	entry for main program.
n + 2	IV <u>S2</u>	
n + 3	ML <u>3</u>	
n + 4	ST <u>I</u>	
n + 5	EXIT	

RETURN X

The action taken by this order depends on the overloading of the pointer X:-

- (a) If X points to a table entry for a procedure, RETURN unstacks the activation record stack, resetting AP from DP. Control is transferred to the address stored in the RA field of the link data pointed to by LP.
- (b) If X points to a table entry for a pattern which contains action primitives, then RETURN transfers control to the address stored in the MRA field of the link data pointed to by LP. In this case, AP is not updated and the activation record stack is not popped.

Example:

(V) The pattern declaration

```
"PATTERN PAT;  
    LOCAL VAR I,J : INTEGER; S : STRING ;  
    BEGIN  
        ("K".("AR" | "EE")."P" ACT I := I+1 TCA)  
        | "CAT" ACT J := J + 1 TCA  
    END PAT;"
```

compiles as

```
n + 0  IB PAT  
n + 1  IV S  
n + 2  RETURN PAT  
n + 3  ML I  
n + 4  L+ 1  
n + 5  ST I  
n + 6  RETURN PAT  
n + 7  ML J  
n + 8  L+ 1  
n + 9  ST J  
n + 10 RETURN PAT
```

B.1.9 The Procedure Control Group (PC)

The Procedure Control group effects the initialisation and termination of procedures. Instructions in this group have the format:

m p₂ where p₂ = table pointer or 0

The instructions are: CP TP

The orders used to control the normal initialisation of the activation records for the entry to and subsequent treatment of function results (if any) are IB, CP and TP. The

call sequence for a procedure is:-

IB X

<code to set up var or const parameters>

CP X

<link to function result>

TP

The procedure body starts with codes for setting up the local variable descriptions in the activation record.

The procedure body terminates with RETURN.

CP X

CP calls the procedure body X and stores its return address in the RA field of the link data pointed to by LP. AP is updated.

TP X

TP loads LP with the contents DP from the link data pointed to by LP, and SB from the SBS of the record subsequently pointed to by LP.

Examples:

(W) The procedure declaration

" PROCEDURE ELEM (CONST X : STRING) ;

EXTERNAL CONST K ;

LOCAL VAR Z : STRING ;

BEGIN

Z := X

V [K] := V [K] + 1

END " compiles as

n + 0 IV Z

n + 1 MLSZE X

n + 2 MLSLA

```

n + 3  INITVS STRASSIGN, Z
n + 4  LSTSTR 0, X
n + 5  STDESCR DSA, Z
n + 6  SUB V, K
n + 7  SUB V, K
n + 8  ML V
n + 9  L+ 1
n + 10 ST V
n + 11 RETURN ELEM

```

The procedure call "ELEM(S1 . S2)" compiles as

```

n + 0  IB ELEM
n + 1  MLSZE S1
n + 2  MLSLA
n + 3  MLSZE S2
n + 4  LSLA+
n + 5  INITVSP STRASSIGN, X
n + 6  LSTSTR 0, S1
n + 7  LSTSTR 0, S2
n + 8  STDESCRP SDA, X
n + 9  CP ELEM
n + 10 TP

```

(X) The (integer) function declaration

"FUNCTION F2 (VAR I : INTEGER) :INTEGER;

BEGIN

I := I*2

END "

compiles as

```

n + 0  ML I
n + 1  L* 2
n + 3  ST I
n + 4  RETURN F2

```

The function call "F2(J)" compiles as

```

n + 0  IB F2
n + 1  ML J
n + 2  STP I
n + 3  CP F2
n + 4  MLP F2
n + 5  TP

```

B.1.10 Parameter Passing Group (PP)

The Parameter Passing group implements all var parameters and also const-parameter structure variables and string function results. Other parameter passing orders have already been covered. (cf. instructions MLP, STP, STDESCRP, INIVSP: sections B.1.1.-1.5).

Instructions in this group have the format:

m p₁ p₂ where p₁, p₂ are table pointers.

The instructions are: PPA TRD

PPA X, Y

X points to the table entry for an actual parameter which is allocated space in the current activation record. Y points to the table entry for a formal parameter which is allocated space in the activation record which is about to become current.

PPA stores the static address of X space in Y space. The pointer field may be overloaded by a vector; in this case, the index is loaded in ACC.

TRD F, W

F points to the table entry for a function identifier which is allocated result space in the activation record

which just has been current.

W points to the table entry for a work-space string variable which is allocated space in the current activation record.

TRD takes the descriptor describing the function result from F space and copies it to W space.

Examples:

(Y) The procedure declaration

"PROCEDURE QT(VAR S :STRING;SV:VECTOR [1..9] OF STRING);

```

BEGIN
    SV [ I ] := S
END  "
```

compiles as

```

n + 0  SUB SV, I
n + 1  MLSZE S
n + 2  MLSLA
n + 3  INITVS STRASSIGN, SV
n + 4  LSTSTR 0, S
n + 5  STDESCR DSA, SV
n + 6  RETURN QT
```

The procedure call "QT (E, VS)" compiles as

```

n + 0  IB QT
n + 1  PPA E, S
n + 2  PPA VS, SV
n + 3  CP QT
n + 4  TP
```


(Z) The skeleton procedure declaration

```
"PROCEDURE GHF (CONST ST:VECTOR [ 1..10 ] OF STRING;  
I : INTEGER);
```

```
  BEGIN
```

```
    <procedure body>
```

```
  END "      compiles as
```

```
n + 0  <code for procedure body>
```

```
n + m  RETURN GHF
```

The procedure call "GHF (VS, 1)" compiles as

```
n + 0  IB GHF
```

```
n + 1  PPA VS, ST
```

```
n + 2  ML 1
```

```
n + 3  STP I
```

```
n + 4  CP GHF
```

```
n + 5  TP
```

(AA) The function declaration

```
"FUNCTION F1 (CONST S:STRING) : STRING;
```

```
  BEGIN
```

```
    F1 := S . "TAG"
```

```
  END "      compiles as
```

```
n + 0  MLSZE S
```

```
n + 1  MLSLA
```

```
n + 2  MLSZE TAG
```

```
n + 3  LSLA+
```

```
n + 4  INITVS STRASSIGN, F1
```

```
n + 5  LSTSTR 0, S
```

```
n + 6  LSTSTR 0, TAG
```

```
n + 7  STDESCR DSA, F1
```

```
n + 8  RETURN F1
```

The function call "F1(S1) compiles as

```
n + 0  IB F1
n + 1  MLSZE S1
n + 2  INITVSP STRASSIGN, S
n + 3  LSTSTR 0, S1
n + 4  STDESCRP S
n + 5  CP F1
n + 6  TRD F1, W      where W is working space
n + 7  TP
```

B.1.11 Library

The accessing of library routines can be handled within existing orders. Two examples illustrate this:

Examples

(AB) "PUT (FL)" compiles as

```
n + 0  IB PUT
n + 1  PPA FL, X
n + 2  CP PUT
n + 3  TP
```

(AC) "CHR(I)" compiles as

```
n + 0  IB CHR
n + 1  ML I
n + 2  STP X
n + 3  CP CHR
n + 4  MLP CHR
n + 5  TP
```

B.2 Summary of Instruction Mnemonics and Parameters

m = instruction mnemonic g = instruction group

p₁, p₂, p₃ = parameters 1, 2, 3 respectively.

m	p ₁	p ₂	p ₃	g	m	p ₁	p ₂	p ₃	g
ML	0	Z	-	ML	LDSA	op1	P	-	LO
ML↵	0	Z	-	ML	LMATCH	0	P	-	LO
MLEFS	0	P	-	ML	ST	0	Z	-	ST
MLP	0	P	-	ML	STDESCR	c	P	-	ST
MLSLA	-	-	-	ML	STP	0	Z	-	ST
MLSZE	0	P	-	ML	STDESCRP	c	P	-	ST
SUB	0	P	Z	SC	LSTETL	op3	P	-	LSS
SUBZ	0	P	-	SC	LSTHD	op3	P	-	LSS
L+	0	Z	-	LO	LSTMTL	op3	P	-	LSS
L-	0	Z	-	LO	LSTSTR	s	P	-	LSS
L*	0	Z	-	LO	INITVS	op2	P	-	SOI
LDIV	0	Z	-	LO	INITVSP	op2	P	-	SOI
LAND	0	Z	-	LO	CASEJ	k	ca	-	J
LOR	0	Z	-	LO	GOTO	0	P	-	J
LSLA+	-	-	-	LO	IFJ	0	ca	-	J
L<	0	Z	-	LO	UJ	0	ca	-	J
L≤	0	Z	-	LO	EXIT	-	-	-	BPC
L=	s	Z	-	LO	IB	0	P	-	BPC
L≠	s	Z	-	LO	IV	0	P	-	BPC
L>	0	Z	-	LO	RETURN	0	P	-	BPC
L≥	0	Z	-	LO					
CP	0	P	-	PC					
TP	-	-	-	PC					
PPA	0	P	P	PP					
TRD	0	P	P	PP					

Key to Summary Table

ca : code address
P : table pointer
Z : table pointer or 0
c : DSA/VSNUL/FSNUL

k : number of case alternatives
s : loaded/0
op1 : SSTRREPLACE/STRINSERT/PATTERN
op2 : SSTRASSIGN/SSTRDELETE/STRINSERT/STRASSIGN/
STRAPPEND/FSTRDELETE
op3 : SSTRREPLACE/SSTRDELETE/STRINSERT

C.0

In this appendix, we justify our beliefs that:

- (1) it is, in practice, relatively easy to define extensions to an LL(1) grammar and to ensure that the grammar remains LL(1), and
- (2) the notation of a programming language need not necessarily be grossly contorted or manipulated in order to allow its (syntactic) description in terms of an LL(1) grammar.

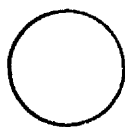
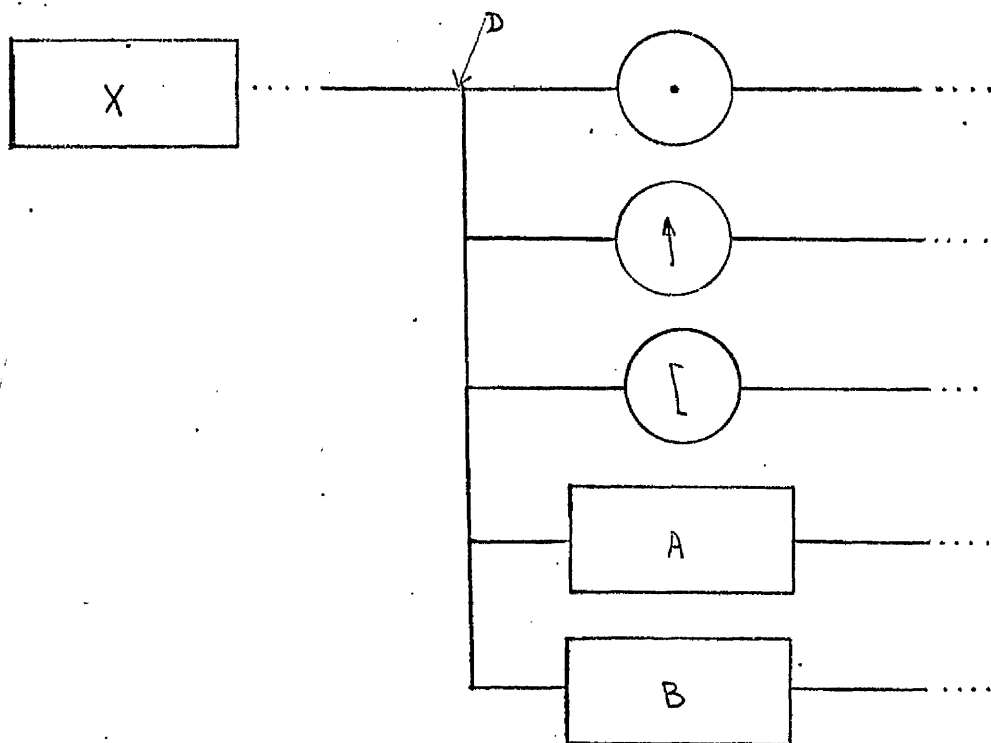
C.1

We define an LL(1) grammar informally as follows:

An LL(1) grammar is a context-free grammar whose sentences can be parsed top-down from left to right with at most one-terminal look-ahead. (Grif 74).

We consider first strong LL(1) grammars i.e. LL(1) grammars in which no non-terminals of the grammar can be expanded to produce the empty string, ϵ .

We can represent an LL(1) grammar by a finite set of separable transition diagrams (Con 63; Wir 71): Each nonterminal of the grammar is represented by a finite-state graph with one entry and one exit point. Each edge connecting two nodes corresponds to a state-transition involving the acceptance either of a basic symbol (if the edge is labelled with that symbol) or of a sentence recognisable by one of the other finite-state graphs (if the edge is labelled with the nonterminal represented by that graph) cf. figure C-1. Transition diagrams appear to provide a useful tool for designing syntax rules and ensuring that they are of LL(1) form cf. (Wir 71). Examples appear



REPRESENTS TERMINALS



REPRESENTS NON-TERMINALS

FIGURE C-1

in section C.2 and in appendix D.

A grammar is LL(1) under the following conditions:

- (a) each transition diagram is deterministic, and
- (b) selection of exit is deterministic (Knut 67).

Equivalently (Grif 74) a grammar is LL(1) if, for each transition graph, and for each divergence of the transition graph, the sets of initial terminal symbols which may occur (one set from each of the alternative branches) are mutually disjoint. We call these sets the starter sets of the divergence point.

Example C-1

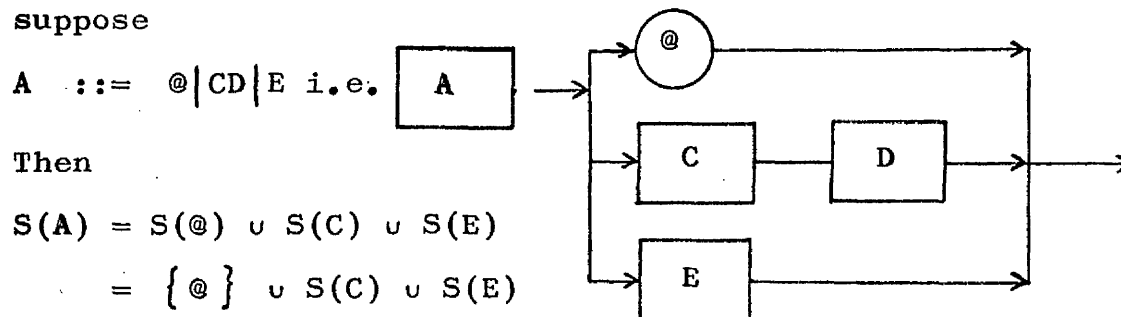
Consider the section of the transition graph X, with divergence point marked D, as shown in figure C-1. We denote the starter symbols which may occur at each branch of the divergence point by $S(\cdot)$, $S(\uparrow)$, $S([\])$, $S(A)$ and $S(B)$ according to the labelling of the branch.

From figure C-1, we see that

$$\begin{aligned} S(\cdot) &= \{ \cdot \} & S(\uparrow) &= \{ \uparrow \} \\ S([\]) &= \{ [\] \} \end{aligned}$$

$S(A)$ and $S(B)$ are determined from the finite-state graphs corresponding to A and B respectively. For example,

suppose



Then

$$\begin{aligned} S(A) &= S(@) \cup S(C) \cup S(E) \\ &= \{ @ \} \cup S(C) \cup S(E) \end{aligned}$$

where $S(C)$ and $S(E)$ must themselves be determined by further elaboration.

The whole grammar is LL(1) if (a) the starters sets at each divergence point are mutually disjoint and

(b) similar conditions exist for the other finite state graphs.

The Empty Symbol

It is often convenient in practice to allow the empty string, ϵ . Unfortunately, it is often harder in this case to determine, merely by inspection whether or not a grammar is LL(1).

When a non-terminal A, say, may produce the empty string, starter symbols include also the terminals which may immediately follow A. We call these terminals the followers of A, denoted by $F(A)$. Thus,

$$S(A) = \begin{array}{ll} \text{first}(A) \cup F(A), & \text{if } A \xRightarrow{*} \epsilon \\ \text{first}(A) & , \text{ otherwise} \end{array}$$

where $\text{first}(A)$ is the set of starter symbols obtained by ignoring the occurrence of the empty symbol ϵ .

Example C-2

We consider the definition of an if-statement cf. figure C-2, at the divergence point marked D,

$$\begin{aligned} S(\underline{\text{ELSE}}) &= \{ \underline{\text{ELSE}} \} \\ S(E) &= F(\text{statement}) \\ &= \{ \underline{\text{ELSE}} . ; \} \quad \text{by inspection of transition} \\ &\quad \langle \text{program} \rangle \end{aligned}$$

These sets are not disjoint and hence the grammar is not LL(1).

Thus, in general, the algorithm to determine whether or not a grammar is LL(1) runs as follows:

- (1) Examine transition diagrams to determine which nonterminals can generate the empty string (either directly or indirectly)..

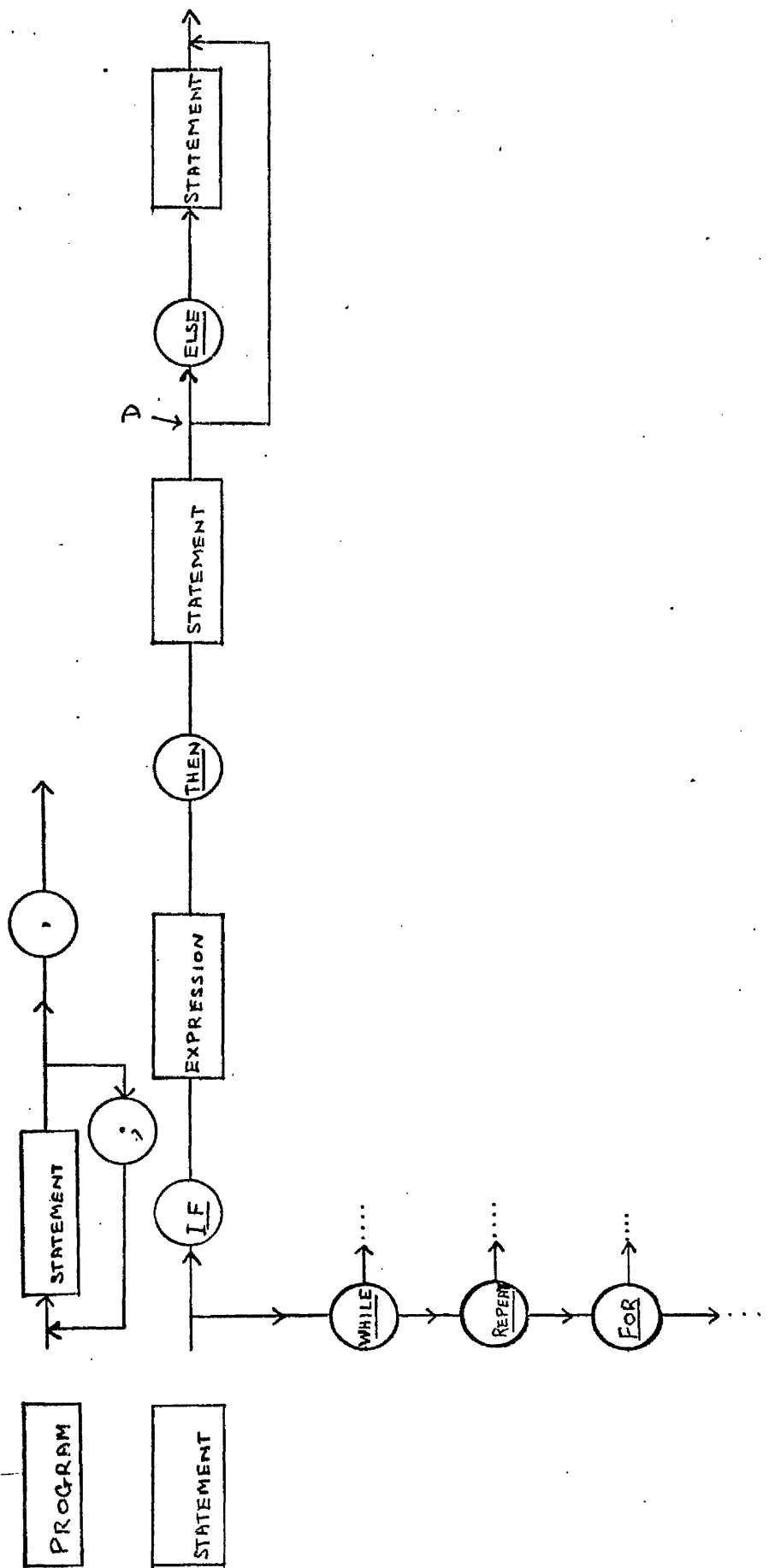


FIGURE C-2 EMPTY SYMBOL EXAMPLE

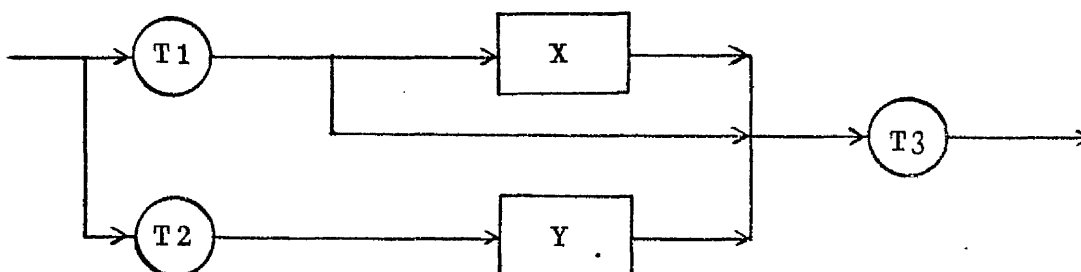
(2) Examine each transition diagram to ascertain whether or not starter sets at each divergence point of the diagram are disjoint; we consider two cases at the (current) divergence point:

- (a) Transition nodes which are either terminals, or non-terminals which do not generate the empty string: starter sets are determined by elaboration of the non-terminals cf. example C-1.
- (b) Transition nodes are non-terminals which may generate the empty string, ϵ : in this case, starters consist of the union of first symbol sets determined as in (a), with the corresponding follower symbol sets which are similarly determined.

In general, and in particularly complex cases, it may be necessary to be more systematic and to iteratively build up matrices of starter and follower symbols as in Griffith's algorithm (Grif 74).

We believe however that the above-noted simple-minded approach will be sufficient for many practical cases. This view is supported by Wirth (Wir 71); appendix D uses Snip transition diagrams to ensure that Snip syntax is LL(1) and thus supports this view. This will at least be true of the case where a transition begins and ends with unique (and disjoint) sets of starters and follower terminals cf. below:

Example C-3



Notation for LL(1) Grammars

Griffiths (Grif 74) observes that an extended form of BNF such as that used in the Vienna definition language is a more suitable notation for LL(1) grammars. In particular, the use of the repetition operator and bracketing reduces the number of recursive rules required in grammars and means that the empty string requires less special treatment. We use asterisk as the repetition operator together with curly brackets: " $\{ \ }^+$ " indicates that the enclosed items must occur one or more times; while " $\{ \ }^*$ " indicates that the enclosed items occur zero or more times.

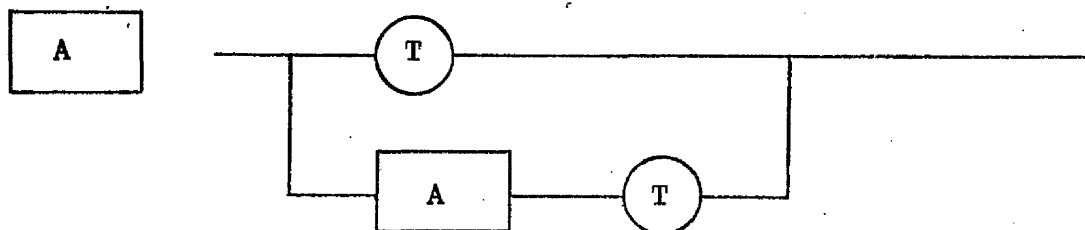
This formalism is more akin to transition diagrams and mirrors more exactly than BNF the way in which we conceive syntax; we consider that it is therefore easier using this formalism to determine by inspection, whether or not a given grammar is LL(1).

We consider a few simple examples as an aid to recognising rules specified in extended BNF which are not LL(1).

(a) Left Recursion

It is not possible for the first symbol on the right-hand side of a production to refer recursively to the symbol on the left-hand side.

e.g.

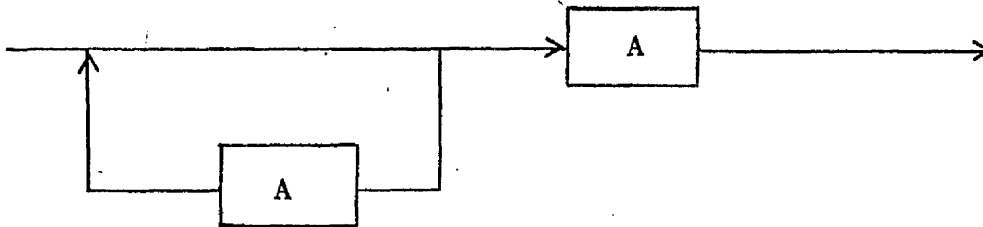


This follows, since $S(T)$ and $S(A)$ are not disjoint:

$$S(A) = S(T) = \{T\}$$

This example illustrates a simple case of left recursion, which no top-down analyser can handle (Grif 74).

(b) It is perhaps not immediately obvious that the construct $\{A\}^*A$ cannot be LL(1); but it is obvious from the corresponding transition diagram section:

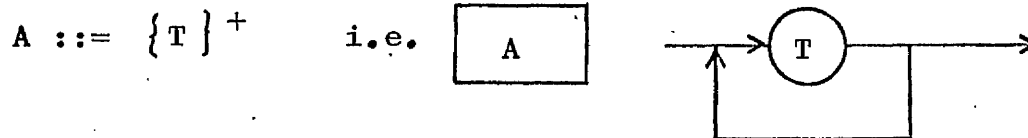


Examples of Moulding Grammar Rules to LL(1) Form

Example C-4 Consider the BNF rule

$$A ::= T \mid TA$$

where A and T are non-terminals. This rule is LL(1) but is more readily seen to be LL(1) when expressed as



Example C-5

Consider BNF rules to define a simple arithmetic expression:

$$EXPR ::= TERM \mid EXPR + TERM$$

$$TERM ::= FACTOR \mid TERM * FACTOR$$

$$FACTOR ::= \underline{ID} \mid (EXPR)$$

where \underline{ID} represents identifiers as terminal symbols. The rules are left recursive in this form. They can however

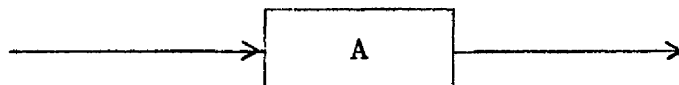
be quite simply expressed in LL(1) form:

$$\begin{aligned} \text{EXPR} &::= \text{TERM} \{ + \text{TERM} \}^* \\ \text{TERM} &::= \text{FACTOR} \{ * \text{FACTOR} \}^* \\ \text{FACTOR} &::= \underline{\text{ID}} \mid (\text{EXPR}) \end{aligned}$$

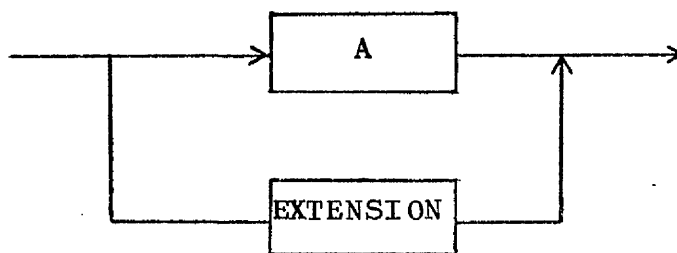
It is perhaps easier to recognise the equivalence from the corresponding transition diagrams (cf. figure C-3).

Extensions

We consider the effect of introducing extensions to LL(1) grammars. Since extensions to our model (cf. section 2.4) may be introduced only by adding new rules to the grammar (i.e. existing rules may not be deleted or replaced), the effect of extensions is to add branches to transition diagrams. Thus, if an existing rule is represented by the transition diagram:

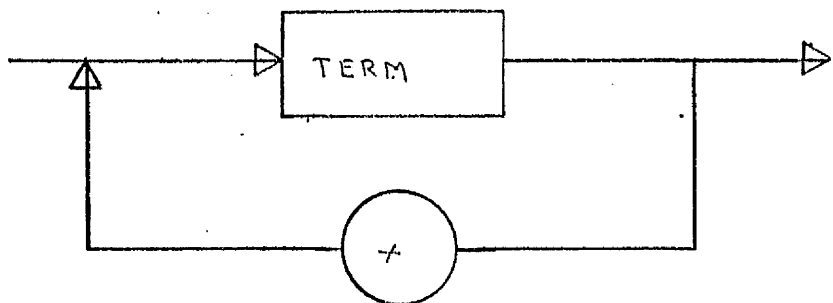


then the extended rule will be represented by:

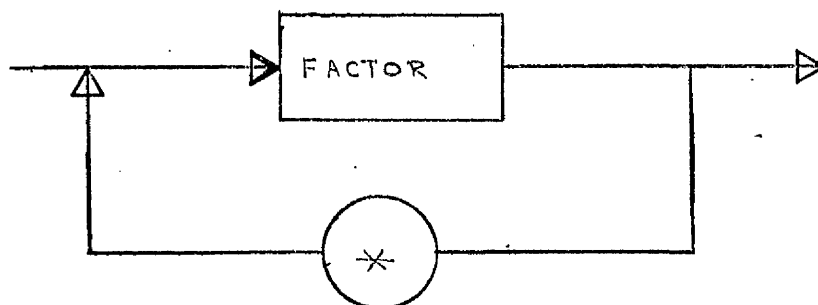


We must ensure that both the extension rule and the modified grammar are LL(1). For simple cases this ought in practice to be relatively easy since it should be possible to work to a large extent from knowledge of the language cf. examples C-2, C-3; examples of extensions

EXPR



TERM



FACTOR

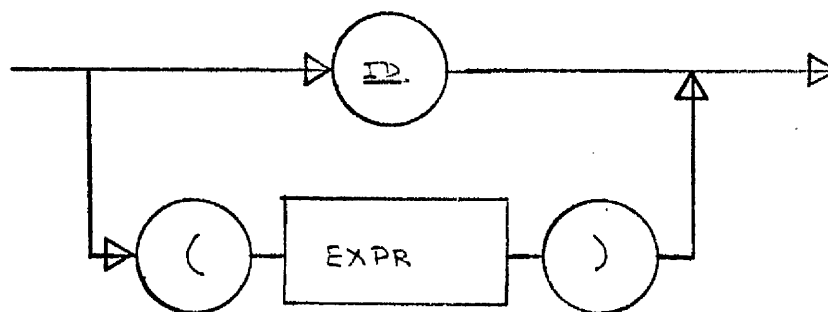


FIGURE C-3

to Snip cf. appendix F support this view. Griffith's algorithm can be used to ensure that the extended grammar is indeed LL(1). This algorithm is fairly lengthy, but provided intermediate tables are retained, it may be possible to avoid repeating the whole process for each extension.

If it turns out that the extended grammar is not LL(1), it will often be possible to rewrite the grammar in a new form which is LL(1). This process cannot be entirely automated (Grif 74): while it is decidable whether or not a given grammar is LL(1), it is, in general, undecidable whether or not the grammar describes an LL(1) language. Griffiths demonstrates, however, that there is an algorithm which goes a long way towards automation.

C.2 Pascal: A Test Case

In this section, we consider the programming language Pascal (Wir 70) as a test case. We consider the problems encountered by Wirth in designing the language to fit LL(1) syntax. Some manipulation of the language was necessary, but we show that this is not particularly complex or prohibitive, nor does it detract from the natural and elegant notation of the language.

(1) Labels

Wirth disallows use of non-integer labels because of the difficulty of description by an LL(1) grammar. The relevant portion of the transition diagram is shown in figure C-4. At point X, the possible sets of starter symbol sets are S(label), S(variable) and S(identifier). Thus, if non-integer labels were permitted, these could be

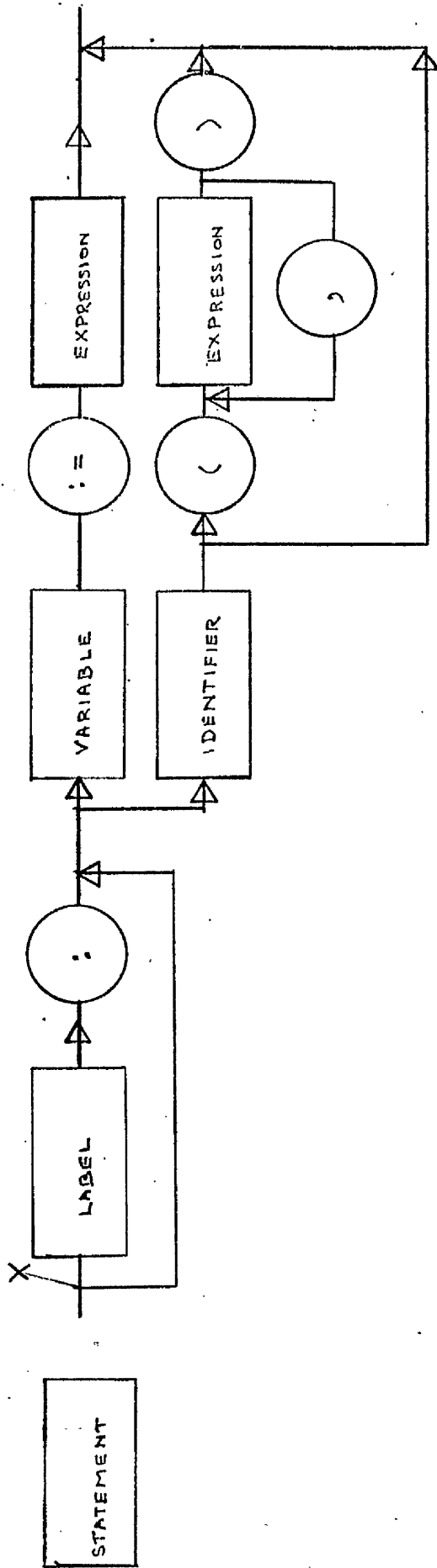
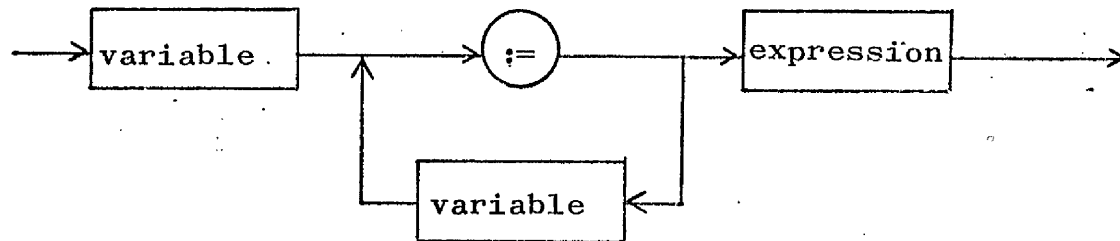


FIGURE C-4

distinguished from identifiers only by looking more than 1 symbol ahead or by use of identifier tables to remove ambiguity. However, we consider, in view of security, that any feature which makes the goto-statement less attractive to use is useful (cf. section 2.1.3).

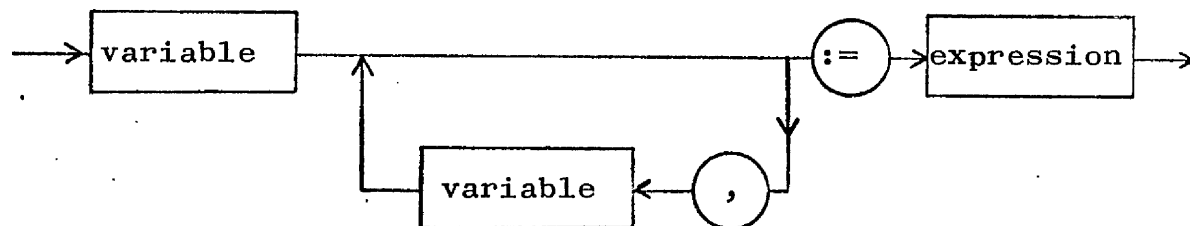
(2) Multiple Assignments

We consider the transition diagram section:



Since $\langle \text{variable} \rangle$ and $\langle \text{expression} \rangle$ may both start with the same terminal, i.e. identifier, it is impossible to describe this statement in terms of an LL(1) rule.

Multiple assignments can however, be easily handled if we introduce a new terminal symbol (cf. PL/1).



It might even be argued that this alternative notation reflects the true structure more precisely.

(3) If-Statements

The Pascal report defines the if-statement in a form which does not reflect an LL(1) structure:

$$\begin{aligned} \langle \text{if-statement} \rangle ::= & \text{ IF } \langle \text{expression} \rangle \text{ THEN } \langle \text{statement} \rangle \mid \\ & \text{ IF } \langle \text{expression} \rangle \text{ THEN } \langle \text{statement} \rangle \\ & \text{ ELSE } \langle \text{statement} \rangle \end{aligned}$$

To indicate the LL(1) structure, we would prefer to rewrite this as:

$$\begin{aligned} \langle \text{if-statement} \rangle ::= & \text{ IF } \langle \text{expression} \rangle \text{ THEN } \langle \text{statement} \rangle \\ & \langle \text{else clause} \rangle \\ \langle \text{else clause} \rangle ::= & \text{ ELSE } \langle \text{statement} \rangle \mid \epsilon \end{aligned}$$

We have already shown (cf. example C-2) that this form of if-statement is in fact ambiguously specified and that the rules are not LL(1).

We can avoid this problem by introducing a special terminator symbol, FI, say (cf. Algol 68). We re-define the else-clause:

$$\langle \text{else clause} \rangle ::= \text{ ELSE } \langle \text{statement} \rangle \text{ FI} \mid \text{ FI}$$

We consider that the introduction of this terminator improves the notation from the point of view of the human reader, rather than impairs it.

(4a) Procedures

We consider the portion of the transition diagram shown in figure C-4. Since variable and identifier both start with the same terminal symbol (effectively, an identifier can be regarded as a terminal symbol, after lexical analysis) the transition diagram is not deterministic with one symbol look-ahead.

At least one Pascal compiler resolves this difficulty by reference to identifier tables. If the identifier denotes a procedure, then the second alternative is chosen; if it denotes a variable, the first alternative

C-11

is chosen. This method of resolution is unsuitable for a general LL(1) parsing strategy and we consider an alternative solution.

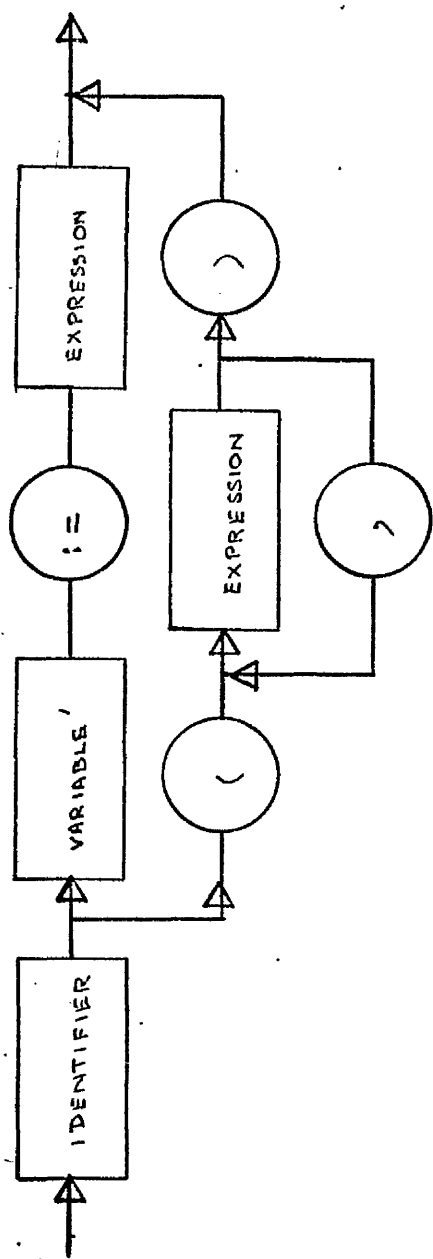
Provided we insist that parameter brackets are always present in a procedure statement, recognition by an LL(1) parser is possible without resort to semantic action. In fact, this restriction is not necessary (although it does make the LL(1) structure more obvious) and the structure is LL(1) provided the transition diagram is rearranged as in figure C-5.

(4b) Functions

A similar situation exists in the finite-state graph for FACTOR. Table-processing is used in the compiler to distinguish between variable identifiers and function identifiers. In this case, however, parameter brackets are already mandatory since functions must have at least one parameter.

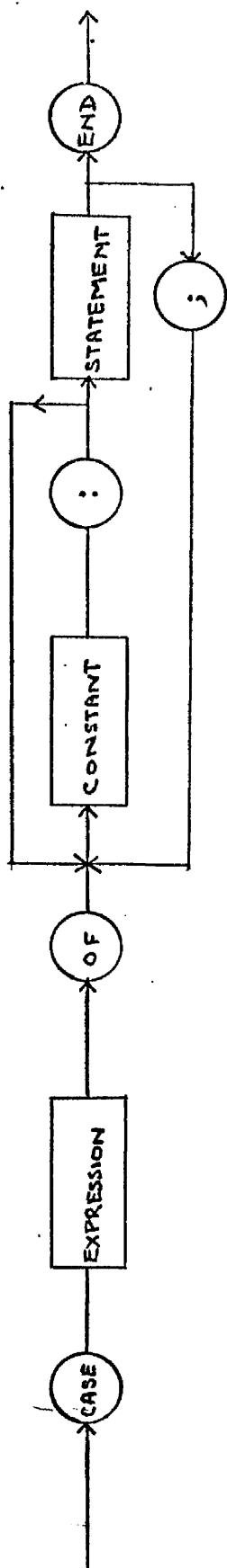
(5) Case Statement

A problem similar to that of multiple assignments arises with case statement labels in (unrevised) Pascal cf. figure C-6(a): a constant in Pascal may be an identifier. Since S(statement) also includes the terminal, "identifier", the diagram is not deterministic with one-symbol look-ahead. At least one Pascal compiler resolves this problem by referring to identifier tables. In revised Pascal, a new terminal is introduced at this point cf. figure C-6(b). We do not consider that this solution compromises the notation.

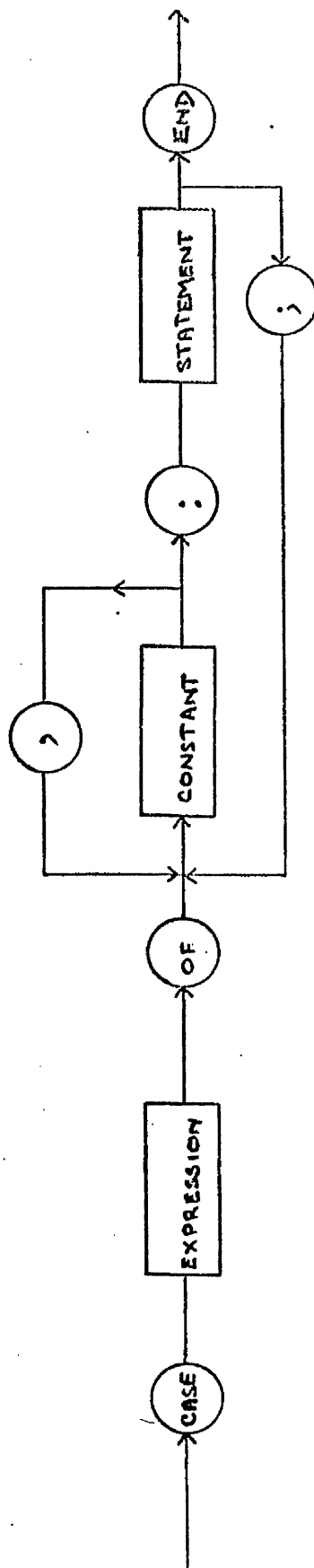


WHERE $\langle \text{VARIABLE} \rangle = \langle \text{IDENTIFIER} \rangle \langle \text{VARIABLE}' \rangle$

FIGURE C-5



(a) ORIGINAL CASE STATEMENT



(b) MODIFIED CASE STATEMENT

FIGURE C-6

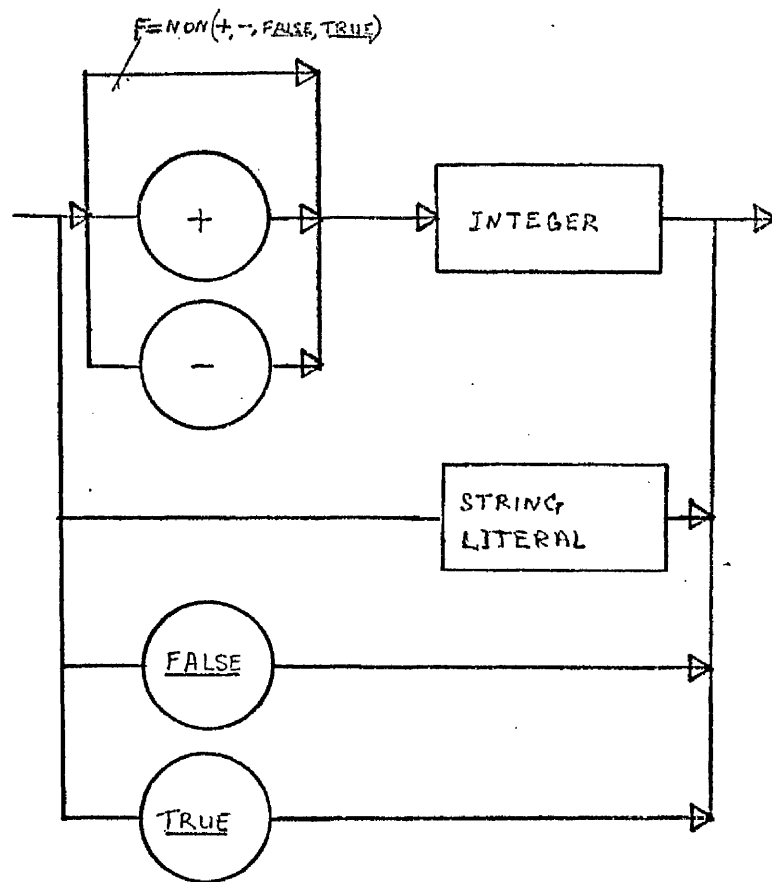
C.3 Conclusions

We have by considering simple examples attempted to illustrate that simple extensions at least, may be defined and verified to be LL(1), without the need of intimate knowledge of the language syntax in total. Appendix F which considers extensions to Snip also supports this view.

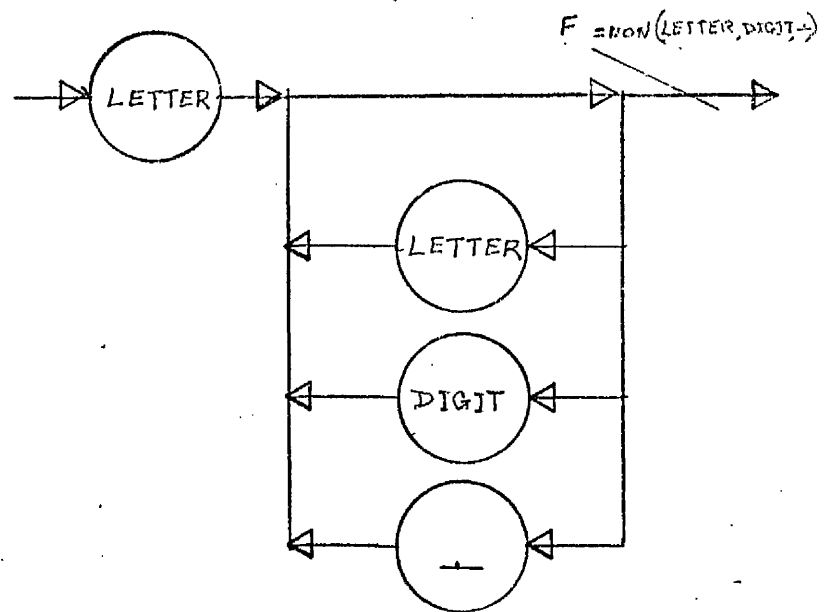
The example of Pascal shows that an LL(1) grammar is capable of defining the syntax of at least one powerful programming language with remarkably few, and easily resolved difficulties. In considering Pascal we demanded that transition diagrams be in the strict sense deterministic, with one symbol look-ahead and without the use of context-sensitive information from identifier tables. The reason for this, in our case, is that without strict determinism, extensions become harder to define and more dependent on translator architecture.

We draw transition diagrams and develop sets of follower and starter symbols where necessary, to verify that the grammar used to describe the Snip syntax is indeed LL(1). The sets of follower and starter symbols are indicated at the appropriate branch points on the transition diagrams.

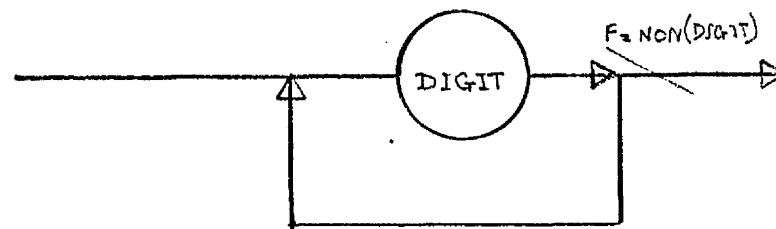
CONSTANT



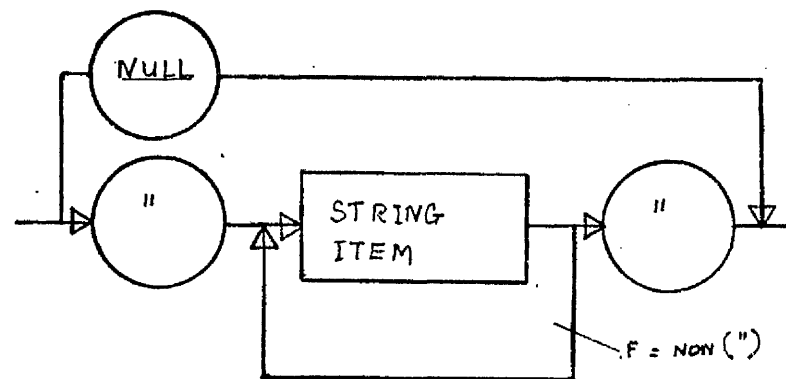
IDENTIFIER

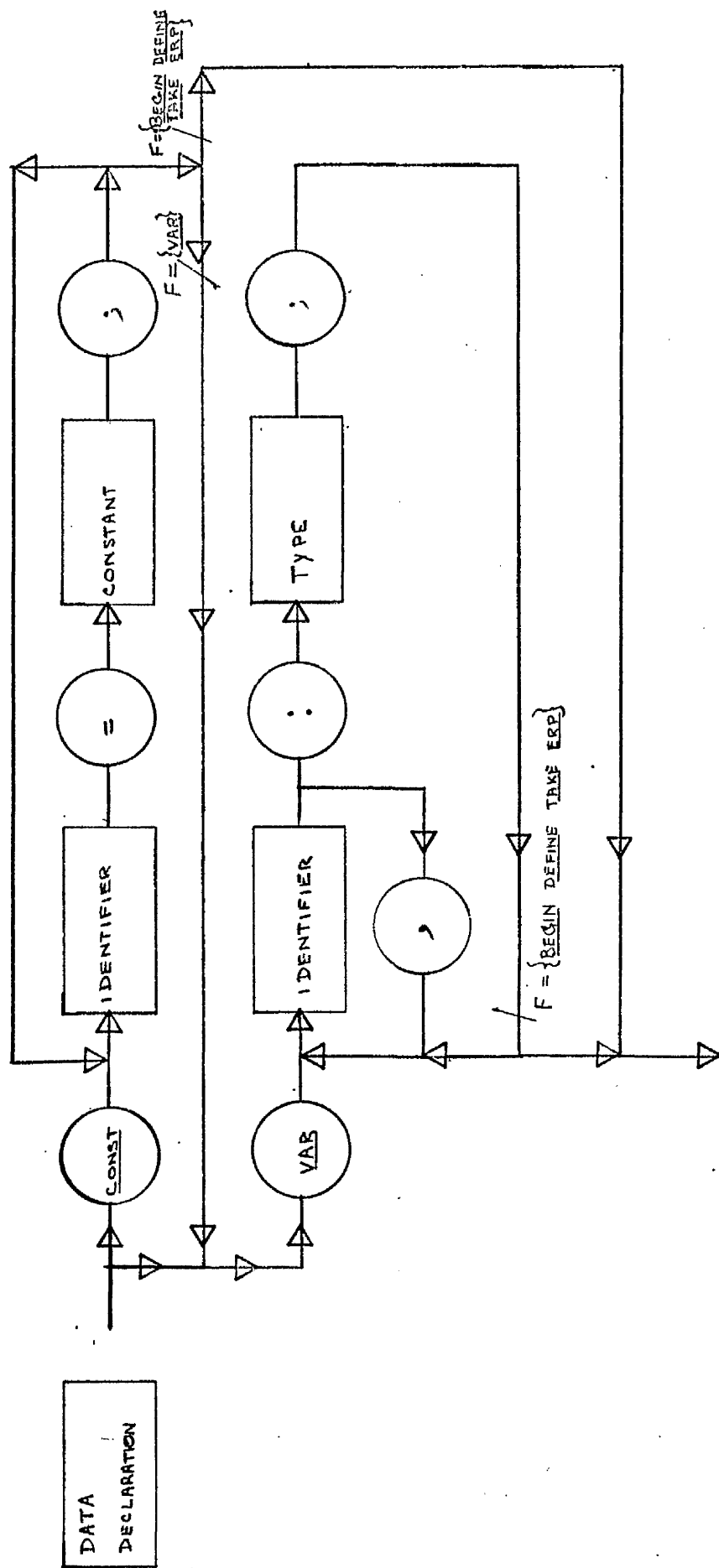


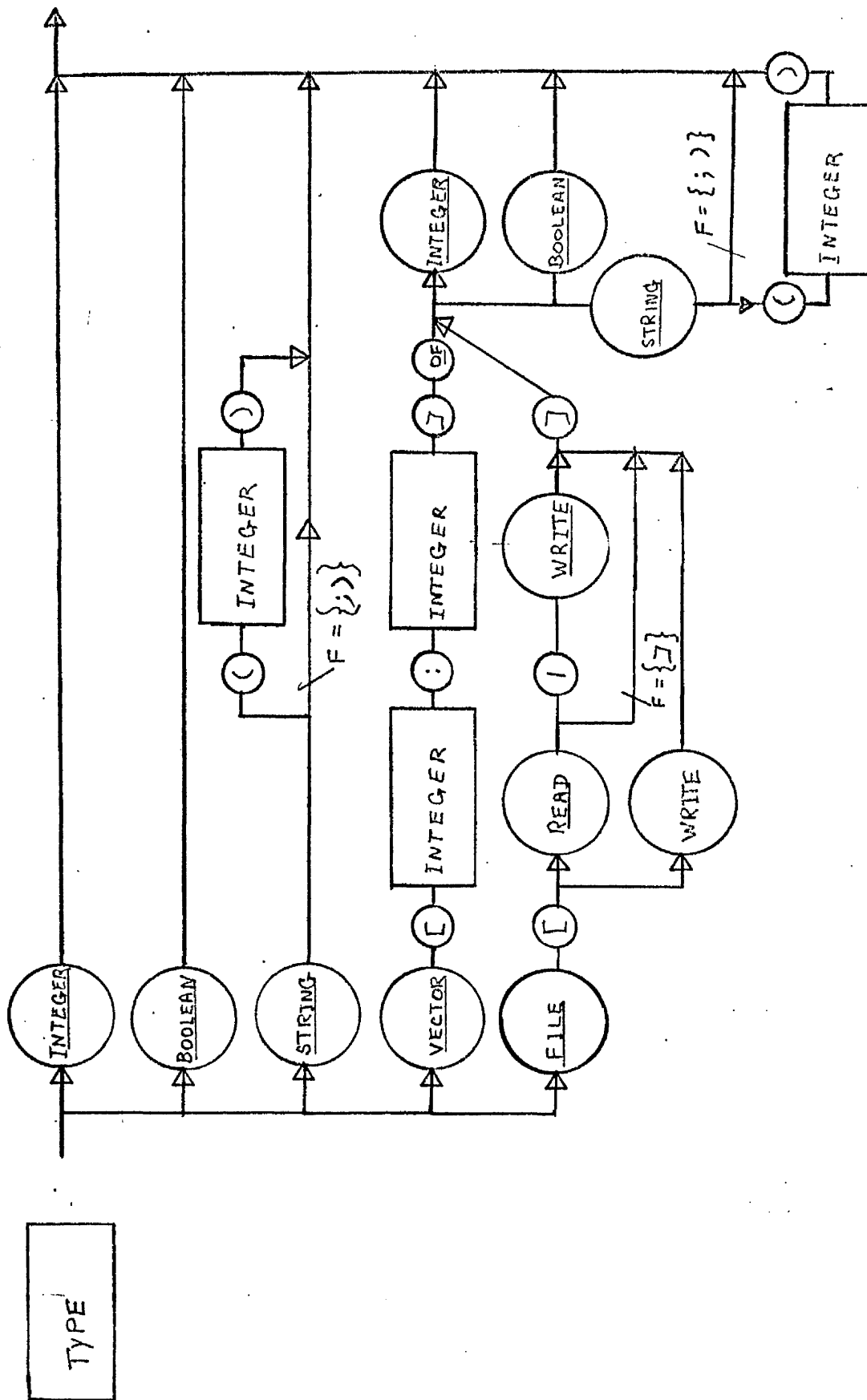
INTEGER

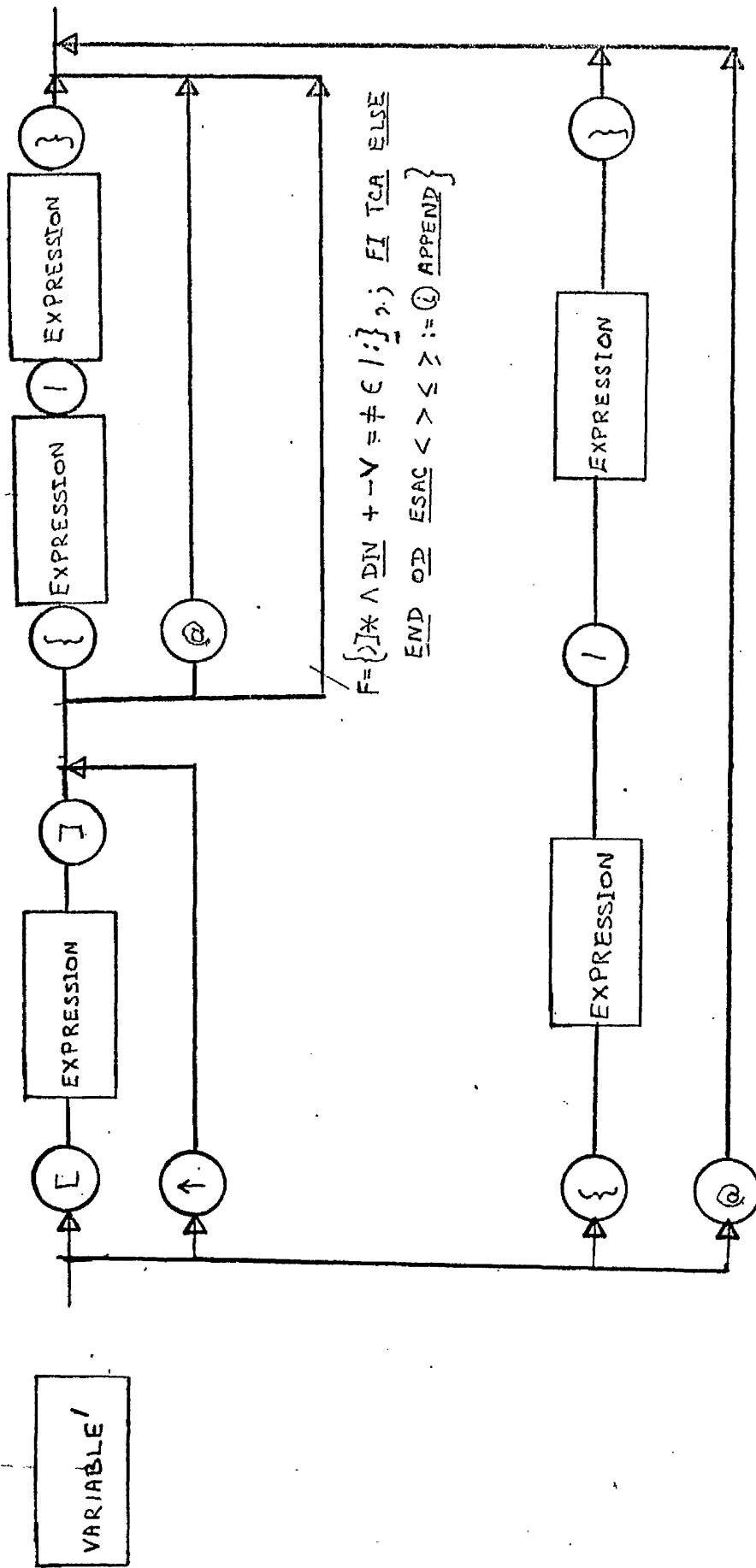


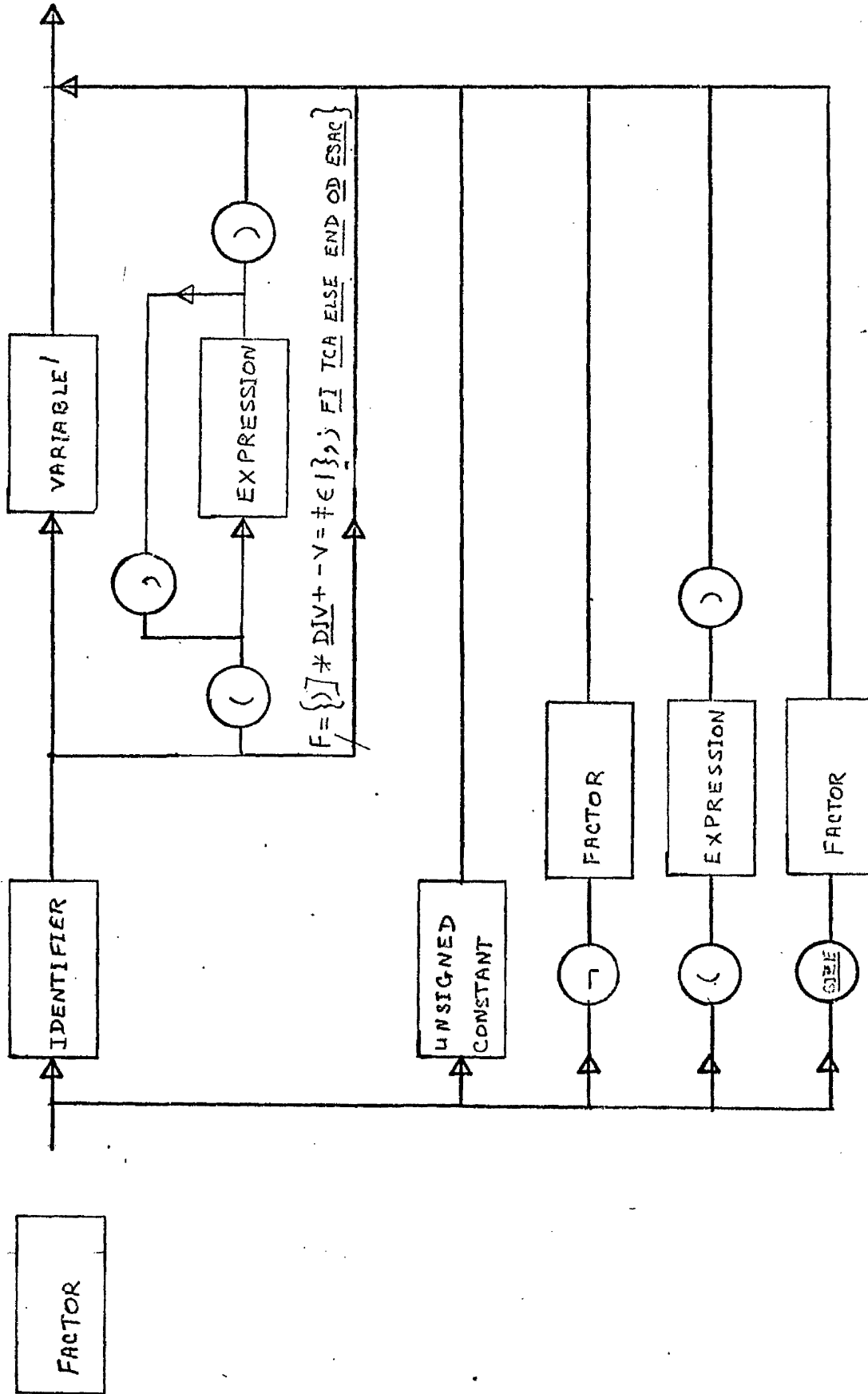
STRING LITERAL

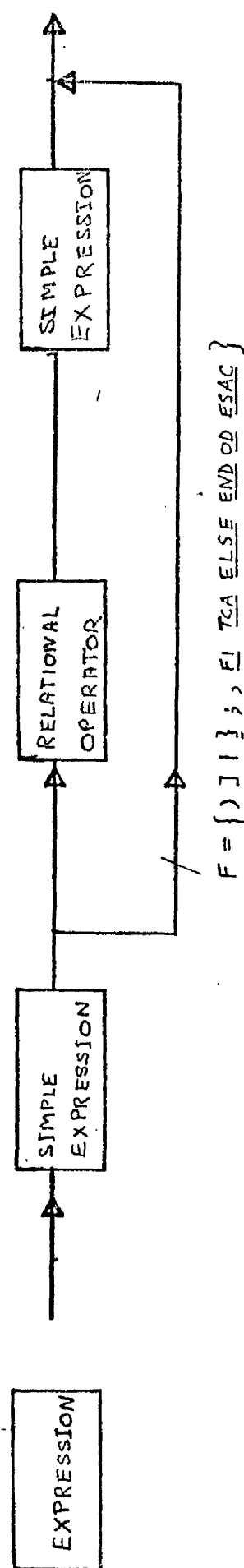
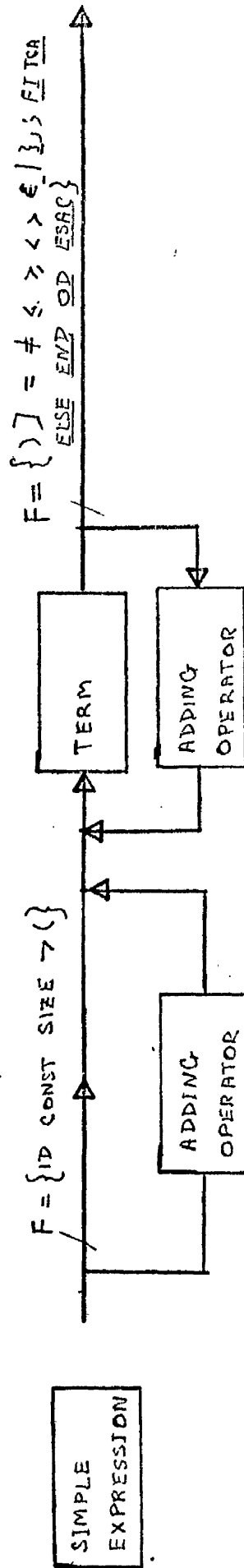
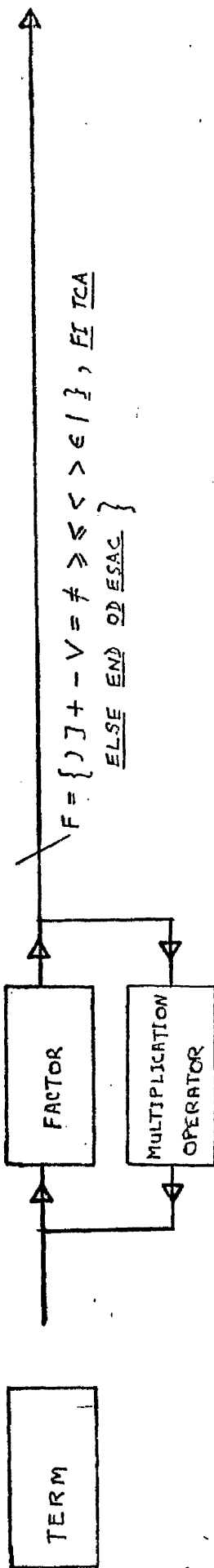


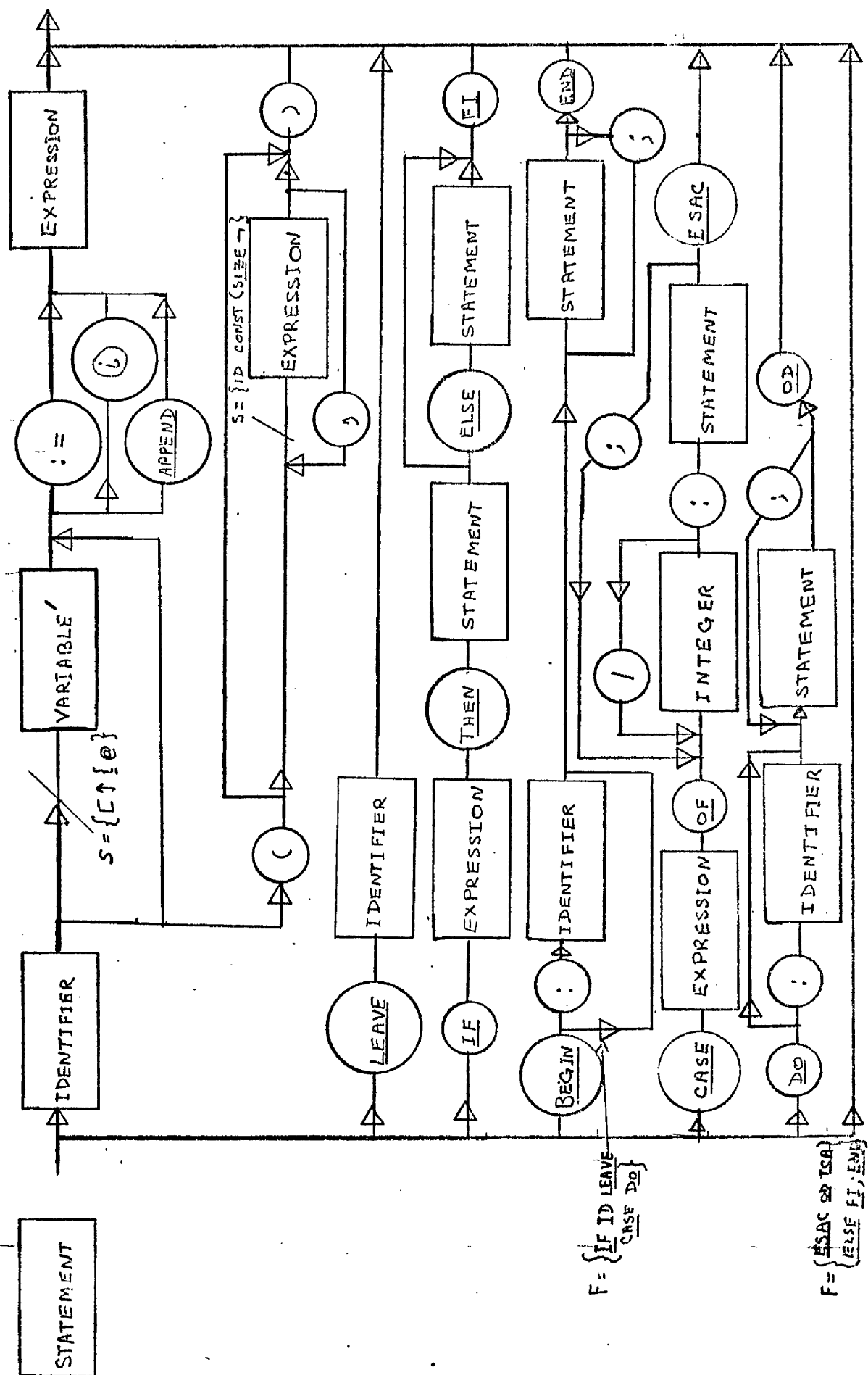


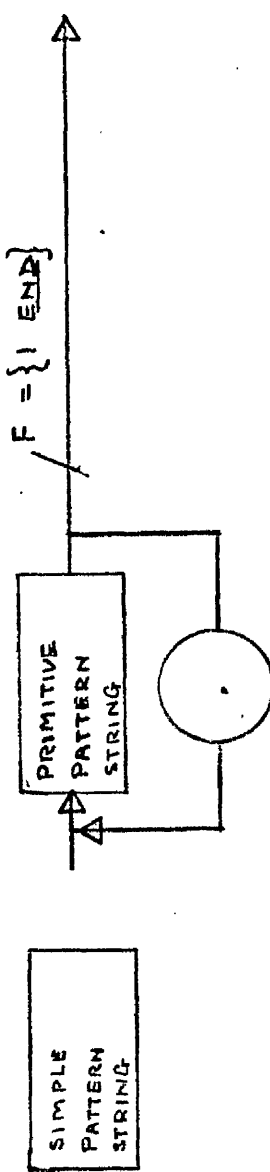


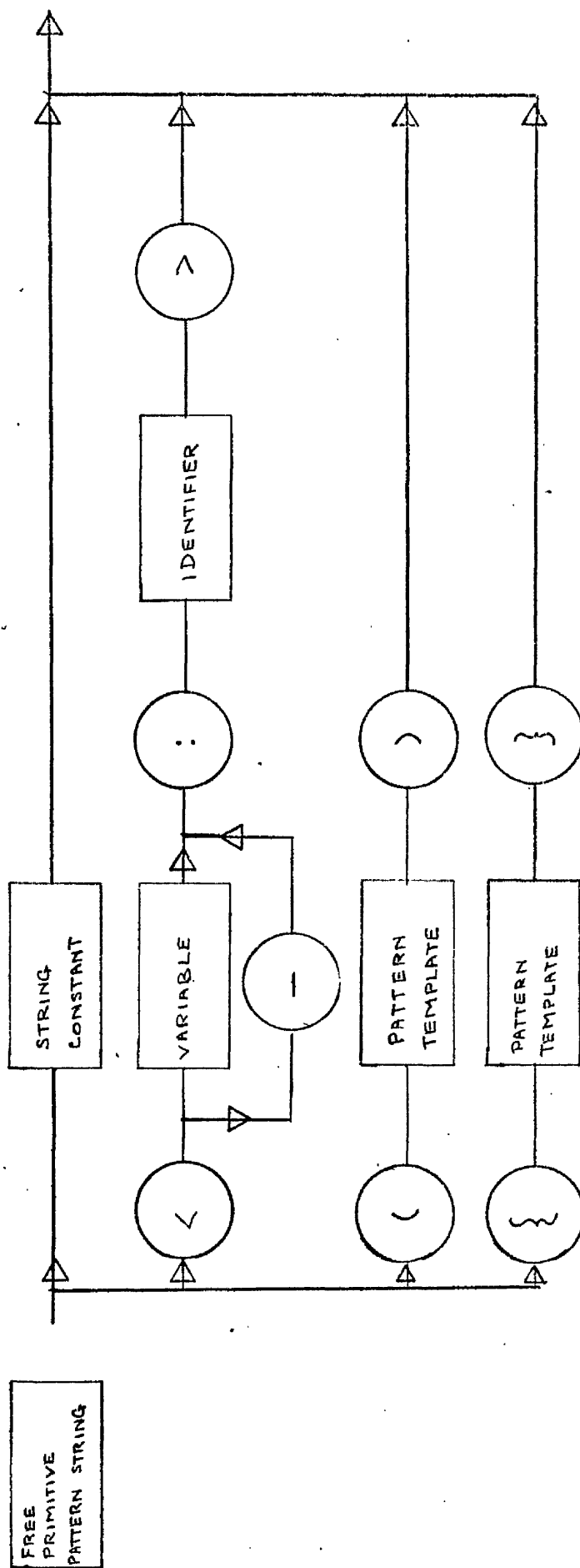
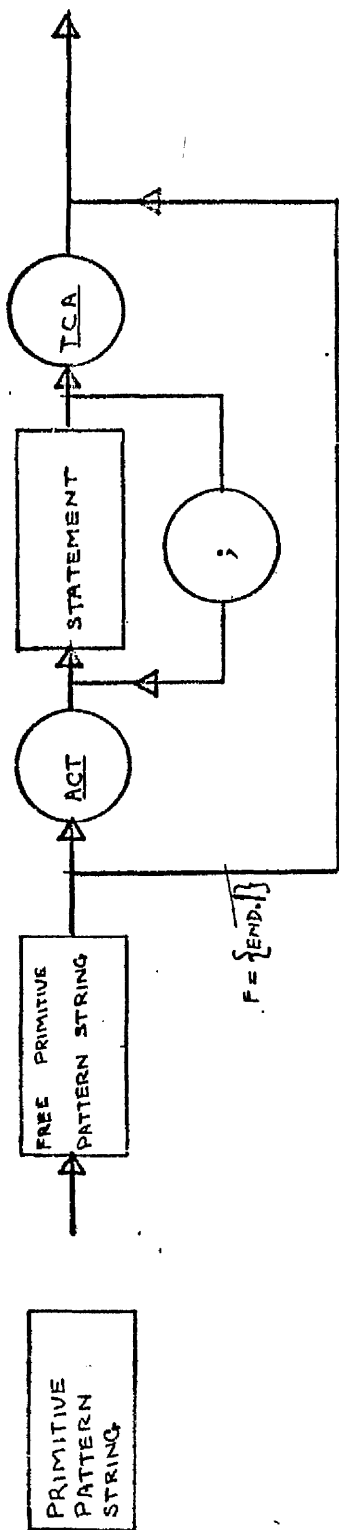


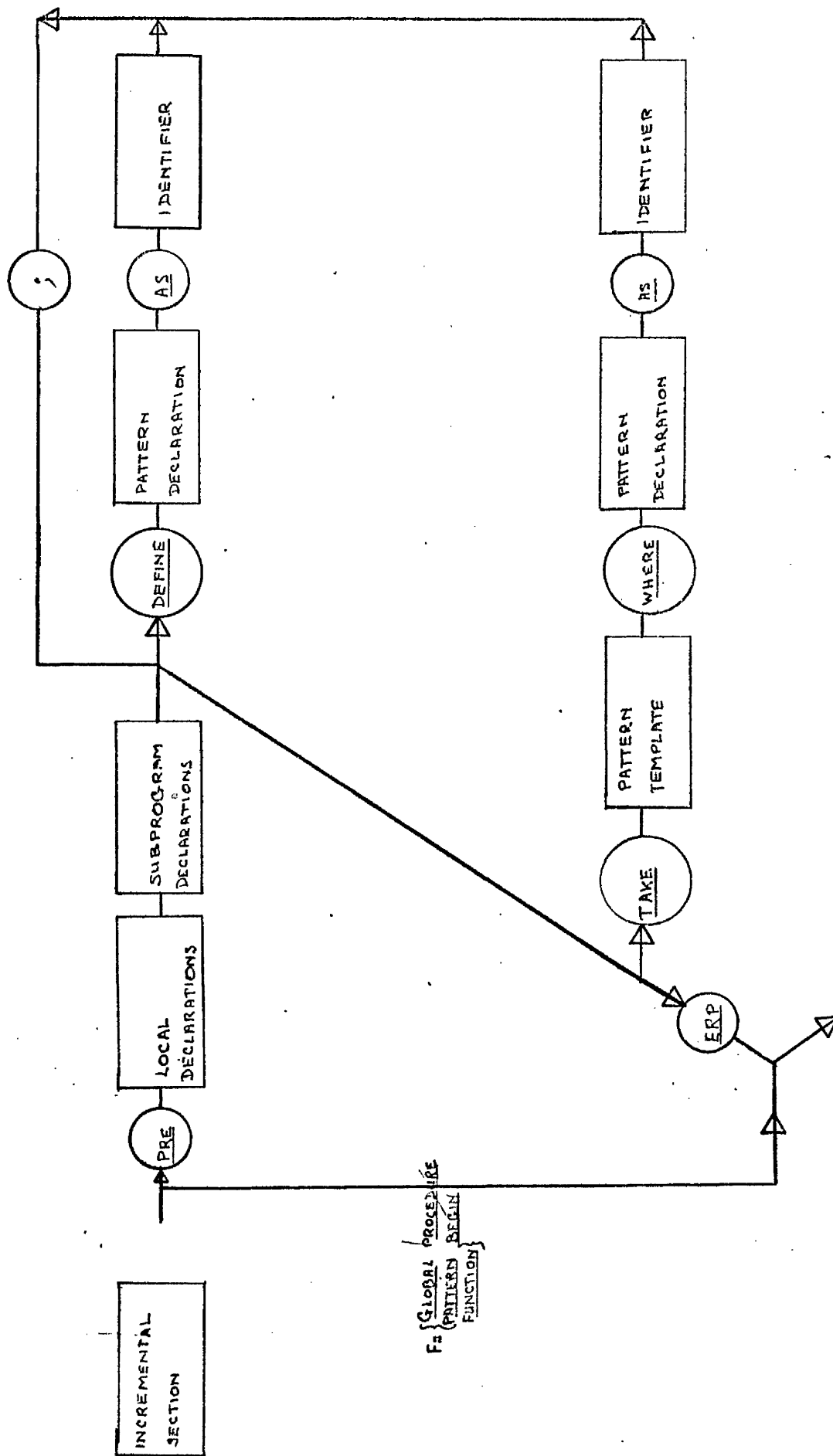


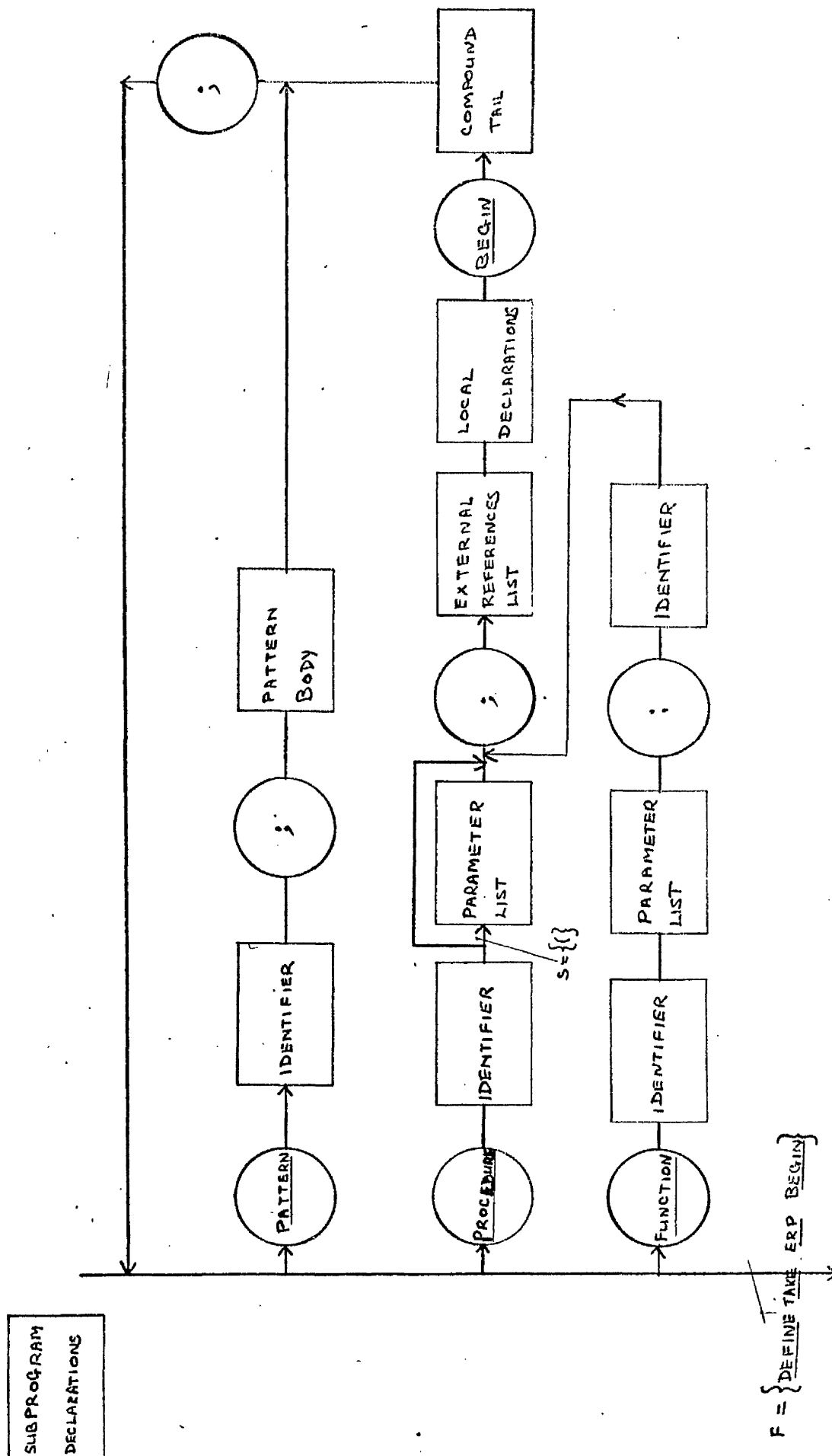




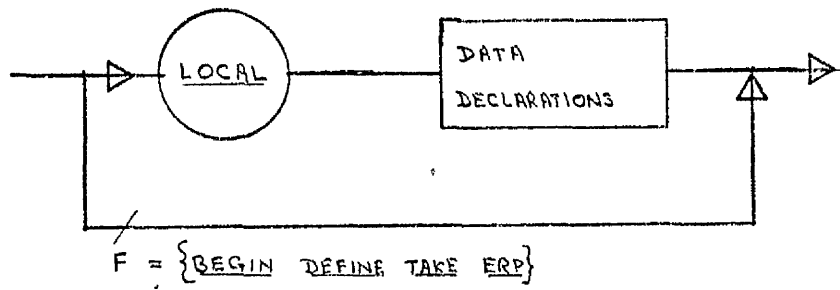




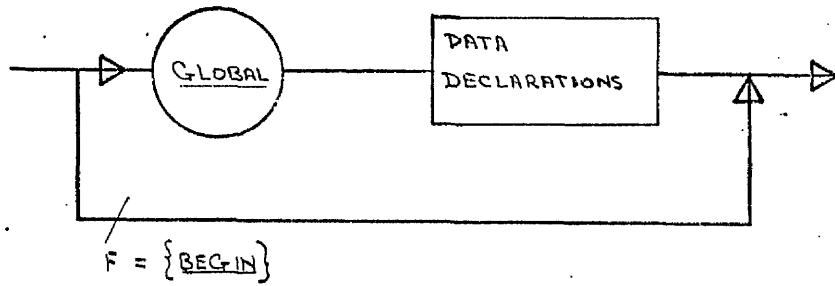


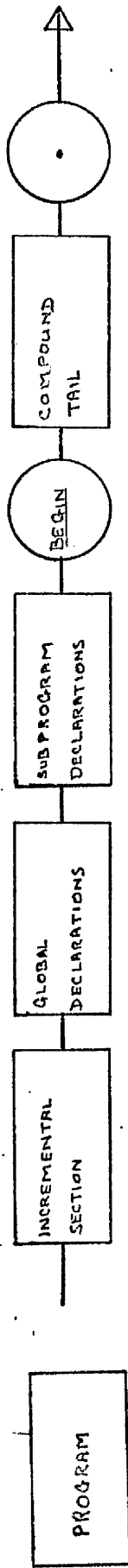


LOCAL
DECLARATIONS



GLOBAL
DECLARATIONS





A translator section to handle simple expressions was written in Snip and hand-translated to SAM. The principal aim in doing so was to obtain some indication of the ease or difficulty of writing a Snip translator in itself. A high proportion of programming errors which occurred arose during the hand-translation process. The program is written using some of the Snip extensions listed in appendix F. It was used to translate sections of small Snip programs, consisting of a series of (restricted) statements.

```

GLOBAL CONST TYPE = 1; ADDR = 2; STATE = 3; /*GATTR,LATTR_
                                indices*/

STR = 1; SUBSTR = 2; CSTR = 3; INT = 4;

CINT = 5; /* variable types */

LOADED = 1; NOT_LOADED = 0; /*states*/

VAR GATTR : VECTOR [ 1..3 ] OF INTEGER; /* attribute
                                record of current expression */

RHS, FIRSTR : BOOLEAN ; /*flags concerning
                                semantic action */

CODE_STR : STRING; /* generated code */

```

PATTERN STATEMENT:

```

BEGIN
  ( "ID" ACT SEARCH TCA . <- : ASSIGN_STATEMENT >
    | NULL ACT FAULT(1) TCA ) .
  { ";" . <- : STATEMENT > }
END STATEMENT ;

```

PATTERN S_EXPRESSION;

```

LOCAL VAR LATTR : VECTOR [ 1..3 ] OF INTEGER; /*local
                                attribute record*/

BEGIN
  <- : TERM > .
  { "+" ACT INT_OP(LATTR)TCA . <-: TERM > ACT
                                ADD_OP(LATTR)TCA }
END S_EXPR;

```


PATTERN TERM;

EXTERNAL CONST GATTR, TYPE;

LOCAL VAR LATTR : VECTOR [1..3] OF INTEGER;

BEGIN

<- : FACTOR> .

{ ("*" ACT INT_OP(LATTR) TCA
 | "." ACT LATTR [TYPE] := GATTR [TYPE] TCA)
 .<- : FACTOR> . ACT MULT_OP(LATTR) TCA }

END TERM ;

PATTERN FACTOR;

BEGIN

"ID" ACT SEARCH TCA . <- : VARIABLE>

| "(" . <- : S_EXPRESSION> . (")" | NULL ACT
FAULT(4) TCA)

| <- : CONST> ACT SEARCH ; STR_ATTR TCA

| NULL ACT FAULT(2) TCA

END FACTOR;

PATTERN VARIABLE;

EXTERNAL CONST GATTR, TYPE;

LOCAL VAR LATTR : VECTOR [1..3] OF INTEGER;

BEGIN /*recognise component denotation of a variable*/

"(" ACT SUB_STR_ATTR(LATTR) TCA .

<-: S_EXPRESSION> ACT SUB_STR_SPEC TCA .

("|" | NULL ACT FAULT(5) TCA) .

<-: S_EXPRESSION> ACT SUB_STR_SPEC TCA .

(")" ACT GATTR:=LATTR TCA | NULL ACT FAULT(5) TCA)

| NULL ACT STR_ATTR TCA

END VARIABLE;

PATTERN ASSIGN_STATEMENT;

EXTERNAL VAR FIRSTR, RHS ;

LOCAL VAR LATTR : VECTOR [1..3] OF INTEGER;

BEGIN

NULL ACT FIRSTR := FALSE ; RHS := FALSE TCA .

<--: VARIABLE> .

("==" ACT RHS_ASSIGN(LATTR) TCA

| NULL ACT FAULT(2) TCA) .

<--: S_EXPRESSION> ACT STORE(LATTR) TCA

END ASSIGN;

PROCEDURE SEARCH ;

EXTERNAL VAR GATTR ;

BEGIN

search _ for _ identifier _ in _ tables ;

set _ up _ global _ attribute _ record

END ;

PROCEDURE FAULT (CONST I:INTEGER);

BEGIN

print _ error _ message

END ;

```

PROCEDURE ADD_OP (CONST LATTR : VECTOR [ 1..3 ] OF INTEGER);
    EXTERNAL CONST ADDR, STATE, LOADED; VAR GATTR,
        CODE_STR;
    BEGIN /* load and add second operand */
        ASSERT operand_types_integer;
        CODE_STR APPEND "LPLUS  $\emptyset$ " .STR(GATTR[ADDR]).
            "EOL";
        GATTR [ ADDR ] :=  $\emptyset$ ; GATTR [ STATE ] := LOADED
            /* new global attribute record*/
    END ADD_OP;

```

```

PROCEDURE MULT_OP (CONST LATTR : VECTOR [ 1..3 ] OF INTEGER);
    EXTERNAL CONST ADDR, STATE, LOADED; VAR GATTR,
        CODE_STR;
    BEGIN /* Load and multiply second operand */
        ASSERT operand_types_integer;
        CODE_STR APPEND "LMULT  $\emptyset$ ".STR(GATTR [ ADDR ] ) .
            "EOL" ;
        GATTR [ ADDR ] :=  $\emptyset$  ; GATTR [ STATE ] := LOADED
            /* new global attribute record */
    END MULT_OP;

```

```

PROCEDURE INT_OP (VAR LATTR : VECTOR [ 1..3 ] OF INTEGER);
    EXTERNAL CONST ADDR, TYPE, NOT_LOADED; VAR GATTR,
        CODE_STR;
    BEGIN /* Load integer operand */
        IF GATTR [ STATE ] = NOT_LOADED THEN
            CODE_STR APPEND "ML  $\emptyset$ ".STR(GATTR [ ADDR ] ).
                "EOL" FI
        LATTR := GATTR; /* local copy of attribute
            record */
    END INT_OP;

```

PROCEDURE RHS_ASSIGN(VAR LATTR:VECTOR [1..3] OF INTEGER);

EXTERNAL CONST GATTR, TYPE, STATE, ADDR;

VAR FIRSTR, RHS;

BEGIN /* set appropriate flags; take local copy
of attribute record */

ASSERT LHS_not_constant;

FIRSTR := TRUE; RHS := TRUE;

LATTR := GATTR

END RHS_ASSIGN;

PROCEDURE SUB_STR_ATTR (VAR LATTR : VECTOR [1..3] OF INTEGER);

BEGIN /* Build parse tree */

ASSERT GATTR_type_string_or_substring;

LATTR := GATTR;

build_parse_record_from_gattr

END;

PROCEDURE SUB_STR_SPEC;

EXTERNAL CONST STATE, ADDR, LOADED, NOT_LOADED

VAR CODE_STR, GATTR;

BEGIN /* Load string offset or length */

ASSERT GATTR_type_integer;

IF GATTR [STATE] = NOT_LOADED THEN

CODE_STR APPEND "ML Ø" . STR(GATTR [ADDR]).
"EOL"

FI ;

GATTR [STATE] := LOADED

END ;

PROCEDURE STR_ATTR;

EXTERNAL CONST GATTR, ADDR, RHS;

```
BEGIN /* load size; build parse record */
```

IF type_string AND RHS THEN

BEGIN

```
build_parse_record_from_gattr;
```

```
CODE_STR APPEND "MLSZE 0" . STR(GATTR [ADDR])
      . "EOL"
```

END

FI ;

SET_SLA

```
END STR_ATTR;
```

PROCEDURE SET_SLA;

```
EXTERNAL CONST RHS; VAR FIRSTR, CODE_STR;
```

```
BEGIN /* load_SLA */
```

IF type_string_or_substring AND RHS THEN

IF FIRSTR THEN

BEGIN

```
FIRSTR := FALSE;
```

```
CODE_STR APPEND "MLSLA 0 0 ".EOL"
```

END

FI

```
ELSE CODE_STR APPEND "LSLA 0 0 ".EOL"
```

FI

```
END SET_SLA;
```

PROCEDURE STORE (CONST LATTR VECTOR [1..3] OF INTEGER);

EXTERNAL CONST GATTR, TYPE, ADDR, STATE, LOADED, NOT_

LOADED, SUBSTR;

VAR CODE_STR;

BEGIN /* generate store operations */

ASSERT types_compatible;

IF LATTR [TYPE] = INT THEN

BEGIN

IF GATTR [STATE] = NOT_LOADED THEN

CODE_STR APPEND "ML \emptyset ".STR(GATTR [ADDR]).

FI;

"EOL"

CODE_STR APPEND "STORE \emptyset ".STR(LATTR [ADDR]).

"EOL"

END

ELSE

BEGIN

CODE_STR APPEND "INITVS \emptyset " . LATTR [ADDR] .

"EOL"

IF LATTR [TYPE] = SUBSTR THEN

CODE_STR APPEND "LSTHD DEQUE".STR(LATTR
[ADDR]). "EOL"

FI;

generate_code_from_parse_record;

IF LATTR [TYPE] = SUBSTR THEN

CODE_STR APPEND "LSTETL DEQUE".STR(LATTR
[ADDR]). "EOL"

FI;

CODE_STR APPEND "STDECSR DSA".STR(LATTR
[ADDR]). "EOL"

END

FI

END STORE;

BEGIN /* Main Program */

 initialise_variables ;

IF STATEMENT \Leftarrow INPUT THEN call_Abstract_Machine FI

END MAIN;

By way of indicating the capabilities of the Snip extension mechanism, we consider some extensions to the Snip base language. As indicated in section 2.4, we expect that the features most conducive to elegant extension of a BCD 1 extension mechanism will be those which have few non-local context sensitive syntactic or semantic features.

Example F-1

We introduce a statement to allow context editing of strings. This has the form

```
CB / <pattern variable> IN <subject string>
                               ← <replacement string>
```

where <subject string> ::= <string variable>

and <replacement string> ::= <string variable>

The leftmost substring of the subject string which matches the pattern variable is replaced by the replacement string. We might define this statement in terms of the semantically equivalent base text:

```
DO : Z
  CURSOR := <subject string> @;
  IF <subject string> ∈ <pattern variable> THEN
    BEGIN
      <subject string> { CURSOR | <subject string> @
                        - CURSOR } := <replacement string>;
      LEAVE Z
    END
  ELSE <subject string> @ := <subject string> @ + 1 FI
OD
```

We express this in Snip as follows:


```

DEFINE PATTERN CONTEXT_REPLACEMENT;

  LOCAL VAR PAT, SSTR, RSTR, BL, CURSOR : STRING;

  BEGIN

    "CB / " . <PAT : PATTERN_VAR> . "IN" .

    <SSTR : STRING_VAR> . "←" . <RSTR : STRING_VAR>

    ACT

      NEW_INT_VAR (CURSOR); NEW_LABEL (BL);

      SUBST ("DO".":".BL.

        CURSOR. ":@" . SSTR. "@;" .

        "IF" . SSTR. "€" . PAT . "THEN".

        "BEGIN".

          SSTR. "{" . CURSOR. "|" . SSTR. "@".

          "-".CURSOR. "}" .

          ":@" . RSTR. ";LEAVE". BL.

          "END".

          "ELSE" . SSTR. "@ :@" . SSTR. "@+1 FI".

          "OD" )

      TCA

    END

```

AS STATEMENT;

Many related constructs oriented towards context amendment can be similarly defined.

Example F-2

We define an assert-statement similar to that of

Algol W. We define ASSERT <boolean expression> as equivalent to

"IF ¬ <boolean expression> THEN ERROR(NUMBER) FI"

where ERROR is assumed to be the translator routine which handles run-time errors, and NUMBER represents an appropriate

error number.

```

DEFINE PATTERN ASSERT_STATEMENT;
  LOCAL VAR B : STRING;
  BEGIN
    "ASSERT" . <B : BOOL_EXPR>
      ACT
        SUBST ( "IF¬" . B . "THEN ERROR(NUMBER) FI" )
      TCA
    END
AS STATEMENT;

```

Example F-3

In generating long substitution strings, text is frequently cleaner if we allow variables to be delimited rather than string literals (cf. macroprocessors). In this example we delimit this form of text by the pair symbols \uparrow and \downarrow and we delimit variables by the pair symbols $|$ and $|$. We find, in fact that this example cannot be completely defined in terms of Snip patterns and we must revert to explicit programming. This example is defined, assuming that the compiler processes text from the input file.

```

DEFINE PATTERN MEXPR;
  LOCAL VAR S,ITEM : STRING; CURSOR : INTEGER;
  BEGIN
    "↑"
    ACT
      ITEM := NEXT(INPUT);
      DO : OUTER
        IF ITEM = "↓" THEN

```

```

IF ITEM = "!" THEN
    BEGIN
        CURSOR := INPUT ↑ @;
        IF ¬(INPUT ↑ ∈ IDENTIFIER) THEN ASSERT
            FALSE
        END
    ELSE
        BEGIN ITEM := INPUT ↑ {CURSOR | INPUT ↑ @
            - CURSOR};
            SUBST (ITEM); ITEM := NEXT (INPUT);
            ASSERT ITEM = "↓"; ITEM := NEXT (INPUT)
        END
    ELSE DO : INNER
        SUBST (" """, ITEM);
        ITEM := NEXT (INPUT);
        IF ITEM = "↓" THEN
            BEGIN SUBST (" " " " " );
                LEAVE OUTER
            END
        ELSE
            IF ITEM = "!" THEN
                BEGIN
                    SUBST (" " " " " );
                    LEAVE INNER
                END
            FI
        FI
    OD
FI
OD

```

TCA

END

AS EXPRESSION;

The text IF | A | THEN | S1 | ELSE | S2 | FI for example is now a legal string expression, equivalent to
"IF" . A . "THEN" . S1 . "ELSE" . S2 . "FI"

Example F-4

We define a simple means of increasing the value of a given variable by 1 e.g. INC I.

DEFINE PATTERN INC_ST;

LOCAL VAR LVAR : STRING;

BEGIN

"INC" . <LVAR:INT_VAR>

ACT SUBST (↑ |LVAR| := |LVAR| + 1 ↓) TCA

END

AS STATEMENT;

Example F-5

We consider the definition of various control structures in this example:

(a) Repeat-statement

```
DEFINE PATTERN REPEAT_ST;  
  LOCAL VAR B, S, BL : STRING;  
  BEGIN  
    "REPEAT" . (":" . <BL : LABEL>  
                | NULL ACT NEW_LABEL (BL) TCA).  
    <S:STATEMENT> .  
    "UNTIL" . <B:BOOL_EXPR>  
    ACT  
    SUBST (↑ DO : | BL |  
           | S | ;  
           IF | B | THEN LEAVE | BL | FI  
           OD ↓ )  
    TCA  
  END  
AS STATEMENT;
```

(b) While-statement

```
DEFINE PATTERN WHILE_ST;  
  LOCAL VAR S, B : STRING;  
  BEGIN  
    "WHILE" . <B : BOOL_EXPR> . "DO".<S:STATEMENT>  
    ACT  
    SUBST (↑ IF | B | THEN REPEAT | S | UNTIL ¬ | B | FI ↓ )  
    TCA  
  END  
AS STATEMENT;
```

(c) Friedman (Fri 74) considers that a while-until statement would be useful. We might define this in Snip as follows:

```

DEFINE PATTERN WHILE_UNTIL_ST;

  LOCAL VAR B1,B2,S,BL:STRING;

  BEGIN
    "WHILE" . <B1 : BOOL_EXPR> . "DO".
        <S : STATEMENT>.
    "UNTIL" . <B2 : BOOL_EXPR>

    ACT
      NEW_LABEL(BL);
      SUBST (↑ REPEAT: | BL|
              | S | ;
              IF ¬ | B1 | THEN LEAVE | BL | FI
              UNTIL | B2 | ↓ )
    TCA
  END

AS STATEMENT;

```

(d) We define a for-statement of the form:

```

FOR <integer variable> [ FROM <integer expression> ]
                        [ BY <integer expression> ]
  ( TO <integer expression> | DOWNTO <integer expression> )
  [ WHILE <boolean expression> ] DO <statement> OD

```

where "[]" indicates options and the FROM- and BY-
expressions are assumed to be 1, if unspecified.

DEFINE PATTERN FOR_ST;

LOCAL VAR I, FROM_EXPR, BY_EXPR, EXIT_EXPR, WHILE_EXPR, ST,
BL, LBY_EXPR, LEXIT_EXPR, EXIT : STRING;
WHILE_MRKR : BOOLEAN;

BEGIN

"FOR" . <I : INT_VAR> .
 ("FROM". <FROM_EXPR:INT_EXPR> | NULL ACT FROM_EXPR
 := "1" TCA).
 ("BY". <BY_EXPR:INT_EXPR> | NULL ACT BY_EXPR := "1"
 TCA).
 ("TO" ACT EXIT := ">" TCA |
 "DOWNT0" ACT EXIT := "<" TCA). <EXIT_EXPR:INT_EXPR> .
 ("WHILE". <WHILE_EXPR : BOOL_EXPR> ACT WHILE_MRKR
 := TRUE TCA |
 NULL ACT WHILE_MRKR := FALSE TCA).
 "DO". <ST:STATEMENT> . "OD"

ACT

NEW_LABEL (BL); NEW_INT_VAR(TL); NEW_INT_VAR(TN);
SUBST (↑ BEGIN : | BL |
 | I | := | FROM_EXPR | ; | LBY_EXPR | :=
 | BY_EXPR | ;
 | LEXIT_EXPR | := | EXIT_EXPR | ;
 DO IF (| I | | EXIT | | LEXIT_EXPR |) ↓);
IF WHILE_MRKR THEN
 SUBST (↑ OR (¬ | WHILE_EXPR |) ↓) FI;
SUBST (↑ THEN LEAVE | BL | FI;
 | S | ; | I | := | I | + | BY_EXPR |
 OD
 END ↓)

TCA

END

AS STATEMENT;

(e) Dijkstra has proposed two new forms of control
 structure (Dij 74; McK 74):

DIF

guard 1 : action 1
 guard 2 : action 2
 guard n : action n

FI

where only those actions which are "unguarded" (i.e. whose guards have the value false) are selected for execution. We will assume normal sequential execution. The statement

DLOOP

```

guard 1 : action 1;
guard 2 : action 2;
.
.
guard n : action n

```

POOLD

has a similar interpretation, but is repeatedly executed until all guards have the value false. The DIF form of statement might be regarded as a generalised case-statement in which the case expression is different for each action. It is also close to the idea of decision tables. This form of statement is useful when guards are not integer or defined-scalar subranges (cf. Pascal) and hence are less appropriate to case-statement evaluation.

We may introduce these structures in Snip as follows:

DEFINE PATTERN DIF_ST;

LOCAL VAR ST, BEXPR : STRING; I, COUNT : INTEGER;

BEGIN

"DIF" . <BEXPR : BOOL_EXPR> . ":" . <ST:STATEMENT>

ACT

SUBST (↑ BEGIN IF | BEXPR | THEN | ST | FI ↓)
TCA .

{ ":" . <BEXPR: BOOL_EXPR> . ":" . <ST:STATEMENT>

ACT

SUBST (↑ ; IF | BEXPR | THEN | ST | FI ↓)
TCA } .

"FID"

ACT SUBST (↑ END ↓) TCA

END

AS STATEMENT;

The DLOOP statement can be similarly implemented, or defined simply in terms of the DIF statement.

(f) We consider the inclusion of case-statements similar to those of Pascal, but labelled by string literals. We observed however in section 3.2.2.2 that we are unable to attain the run-time efficiency allowed by Pascal defined scalars. We must therefore introduce some form of run-time string hashing function in order to handle the case-statement by the method used in Pascal. It is probably therefore equally efficient in many cases to define this form of statement in terms of a Dijkstra if-statement. For certain applications it would be appropriate also to ensure that one and only one guard has the value true.

We thus define

SCASE <string variable> OF

 <string literal 1> : <statement 1>;

 .

 <string literal n> : <statement n>

ESACS

as

DIF

 <string variable> = <string literal 1> : <statement 1>;

 .

 <string variable> = <string literal n> : <statement n>;

FID

DEFINE PATTERN SCASE;

LOCAL VAR LIT,ST, CASEVAR : STRING;

BEGIN

"SCASE" . <CASEVAR : STR_VARIABLE> . "OF" .

<LIT:STR_LITERAL> . ":" . <ST:STATEMENT>

ACT

SUBST (↑ DIF |CASEVAR| = |LIT| : |ST| ↓)

TCA

{ ";" . <LIT:STR_LITERAL> . ":" . <ST:STATEMENT>

ACT

SUBST (↑ ; |CASEVAR| = |LIT| : |ST| ↓)

TCA } .

"ESACS" ACT SUBST (" FID ") TCA

END

AS STATEMENT;

(g) Bochman (Boc 73) has considered the introduction of "multiple-exit statements", an extension of the escape-statement notion. These statements conform to the idea of structured programming and allow, Bochman claims, easier optimisation than simple goto-statements. Essentially, multiple exits combine an escape-from-block with execution of a case-statement after escape.

Example:

BEGIN : BL

: IF A THEN MEXIT 3 FROM BL FI;

:

: IF B THEN MEXIT 2 FROM BL FI;

:

MXEND

1 : X := Y;

2 : Y := Z;

3 : X := Z

ESAC

We might define this as follows:

```

PRE
  LOCAL VAR EXIT_NO : STRING; FIRST : BOOLEAN;
  TAKE
    <-: STATEMENT> . { ";" . <-:STATEMENT> } . ("END" |
      <-: MCASE > )

  WHERE PATTERN MCASE;
    EXTERNAL CONST EXIT_NO;
    LOCAL VAR ST,BL: STRING;
    BEGIN
      "MEXEND"
      -ACT
        SUBST ( ↑ |EXIT_NO| := 1 END ;
          CASE |EXIT_NO| OF ↓ )
      TCA
    END
  AS COMPOUND TAIL;

  DEFINE PATTERN MEXIT;
    EXTERNAL CONST EXIT_NO;
    LOCAL VAR INT, BL: STRING;
    BEGIN
      "MEXIT" . <INT:INTEGER> . "FROM" . <BL : LABEL>
      ACT
        IF FIRST THEN BEGIN NEW_INT_VAR(EXIT_NO);
          FIRST:=FALSE END FI;
        SUBST ( ↑ BEGIN |EXIT_NO| := |INT| ;
          LEAVE |BL| END ↓ )
      TCA
    END
  AS STATEMENT;
ERP

```

Similar constructs may be defined for control structures such as repeat- and while-statements.

This example shows that there is a good case for allowing the user of an extensible language facility of this form to define synonymous lexical items (cf. Newey (New 68))

e.g. ESAC and ENDEX etc.

Example F-6

We might improve the notation for patterns for particular applications:

(a) Optional notation $[A]$ etc. may be defined by the statement:

```

DEFINE PATTERN P_OPTION;
    LOCAL VAR ENC: STRING;
    BEGIN
        " [ " .. <ENC : PATTERN_TEMPLATE> . " ] "
        ACT
            SUBST ( ↑ ( | ENC | | NULL ) ↓ )
        TCA
    END
AS FREE_PRIMITIVE_PATTERN_STR;

```

(b) The ARB notation of Snobol 4 might be simulated by the following extension, provided matching occurs on the input file only.

```

DEFINE PATTERN ARB;
    BEGIN
        "ARB"
        ACT
            SUBST ("NULL");
            INPUT ↑ @ := INPUT ↑ @ + WORD_LENGTH
        TCA
    END
AS FREE_PRIMITIVE_PATTERN_STR;

```

(c) The built-in pattern ANY, of Snobol 4 might be defined as follows:

```

DEFINE PATTERN ANY;
    LOCAL VAR LIT:STRING;
    BEGIN
        "ANY" . "(" . <LIT : STR_LITERAL>
            ACT
            SUBST (↑ ( !LIT ↓)
            TCA .
        { ", " . <LIT : STR_LITERAL>
            ACT
            SUBST (↑ | !LIT ↓)
            TCA }
        . ")" ACT SUBST ("") TCA
    END

```

AS FREE_PRIMITIVE_PATTERN_STR;

Snip Program Example F-7

We introduce a program example which uses some of the extensions considered in the foregoing text.

The program searches a "rectangular string" of characters (of width 3) for an occurrence of the 2-dimensional pattern.

$$\begin{aligned}
 &+ (\emptyset \mid -) (\emptyset \mid -) + \\
 &+ (\emptyset \mid -) (\emptyset \mid -) + \\
 &+ (\emptyset \mid -) (\emptyset \mid -) +
 \end{aligned}$$

We would expect the input/output procedures INTEXT, OUTTEXT, WRITE to be pre-defined, but we introduce them for completeness.

```

PROCEDURE INTEXT (VAR S : STRING);
  EXTERNAL VAR INPUT;
  LOCAL VAR CH : STRING(1) ; /* Current character */
  BEGIN /* Read text string */
    S := NULL ;
    CH := NEXTCH (INPUT) ; ASSERT CH = "";
    DO : READ_LOOP
      CH := NEXTCH (INPUT);
      IF CH = "" THEN LEAVE READ_LOOP
      ELSE S APPEND CH
      FI
    OD
  END INTEXT;

```

```

FUNCTION NEXTCH (VAR F:FILE [READ] OF STRING (81)):STRING(1);
  BEGIN /* Fetch next character */
    IF EOF (F↑) THEN BEGIN /* Fetch new component */
      GET (F)
      F↑ @ := ∅
    END
    FI;
    NEXTCH := F↑ { F↑ @ | 1 } ; INC F↑ @
  END NEXTCH;

```

PROCEDURE OUTTEXT (VAR S : STRING);

EXTERNAL VAR OUTPUT;

LOCAL VAR LCURSOR : INTEGER;

BEGIN /* Print text string */

LCURSOR := S@; S@ := ∅; /* Local copy */

REPEAT : PRINT_LOOP

CH := S { S@ | 1 }; INC S@: /* Next character */

PUTCH (CH , OUTPUT)

UNTIL EOF(S);

S@ := LCURSOR /* Restore */

END OUTTEXT;

PROCEDURE PUTCH (CONST CH:STRING(1); VAR F:FILE [WRITE] OF
STRING (121));

BEGIN /* Print character */

F ↑ { F ↑ @ | 1 } := CH ; INC F ↑ @ ;

IF (CH = EOL) ∨ EOF(F ↑) THEN

BEGIN /* Remainder of component (if any)

is NULL. Print line */

F ↑ { F ↑ @ | SIZE F ↑ - F ↑ @ } :=

EOL;

PUT (F)

END

FI

END PUTCH;

PROCEDURE WRITE (CONST NUM:INTEGER);

LOCAL CONST INT_SIZE = 10; /* 32-bit machine */

VAR S:STRING; DIGIT, I, LNUM : INTEGER;

BEGIN /* Write integer */

S := NULL; LNUM := NUM; /* Local copy */

REPEAT /* Convert to string */

DIGIT := LNUM MOD 10; LNUM := LNUM DIV 10;

/* Next Digit */

S := CHR (DIGIT + INT ("0")).S

UNTIL LNUM = 0;

OUTTEXT (S)

END WRITE;

GLOBAL CONST LB= 1, UB = 3; /* Bounds for LINE, COORD */

VAR I:INTEGER; /* Index for LINE */

COORD: VECTOR [LB..UB] OF INTEGER;

/*Coordinates of matched strings */

LINE: VECTOR [LB..UB] OF STRING; /* 2D-string*/

PATTERN COMPONENT; /* 1-dimensional pattern component */

EXTERNAL CONST LINE, I; VAR COORD;

BEGIN

"+" . ("-" | "0") . ("-" | "0") . "+"

ACT COORD [I] := LINE [I] @ - 4 /* match
coordinate */ TCA

END COMPONENT;

FUNCTION MATCH_2D (RSTRING:VECTOR [LB..UB] OF STRING):
BOOLEAN;

EXTERNAL CONST COORD;

LOCAL VAR J: INTEGER; /* Index for RSTRING */

BEGIN /* 2-dimensional match */

FOR J FROM LB TO UB DO

BEGIN : MATCH_LOOP /* 1-dimensional match */

IF LINE [J] \in COMPONENT THEN

IF J < UB THEN LINE [J+1] @ := COORD [J]

/* Set new cursor value */

ELSE MEXIT 2 FROM MATCH_LOOP FI

FI

MEXEND

1 : MATCH_2D := TRUE;

2 : MATCH_2D := FALSE

ESAC

END MATCH_2D

BEGIN : MAIN

FOR I FROM LB TO UB DO INTEXT (LINE [I]);

REPEAT : R /* Exhaustive string match */

IF MATCH_2D (LINE) THEN MEXIT 2 FROM R

ELSE INC LINE [LB] @ /* Next character */

FI

MEXUNTIL EOF (LINE [LB])

2 : FOR I FROM LB TO UB DO WRITE (COORD [I]);

1 : OUTTEXT ("NO MATCH")

ESAC

END.

Some studies of errors in Algol W and Algol 60 have been carried out by Pirie (Pir 75). Pirie collected error statistics for programs written by undergraduate students with a view to devising a means of automatically grading student programs. We use Pirie's results to consider 200 programs (10 programs written by each of 20 students) in each language. A total of approximately 6000 statements are studied for each language.

Characteristic errors are dependent to some extent on the programmers and the application area. Thus, for example, we would expect student programmers to make many trivial errors due to lack of understanding and lack of fluency. Since we expect that different kinds of errors are liable to occur in longer, more complex programs, we examine also some programs written by postgraduate students in Algol W and Algol 60. We examine approximately 3000 statements in each language. We attempt no comparison of these groups because of the relatively small number of errors.

We are interested only in a quantitative and not a qualitative study of characteristic errors: we need to know only those errors which are likely to occur frequently.

We compare the figures obtained for Algol W with those obtained for Algol 60 only in a very broad sense. We have several reasons to expect that they are not directly comparable:

- (a) The programs were written by different groups of programmers.

(b) The undergraduates working in Algol W were taught to use structured programming techniques, while those writing in Algol 60 were not. The Algol W exercises were carefully graded in terms of increasing complexity, while those in Algol 60 were not. The sets of programs were in any case different.

The error classification scheme used by Pirie is in many cases insufficiently detailed for our purposes and we subdivide many of his categories. Since we are particularly interested in errors which are undetected or insufficiently diagnosed, we re-classify errors as

- (1) Compile-time detected,
- (2) Run-time detected, and
- (3) Undetected.

The results obtained are summarised in figure G-1. We discuss and explain the classification scheme:-

Type 1 Compile-time detected errors

We are interested in particular in error patterns which suggest that certain constructs in other languages are prone to instability. (For example, if quotes are frequently omitted from string literals e.g. "A", there is some risk of confusion with an identifier of the same name).

1.1 Errors detected by context sensitive checking

1.1.1 Undeclared variable identifiers.

1.1.2 Mis-spelled identifiers (undetected in Fortran or Snobol).

1.1.3 Type incompatibility (undetected in typeless languages).

1.1.4 Incorrect number of subscripts.

1.2 Errors caused by scope difficulties

1.2.1 Reference to (currently) undefined label in inner scope (does not occur in sample Algol W programs as students used structured programming techniques).

1.2.2 Identifier declared two or more times in same scope.

1.3 Errors caused by omission of symbol(s)

1.3.1 Mandatory space omitted (occurs in Algol W because of the use of reserved words as program delimiters).

1.3.2 Comma omitted.

1.3.3 Semi-colon omitted.

1.3.4 String quote omitted.

1.3.5 Array identifier omitted from array designator (undetected in Coral 66 where $\left[v \right]$ refers to v^{th} location in core).

1.4 Simple Syntax Errors.

1.4.1 Mismatched brackets in arithmetic expression or begin-end pairs in block structure.

1.4.2 Other syntax errors in which we find no patterns, or at least no patterns of interest e.g. punching error, semi colon before ELSE etc.

Type 2 Run-time detected errors.

Some run-time detected errors (for example time/page limit, overflow failure) are insufficiently diagnosed. Ideally, in this situation, we should ascertain the true cause of these errors, and hence classify them as undetected errors. However, it was not possible to do this using the

source of information available. Since the number of badly diagnosed errors ought not to be large (compared to the number of undetected errors) and since we are interested only in a quantitative analysis, we do not expect our results to be radically affected.

2.1 Type incompatibility detected at run time. Frequently caused by input data error.

2.2 Over-reading input data file.

2.2.1 In Algol W programs this error was frequently caused by mis-use of read and readon procedures.

2.2.2 Others.

2.3 Array subscript out of range.

2.4 Case/switch expression out of range.

2.5 Overflow.

2.6 Time or page limit exceeded.

2.7 Assert failure. Assertion statements as such are provided only in Algol W (but could be explicitly programmed in Algol 60 - although students, at least, rarely appear to have done this).

2.8 Error occurring in standard function (due to incorrect data).

Type 3 Undetected Errors

Errors which are undetected at either compile or run-time. It is not always possible to state categorically that a particular error was caused by a punching error rather than misuse of an identifier. We choose the most likely explanation in these cases.

- 3.1 Ordering Errors. A large number of errors are caused by incorrect sequencing of an ordering list of items.
 - 3.1.1 Incorrect order of array subscripts or procedure parameters.
 - 3.1.2 Incorrect order of array subscripts in sliced array. Algol W allows a slice of an array to be passed as a parameter.
 - 3.1.3 Incorrect ordering of case statement components. The case statement components in Algol W are unlabelled.
- 3.2 Omission Errors. Most errors of omission are detected at compile time, but a few are not.
 - 3.2.1 Omission of case statement component. This is undetected in Algol W because components are unlabelled.
 - 3.2.2 Failure to increment the value of a variable in a loop.
 - 3.2.3 Semi-colon omitted after a comment, causing the subsequent statement to be ignored.
 - 3.2.4 Semi-colon omitted after end-comment, causing the subsequent statement to be ignored (in Algol 60 only).
 - 3.2.5 Omission of string quotes causing confusion between string literal and identifier consisting of the same characters.
- 3.3 Logic Errors.
 - 3.3.1 Type error, undetected because of coercion (for example assignment of real to integer variable in Algol 60).

- 3.3.2 Failure to initialise a variable
- 3.3.3 Use of wrong identifier (for example, in nested for-statements, use of control variable I where J should be used).
- 3.3.4 Incorrect statement order (for example misplaced end of compound statement).
- 3.3.5 Misuse of goto-statement.
- 3.3.6 Other logic errors. With the source of information available, we were unable to break down this group further. It includes errors such as
- incorrect data
 - failure to consider exception conditions
 - use of wrong operator (e.g. "<" instead of "≤")
 - errors for which we can find no particular pattern
 - and others

3.4 Scope Errors

- 3.4.1 Reference to control variable whose value is undefined. This error is prevented in Algol W.
- 3.4.2 Identifier redeclared in inner scope, preventing non local access or access to parameter of the same name.

3.5 Parameter passing errors.

- 3.5.1 Value call where name is required, resulting in failure to update non local value.
- 3.5.2 Value-result call where name is required (Algol W only). For example, we suppose that variable V is passed by value-result to procedure X. Procedure X updates the value of V and calls

procedure Y. If procedure Y accesses V non locally, it receives the original and not the updated value of V.

3.5.3 Misuse of side effects

3.6 Punching and formulation errors

3.6.1 Mispunching of 1-letter identifiers. The risk of non-detection because of confusion with another 1-letter identifier appears significant.

3.6.2 Failure to remove corrected cards.

Conclusions

In considering security, we are particularly concerned with those errors which are either undetected or detected, but poorly diagnosed.

Algol W prevents some of the errors which may go undetected in Algol 60 (error types 3.2.4, 3.4.1), but some others are introduced (error type 2.2.1, 3.1.2, 3.5.2).

A brief examination of the errors suggests that approximately 50% of the undetected errors could be prevented by language design (cf. section 3.2.3 and 3.3.2). We would expect also that a considerable number of the remaining errors would be detected by the use of assertions and invariants.

FIGURE G-1

Characteristic Errors in Sample of Algol W
Programs (Stanford Compiler), Sample of Algol
60 Programs (Delft Compiler).

Error Category	Frequency of Error	
	Algol W	Algol 60
1. Compile-time detected errors	(283)	(294)
1.1 Context sensitive	(90)	(88)
1.1.1 Undeclared variable identifiers	51	64
1.1.2 Mis-spelled identifiers	15	9
1.1.3 Type incompatibility	19	7
1.1.4 Incorrect number of subscripts	5	8
1.2 Scope errors	(7)	(15)
1.2.1 Reference to label in inner block	-	6
1.2.2 Identifier redeclared in same scope	7	9
1.3 Omission errors	(75)	(49)
1.3.1 Mandatory space omitted	13	0
1.3.2 Comma omitted	15	10
1.3.3 Semi-colon omitted	29	37
1.3.4 String quote omitted	19	-
1.3.5 Array identifier omitted from array designator	0	2
1.4 Simple syntax errors	(111)	(142)
1.4.1 Mismatched brackets (including <u>begin-end</u> pairs)	37	29
1.4.2 Other syntax errors	74	113
2. Run-time detected errors	(76)	(69)
2.1 Type incompatibility	8	3
2.2 Over-reading input file	(28)	(14)

Error Category	Frequency of Error	
	Algol W	Algol 60
2.2.1 Read/readon errors	11	-
2.2.2 Others	17	14
2.3 Array subscript out of range	16	29
2.4 Case/switch expression out of range	1	1
2.5 Overflow	3	5
2.6 Time or page limit exceeded	11	14
2.7 Assert failure	9	-
2.8 Standard function error	0	3
3. Undetected errors	(102)	(122)
3.1 Ordering errors	(14)	(5)
3.1.1 Incorrect order of array subscripts or procedure parameters	8	5
3.1.2 Incorrect order of array subscripts in sliced array	4	-
3.1.3 Incorrect ordering of case components	2	-
3.2 Omission errors	(12)	(9)
3.2.1 Omission of case component	2	-
3.2.2 Variable increment omitted	3	2
3.2.3 Semi-colon omitted after comment	6	4
3.2.4 Semi-colon omitted after end-comment	-	3
3.2.5 Omission of string quotes	1	-
3.3 Logic Errors	(66)	(91)
3.3.1 Incorrect variable type	0	5
3.3.2 Failure to initialise a variable	13	17
3.3.3 Use of wrong identifier	12	9

Error category	Frequency of Error	
	Algol W	Algol 60
3.3.4 Incorrect statement order	8	13
3.3.5 Misuse of goto statement	0	6
3.3.6 Other logic errors	33	41
3.4 Scope Errors	(1)	(7)
3.4.1 Reference to control variable whose value is undefined	-	4
3.4.2 Identifier redeclared in inner scope	1	3
3.5 Parameter passing errors	(3)	(2)
3.5.1 Value call where name is required	1	1
3.5.2 Value-result call where name is required	2	-
3.5.3 Misuse of side-effects	0	1
3.6 Punching and formulation errors	(6)	(8)
3.6.1 Mispunching of 1-letter identifiers or 1 digit numbers	2	4
3.6.2 Failure to remove corrected cards	4	4
Total number of errors	(461)	(485)

APPENDIX H CONFLICTS OF BASE LANGUAGE DESIGN CRITERIA

In this appendix our intention is to determine which design criteria are highly conflicting, and which are relatively similar or independent of each other. We compare each criterion with each other criterion, in turn, and consider how they are related on the following scale:

- ✓ ⇒ the 2 criteria are well correlated or similar.
- (✓) ⇒ " " " are similar in some respects,
independent in others.
- 0 ⇒ " " " are relatively independent.
- (X) ⇒ " " " are dissimilar in some respects,
independent in others.
- X ⇒ " " " are poorly correlated or dissimilar.

The results are presented in matrix form in figure H-1.

We conclude that the criteria of involution and orthogonality are in fact the same, since their entries are identical.

We therefore delete one row and one column from the matrix.

We correct this matrix for inconsistencies by comparing similarities between one row (column) and each other row. We thus quantify correlation by considering two such rows element by element, adding 2 to the "correlation" if the elements are the same, and 1 if the elements differ by one position only on the scale ✓ (✓) 0 (X) X. Thus, for Security-Extensibility we consider the rows:

Security (✓) 0 0 X ✓ 0 ✓ 0 0

Extens-
ibility 0 0 X ✓ (✓) ✓ 0 (✓) ✓

Value
Assigned 1 2 0 0 1 0 0 1 0 Total = 5

	Modesty	Elegance	Efficiency	Generality	CSC	Orthogonality	Involution	Security	Portability	Extensibility
Modesty	✓	(✓)	0	X	X	✓	✓	(✓)	✓	0
Elegance	(✓)	✓	0	✓	X	✓	✓	0	0	0
Efficiency	0	0	✓	X	✓	0	0	0	(✓)	X
Generality	X	✓	X	✓	X	✓	✓	X	X	✓
CSC	X	X	✓	X	✓	X	X	✓	(✓)	(✓)
Orthogonality	✓	✓	0	✓	X	✓	✓	0	✓	✓
Involution	✓	✓	0	✓	X	✓	✓	0	✓	✓
Security	(✓)	0	0	X	✓	0	0	✓	0	0
Portability	✓	0	(✓)	X	(✓)	✓	✓	0	✓	(✓)
Extensibility	0	0	X	✓	(✓)	✓	✓	0	(✓)	✓

FIGURE H-1

The results of this process are shown in figure H-2.

We re-construct the original matrix (figure H-1) from figure H-2, using a "best fit". The reconstructed matrix cf. figure H-3 is not wildly different from the original. Since the matrix is symmetrical about the diagonal, only half is reconstructed. This process is in fact carried out iteratively correcting the initial matrix for inconsistencies. The illustrated matrices represent the final iteration.

	Modesty	Elegance	Efficiency	Generality	CSC	Orthogonality	Security	Portability	Extensibility
Modesty	20	11	5	5	5	12	9	12	5
Elegance	11	20	4	8	2	13	8	7	8
Efficiency	5	4	20	0	8	3	10	9	9
Generality	5	8	0	20	3	10	0	3	8
CSC	5	2	8	3	20	2	8	7	4
Orthogonality	12	13	3	10	2	20	3	10	8
Security	9	8	10	0	8	3	20	8	5
Portability	12	7	9	3	7	10	8	20	10
Extensibility	5	8	9	8	4	8	5	10	20

FIGURE H-2

	Modesty	Elegance	Efficiency	Generality	CSC	Orthogonality	Security	Portability	Extensibility
Modesty									
Elegance	✓								
Efficiency	0	(X)							
Generality	0	(✓)	X						
CSC	0	(X)	(✓)	X					
Orthogonality	✓	✓	X	✓	X				
Security	(✓)	(✓)	✓	X	(✓)	X			
Portability	✓	0	(✓)	X	0	✓	(✓)		
Extensibility	0	(✓)	(✓)	(✓)	(X)	(✓)	0	✓	

FIGURE H-3

BIBLIOGRAPHY

- (And 71) Anderson T., Eve J., Horning J.J. Efficient LR(1) parsers TR24, Univ. of Newcastle, 1971.
- (Ard 69) Arden B.W., Galler B.A., Graham R.M. The MAD definition facility Comm. ACM 10, 1969.
- (Ash 65) Ash R. Information theory Interscience Tracts no. 19, Wiley 1965.
- (Bar 72) Barron D.W., Jackson I.R. Evolution of job control languages Softw. P. and E. 2, 2 1972.
- (Bar 74) Barron D.W. Job control languages and job control programs Comp. J. 17, 3 1974.
- (Bau 71) Bauer H.R. et al. Algol W reference manual - Stanford Univ., 1971.
- (Bau 73) Bauer F.L., Beckmann M. (ed) Advanced course on software engineering Springer-Verlag, 1973.
- (Bcs 74) British Computer Soc. Job control languages - past, present, future BCS Symp., London, 1974.
- (Bec 75) Beckman A. Secondary Effects SIGPLAN Notices 10, 2 (Feb 1975).
- (Bel 69) Bell J.R. Transformations: the extension facility of Proteus SIGPLAN notices 4, 8 (Aug 1969), Proc. Int. Symp. on Extensible Languages.
- (Bob 68) Bobrow D.G. (ed) Symbol manipulation languages and techniques Proc. IFIP Congress, 1966.
- (Boc 73) Bochman G.V. Multiple exits from a loop without the goto Comm. ACM 16, 7 1973.
- (Bos 73) Bosch R. et al. ALEPH Proc. Int. Comp. Symp. 1973 (ed. Davos).
- (Bow 71) Bowlden H.J. Macros in higher level languages SIGPLAN Notices 6, 12 (Dec. 1971), Proc. Int. Symp. on Extensible Languages.
- (Bro 67) Brown P.J. ML/1 Text Manipulation language Comm. ACM 10, 10 1967.
- (Bro 71) Brown P.J. Survey of macro processors Annual Rev. in autom. Prog. 6, 1971.
- (Broo 63) Brooker R.A. et al. The compiler compiler Annual Rev. in autom. Prog 3, 1963.
- (Car 66) Carraciolo F. et al. Panon 1-B: A programming language for symbol manipulation SICSAM Symposium, Washington, 1966.

- (Che 66) Cheatham T.E. Introduction of definitional facilities into high level languages Proc. AFIPS 1966 FJCC Vol. 29.
- (Che 68) Cheatham T.E. et al. On the basis for ELF - an extensible language facility Proc. AFIPS 1968 FJCC.
- (Che 69) Cheatham T.E. Motivation for extensible languages SIGPLAN Notices 4,8 (Aug. 1969), Proc. Int. Symp. on Extensible Languages.
- (Chr 65) Christensen C. Ambit Proc. ACM 20th Nat. Conf., 1965.
- (Chr 69) Christensen C., Shaw C.J. (ed) Proceedings of the International Symposium on Extensible Languages SIGPLAN Notices 4,8 (Aug) 1969.
- (Coh 65) Cohen K., Wegstein J.H. Axle: An axiomatic language for string transformations Comm. ACM 8,11 1965.
- (Con 63) Conway M.E. Design of a separable transition diagram compiler Comm. ACM 6,7 1963.
- (Dah 72) Dahl O.J., Dijkstra E.W., Hoare C.A.R. Structured programming, Academic Press, 1972.
- (Dak 72) Dakin R.J. Towards a general control language Internal Report, Culham Lab., 1972.
- (Daw 73) Dawson J.L. Combining interpretive code with machine code Comp. J. 16,3 1973.
- (Dij 68) Dijkstra E.W. Goto statement considered harmful Comm. ACM 11, 1968.
- (Dij 72) Dijkstra E.W. Humble Programmer Comm. ACM 15,10 1972.
- (Dij 74) Dijkstra E.W. Axiomatic definition of semantics of deterministic programming languages Advanced Course on Computer Systems Architecture, Grenoble, 1974.
- (Dub 71) Duby J.J. Extensible languages: The users' point of view SIGPLAN Notices 4,8 (Aug. 1969), Proc. Int. Symp. on Extensible Languages.
- (Ear 70) Earley J., Sturgis H. Formalism for translator interactions Comm. ACM 13,10 1970.
- (Els 73) Elson M. Concepts of Programming Languages SRA, 1973.
- (Eng 71) Engeler E. (ed) Symposium on Semantics of Algorithmic languages Springer-Verlag, 1971.
- (Fel 64) Feldman J.A. Formal semantics for computer oriented languages Ph.D. thesis, Carnegie-

Mellon Inst., 1964.

- (Fel 68) Feldman J., Gries D. Translator writing systems Comm. ACM 11,2 1968.
- (Fisd 70) Fisher D.A. Control Structures for programming languages Ph.D. thesis, Carnegie-Mellon Inst., 1970.
- (Fisr 73) Fisher R.N. Snap - A user extensible high level language Seminar, St. Andrews, 1973.
- (Fism 74) Fisher M.J., Patterson M.S. String matching and other products MAC Memorandum 41, 1974.
- (Flo 67) Floyd R.W. Non-deterministic algorithms J. ACM 14,1 1967.
- (Flo 67b) Floyd R.W. Assigning meanings to programs Proc. Symp. in applied mathematics Vol. 19, American Math. Soc. 1967.
- (Flo 70) Floyd R.W., Knuth D.E. Notes on avoiding goto statements TR Stanford, 1970.
- (Fri 74) Friedman D.P., Shapiro S.C. A case for "while-until" SIGPLAN Notices 9,7 1974.
- (Gal 67) Galler B.A., Perlis A.J. Proposals for definitions in algol Comm ACM 10, 1967.
- (Gal 74) Galler B.A. Extensible languages Proc. IFIP Congress, Stockholm, 1974.
- (Gan 75) Gannon J.D., Horning J.J. Impact of language design on the production of reliable software TR CSRG-45, Univ. of Toronto, Dec. 1974.
- (Gar 68) Garwick J.V. GPL, a general purpose language Comm. ACM 11,9 1968.
- (Gim 73) Gimpel J.F. A theory of discrete patterns and implementation in Snobol 4 Comm. ACM 16,2 1973.
- (Gor 61) Gorn S. Some basic terminology connected with mechanical languages and their processors Comm. ACM 4,8 1961.
- (Gra 71) Grant C.A. Syntax translation with context macros SIGPLAN Notices 6,12 (Dec. 1971), Proc. Int. Symp. on Extensible Languages.
- (Grie 71) Gries D. Compiler construction for digital computers Wiley, 1971.
- (Grif 74) Griffiths M. LL(1) grammars and analysers Advanced Course on Compiler Construction, Munich, 1974.

- (Gris 71) Griswold R.E., Poage I.F., Polansky I.P. The Snobol 4 programming language. Prentice-Hall, 1971.
- (Gris 72) Griswold R.E. The macro implementation of Snobol 4. Freeman and Company, San Francisco, 1972.
- (Gris 74) Griswold R.E. Suggested revisions to syntax and control mechanisms of Snobol 4. SIGPLAN Notices 9,2 1974.
- (Gut 75) Gutttag J. Annotated bibliography on computer program engineering. TR CSRG-54, University of Toronto, April 1975.
- (Hal 64) Halpern M.I. XPOP: A metalanguage without metaphysics. Proc. AFIPS 1964 Vol. 26.
- (Hal 68) Halpern M.I. Towards a general processor for programming languages. Comm. ACM 11,1 1968.
- (Hen 72) Henderson P., Snowden R. An experiment in structured programming. BIT 12, 1972.
- (Hoa 66) Hoare C.A.R., Wirth N. Contribution to the development of Algol 60. Comm. ACM 9,6 1966.
- (Hoa 73) Hoare C.A.R. Hints on programming language design. TR Stanford Univ., Calif., 1973.
- (Hoa 73b) Hoare C.A.R. Recursive data structures. A.I. Lab., Stanford, Oct. 1973.
- (Hoa 75) Hoare C.A.R. Data reliability. SIGPLAN Notices 10,6 (Jun 1975), Int. Conf. on Reliable Software.
- (Hop 69) Hopcraft J.E., Ullman J.D. Formal languages and their relation to automata. Addison-Wesley, 1969.
- (Ibr 74) Ibrahim S.Z.M. Abstract machines for programming languages. Ph.D. thesis, Univ. of Glasgow, 1974.
- (Ich 74) Ichbiah J.D. LIS, a system implementation language. Advanced Course on Computer Systems Architecture, Grenoble, 1974.
- (Ill 69) Iliffe J.K. Elements of BLM. Comp. J. 12,2 1969.
- (Iro 70) Irons E.T. Experience with an extensible language. Comm. ACM 13, 1 1970.
- (Jor 71) Jorrand P. Data types and extensible languages. SIGPLAN Notices 6,12 (Dec. 1971), Proc. Int. Symp. on Extensible Languages.

- (Jen 74) Jensen K., Wirth N. PASCAL User manual and report Springer Verlag, 1974.
- (Kat 70) Katzan A.N.H. Advanced programming Van Nostrand, 1970.
- (Knud 74) Knudsen M.J. PMSL - Interactive language for system level description Ph.D. thesis, Carnegie-Mellon Univ. Pitts., 1974.
- (Knut 67) Knuth D.E. Top down syntax analysis Acta Inf. 1,2 1971.
- (Knut 73) Knuth D.E. Expirical study of Fortran programs C.S. Dept. TR Stanford Univ., Calif., 1973.
- (Knut 74) Knuth D.E. Structured programming with GO TO statements C.S. Dept. TR, May 1974.
- (Kos 71) Koster C.H.A. Affix Grammars Algol 68 Implementation (ed. Peck J.E.L.), North-Holland, 1971.
- (Lea 66) Leavenworth B.M. Syntax macros and extended translation Comm. ACM 9,11 1966.
- (Lea 72) Leavenworth B.M. Programming with(out) the GOTO Proc. ACM 1972 Annual Conference, 1972.
- (Led 69) Ledgard H.F. A formal notion for defining the syntax and semantics of computer languages Ph.D. thesis, MIT, Mass., 1969.
- (Led 71) Ledgard H.F. Ten mini languages C. Survey 3, 1971.
- (Les 71) Lester B.P. Cost analysis of debugging systems Ph.D. thesis, MIT, Mass., 1971.
- (Lew 68) Lewis P.M., Stearns R.E. Syntax-Directed transduction. J.ACM 15,3 1968.
- (Lis 74) Liskov B.E., Zilles S. Programming with abstract data types SIGPLAN Notices 9,4 (April 1974).
- (Lis 75) Liskov B.H., Zilles S. Specification techniques for data abstractions SIGPLAN Notices 10,6 (April 1975).
- (McI 60) McIlroy M.D. Macro instruction extension of compiler languages Comm. ACM 3,4 1960.
- (McK 70) McKeeman W.M., Horning J.J., Wortmann D.B. A compiler generator Prentice-Hall, 1970.
- (McK 74) McKeeman W.M. Programming language design Advanced Course on Compiler Construction (ed. Bauer F.L.), Munich, 1974.

- (McK 74b) McKeeman W.M. Compiler construction Advanced Course on Compiler Construction (ed. Bauer, F.L.), Munich, 1974.
- (Mane 73) Marneffe P.A., Ribbens D. Holon Programming Proc. Int. Comp. Symp. 1973 (Davos ed)..
- (Mann 68) Manna Z. Formalisation of properties of programs TR A.I. Lab., Stanford, July 1968.
- (Milg 71) Milgram E. AEPL - an extensible programming language SIGPLAN Notices 6,12 (Dec. 1971), Proc. Int. Symp. on Extensible Languages.
- (Mit 70) Mitchell J.G. Design and construction of flexible and efficient interactive programming systems Ph.D. thesis, Carnegie-Mellon Univ., Pitts., 1970.
- (Moo 66) Mooers C.N. TRAC, a procedure describing language for the reactive type-writer Comm. ACM 9,3 1966.
- (Nap 67) Napper R.B.E. Some proposals for Snap, a high level language with macro facilities Comp. J. 10, 1967.
- (Nau 62) Naur P. (ed) Revised report on the algorithmic language Algol 60.
- (New 68) Newey M.C. An efficient system for user extensible languages Proc. AFIPS 1968 FJCC 33,2. ...
- (Nie 70) Nievergelt J., Irland M.I. Bounce-and-skip Comp. J. 13,3 1970.
- (Pal 74) Palme J. Simula as a tool for extensible program products SIGPLAN Notices 9,2 1974.
- (Par 72) Parnas D.L., Siewiorek D.P. Use of the concept of transparency in the design of hierarchically structured systems TR. Carnegie-Mellon Univ., Pitts., 1972.
- (Pat 72) Patterson J.W. ALGOL W Abstract Machine, TR., Computing Science Dept., University of Glasgow, 1974.
- (Pat 75) Patterson J.W. Abstract Pascal Machine, Working Document, Univ. of Glasgow, 1975.
- (Per 67) Perlis A.J. The synthesis of algorithmic systems J. ACM 14,1 1967.
- (Pet 73) Petersen W.W. et al. On the capabilities of while, repeat and exit statements Comm. ACM 16,8 1973.
- (Pir 75) Pirie I.G. Measurement of Programming Ability Ph.D. thesis, University of Glasgow, 1975.

- (Poo 73) Poole P.C., Waite W.M. Portability and adaptability Advanced Course on Software Engineering (ed Bauer F.L.), Springer-Verlag, 1973.
- (Poo 74) Poole P.C. Portable and adaptable compilers Advanced Course on Compiler Construction (ed. Bauer F.L.), Munich, 1974.
- (Rat 66) Rathbone R.R. Communicating Technical Information Addison-Wesley, 1966.
- (Rem 71) De Remer F.L. Simple LR(k) grammars Comm. ACM 14,7 1971.
- (Ross 74) Rossman Functional Memory-based dynamic microprocessors for high level languages SIGPLAN Notices 9,8 1974.
- (Sam 69) Sammet J.E. Programming Languages: History and Fundamentals, Prentice-Hall, N.J., 1969.
- (Sch 70) Schuman S.A., Jorrand P. Definitional mechanisms in extensible programming languages Proc. AFIPS 1970 Vol. 37.
- (Sch 71a) Schuman S.A. (ed), Proc. International Symposium on Extensible Languages SIGPLAN Notices 6,12 Grenoble, 1971.
- (Sch 71b) Schuman S.A. An extensible interpreter SIGPLAN Notices 6,12 (Dec. 1971), Proc. Int. Symp. on Extensible Languages.
- (Sco 71) Scowen R.S. Babel and Soap: an application of extensible compilers Softw. P. and E. 3,1 1973.
- (Sco 73) Scowen R.S. Wichmann B.A. Definition of comments in programming languages NPL Report NAC 34, May 1973.
- (Sha 71) Shaw M. Language structures for contractible compilers Ph.D. thesis, Carnegie-Mellon Univ., Pitts., 1971.
- (Sim 73) Sime M.E. et al. Psychological evaluation of two conditional constructs used in computer languages Int. Journal Man-Machine Studies 5,1 1973.
- (Sol 74) Solntseff N. Yezeriski A. A survey of extensible programming languages Annual Review in Automatic Programming 7, Pergamon Press, London, 1974.
- (Sta 69) Standish T.A. Some features of PPL, a polymorphic programming language SIGPLAN Notices 4,8 (Aug. 1969), Proc. Int. Symp. on Extensible Languages.

- (Stee 61) Steel T.B. UNCOL: the myth and the fact
Annual Review in Automatic Programming 7,
Pergamon Press, London, 1961.
- (Stew 74) Stewart G.F. An algebraic model for string
processors Ph.D. thesis, Univ. of Toronto,
1974.
- (Str 65) Strachey C. A general purpose macro generator
Comp. J. 8, 1965.
- (Tsi 73) Tsichritzis D. Software Reliability Infor.
11,2 1973.
- (Wai 67) Waite W.M. A language independent macro-
processor Comm. ACM 10,7 1967
- (Wai 70) Waite W.M. The mobile programming system:
Stage 2 Comm. ACM 13,7 1970.
- (Wai 73) Waite W.M. Implementing software for non-
numeric applications Prentice-Hall, 1973.
- (Wat 74) Watt D.A. Analysis-oriented two level grammars
Ph.D. thesis, Univ. of Glasgow, 1974.
- (Web 66) Weber H., Wirth N. Euler: a generalisation of
Algol and its formal definition Comm. ACM 9,
1966.
- (Weg 68) Wegner P. Programming Languages, Information
Structures and Machine Organisation McGraw-
Hill, 1968.
- (Wein 71) Weinberg G.M. Psychology of Computer Programming
Van Nostrand, 1971.
- (Wein 75) Weinberg G.M. et al. IF-THEN-ELSE considered
harmful SIGPLAN Notices 10,8 (Aug. 1975).
- (Weis 74) Weissman L.M. A methodology for studying the
psychological complexity of computer programs
CSRG 37, Univ. of Toronto, 1974.
- (Wic 73) Wichmann B.A. Basic statement times for Algol
60 TR NPL, Teddington, 1973.
- (Wij 69) Wijngaarden A. Report on the algorithmic
language Algol 68 Num. Maths 14, 1969.
- (Wir 70) Wirth N. The programming language Pascal Acta
Inf. 1, 1971.
- (Wir 71) Wirth N. The design of a Pascal compiler
Softw. P. and E. 1, 1971.
- (Wir 72) Wirth N., Programming language Pascal (Revised
Report) 1972.

- (Wir 74) Wirth N. Programming Language Design Proc. IFIP Congress, Stockholm, 1974.
- (Wir 75) Wirth N. Assessment of the programming language Pascal SIGPLAN Notices 10,6 (Jun 1975).
- (Woo 71) Woolley J.D. Menelaus: a system for measurement of the extensible language system, Proteus SIGPLAN Notices 6,12 (Dec. 1971), Proc. Int. Symp. on Extensible Languages.
- (Wor 74) Wortman D.B. Notes from a workshop on the attainment of reliable software, TR CSRG 41, Univ. of Toronto, 1974.
- (Wul 70) Wulf W.A. et al. Bliss reference manual TR, Carnegie-Mellon Univ., Pitts., 1970.
- (Wul 71) Wulf W.A. Programming without the GOTO Proc. IFIP Congress Vol. 1, 1971.
- (Wul 73) Wulf W.A., Shaw M. Global variables considered harmful SIGPLAN Notices 8,2 (Feb. 1973).
- (Yng 63) Yngve V.H. Comit as an IR language Comm. ACM 5, 1963.
- (You 74) Youngs E.A. Human errors in programming Int. Journal Man-machine Studies 6,3 (May 1974).