ABSTRACT

This thesis describes an implementation of a syntax-directed translator originally due to C.M. Reeves. The translator program simulates a special-purpose stored-program digital computer. A program for this machine represents the syntax and semantics of some language - the source language. The translator is program-driven, backtracking and performs the parsing of the source text in a top-down manner. It does not allow left-recursive grammars. The program is written in ALGOL W on the IBM 360/44 computer.

The translator accepts language specifications (syntax and semantics) in extended BNF - a meta language described in the thesis - and eventually produces a recogniser for statements in the source language.

The translator is composed of two machines - a parsing machine and an editor machine. The parsing machine has 17 machine instructions, some of these instructions are of single address type while others have no operand. Details of these are described. Briefly they are used by the parser machine to make decisions about whether or not a source text is grammatical. The output produced by the parser machine is determined by the semantics embodied in the program. This output is then carried to the editor machine and is treated as its input.

The input to the editor machine is an ordered sequence of symbols which contains some edit instructions. With the help of these edit codes, this input is interpreted on the editor machine. The editor machine has 6 machine instructions which normally operate on the top one or two elements of the editor stack. The output produced by the editor machine is then assembled. The program thus obtained can be run on the hypothetical computer as a compiler.

A summary of related work is given.

AN IMPLEMENTATION OF
REEVES' SYNTAX DIRECTED TRANSLATOR

being a thesis

presented by

Ambrish Vashishta

to the

UNIVERSITY OF ST ANDREWS

in application for

THE DEGREE OF MASTER OF SCIENCE

St. Andrews
Scotland

July, 1978

DECLARATION

I declare that the study and research for the subject matter for this thesis has been carried out by myself, that this thesis is my own composition, and that it has not previously been presented in application for a higher degree.

The work for this project was done in the Department of Computational Science of the University of St Andrews under the direction of Mr A.J.T. Davie.

Ambrish Vashishta

CHAPTER 1

INTRODUCTION

ACKNOWLEDGEMENTS

I sincerely thank Mr A.J.T. Davie for providing the topic of research and for his guidance and constant encouragement throughout this work.

I am indebted to Professor Cole for providing me with an opportunity to work with Mr Davie and to the University of St Andrews for the award of a postgraduate scholarship and for tuition.

I would like to thank the academic staff and my fellow research students for their friendly assistance, Dr R. Erskine and other members of the Computing Laboratory Staff for providing the facilities of the IBM 360/44 computer and helping me with the running of my programs in some way, and Miss Seonaid McCallum for carefully typing this thesis.

Finally, I wish to thank my brother Dr P. Vashishta for his generosity for providing me with travelling expenses.

A. VASHISHTA

C O N T E N T S

1.1 Description

This thesis describes an implementation of a Syntax-director translator originally due to C.M. Reeves (Ree-67). In order to describe the implementation and for ease of understanding in this chapter we shall define some basic terminology which we shall use in describing compilers; this will include the definition of compilers, syntax, semantics (of programming languages) and a brief description of the phases of compilation: and finally a description of the layout of the rest of the thesis.

1.2 Definitions

A translator is a program which translates a source language program into an equivalent object program. The Source program is written in source language, the object program is a string of the object language. If the source language is a high-level language like FORTRAN, ALGOL or COBOL, and if the object language is the assembly language or machine language of the computer, the translator is called a compiler. Machine language is sometimes called code; hence the object program is sometimes called object code. The translation of the source program occurs at compile time; the actual execution of the object program at run time (Gri-71).

An interpreter for a source language accepts a source program written in that language as input and executes it. The difference between a compiler and an interpreter is that the interpreter does not produce an object program to be executed; it executes (or interprets) source program itself (or an intermediate object code program if compilation and interpretation are both involved (Gri-71).

A compiler must perform an analysis of the source program and a synthesis of the object program. It first analyses the source program into its basic parts; and then builds equivalent object program parts from them. The decomposition involves syntax analysis of the structure of the source program and then the rebuilding consists of a series of semantics actions guided by the analysis. In order to do this the compiler builds several tables during the analysis phase which are used during analysis and synthesis. In a high-level language compilation - as a program is analysed, information is obtained from declarations, procedure headings, for-loops, and so forth, and saved for later use. This information is detected and collected so that we have access to it from all parts of the compiler. It is necessary to know with each use of an identifier how that identifier was declared and used elsewhere. Exactly what must be saved depends, of course, upon the source and object language, and upon how sophisticated the compiler is. But every compiler uses a symbol table in one way or another (Gri-71).

A source program in a source language is nothing more than a string of characters. A compiler ultimately converts this string of characters into another string - the object code. In this process, sub-processes are identified as following:

(i) Lexical analysis

(ii) Bookkeeping - including symbol table maintenance

(iii) Parsing or syntax analysis

(iv) Code generation or translation to intermediate code by executing semantic actions

(v) code optimization

1.3 Phases of Compilation

We shall describe these five phases of compilation briefly. These phases do not necessarily occur separately in an actual compiler. However, it is often convenient to partition a compiler into them conceptually in order to isolate the problems that are unique to that part of compilation process (Aho-72).

1.3.1 Lexical Analysis

The lexical analysis phase comes first. It is the job of a lexical analyser to group together certain characters into single entities which may be called tokens. What constitutes a token is implied by specification of the programming language. Certain languages have keywords such as BEGIN, END, DO, INTEGER, and so forth, which are treated as single entities. A string of one or more blanks is often treated as a single blank or simply ignored. Strings representing numerical constants are treated as single items. Identifiers used as names of variables, functions, procedures, labels and the like are another example of a single lexical unit in programming languages.

Thus the lexical analyser is a translator whose input is the string of symbols representing the source program and whose output is a stream of tokens. This output forms the input to the syntactic analyser.

1.3.2 Bookkeeping

As tokens are uncovered in the lexical analysis, information about certain tokens is collected and stored in one or more symbol tables. What

this bookkeeping information is depends upon the language; but, for instance, properties of identifiers such as their types may be gathered and noted.

1.3.3 Syntax and its analysis

Parsing or syntax analysis is the process in which the string of tokens produced by the lexical analysis is examined to determine whether it obeys certain structural conventions given explicitly in the definition of language (see below). From a set of syntactic rules it is possible to construct parsers automatically which make sure that a source program obeys syntactic structure defined by these syntactic rules. The conceptual output from the parser is a tree which represents the syntactic structure inherent in the source program.

Before describing parsing, it is necessary, therefore, to define some notation for describing syntax of a programming language. Most of the work described here derives from the work of Chomsky (Cho-56) in the field of natural languages. The standard method adopted for describing a language is a generating scheme whereby a procedure is given for producing all the legal sentences of a language.

1.3.3.1 Parsing

As we might recall from elementary school an English sentence might be defined as a subject followed by a predicate. The subject may be a noun phrase which might consist of an article followed by a noun. The predicate could be defined as a verb followed by an object. The object may be a noun phrase which again might consist of an article followed by a noun.

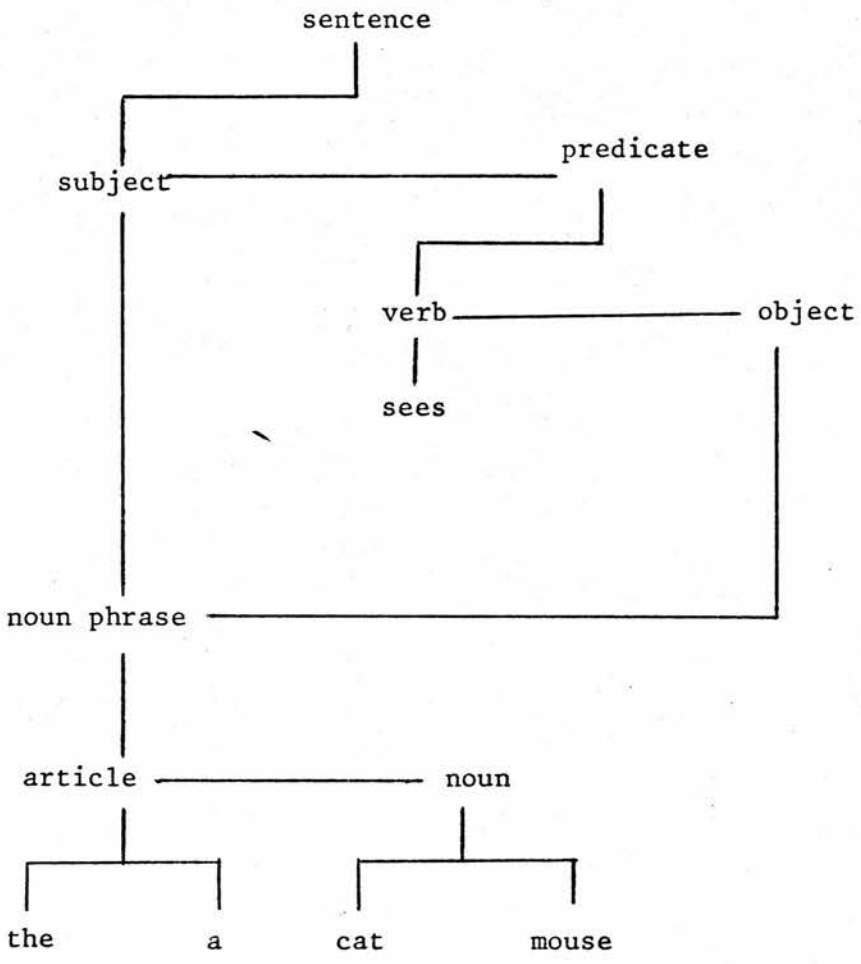
The terms "subject", "predicate", etc. are called syntactic constituents (or syntactic elements) of the sentence. A subset of a complete set of English sentence could be defined by allowing only the nouns 'cat', 'mouse', the verb 'sees' and the articles 'a' and 'the'. The words in inverted commas are English words which will appear in the English sentence produced and are therefore given name ultimate constituents, or basic (or terminal) symbols (or words).

An example of a notation adopted for describing these constructs is:

<sentence> → <subject><predicate>
<subject> → <noun phrase>
<noun phrase> → <article><noun>
<predicate> → <verb><object>
<object> → <noun phrase>
<verb> → sees
<article> → the, a
<noun> → cat, mouse

(Figure - 1.1) Grammar of an English Sentence

In this notation the syntactic elements are enclosed by '<' and '>'. The same idea can also be explained by a technique of diagramming. For example the grammar may be diagrammed as shown over in Figure - 1.2.



(Figure-1.2) - Grammar of English sentence in a tree fashion

Such a diagram displays the syntax (grammatical structure) of a sentence in a tree fashion. Each node represents a syntactic constituent of the syntax. The words 'the', 'eat', 'sees', 'a' and 'mouse' are the ultimate constituents (or terminal words) of the sentence. We shall discuss two methods of parsing such a sentence.

The parsing methods are goal-oriented. At each stage a goal is set-up. This will in turn involve finding some sub-goals. In top-down parsing subgoals are checked from left-to-right. The information that successful recognition has taken place is passed back to the goal which this is a sub-goal, and if all the sub-goals are found the goal itself is found.

1.3.3.2 Top-Down Parsing

Considering the tree shown in Figure 1.2 our main goal is <sentence>. <sentence> has two sub-goals, <subject> and <predicate>. The sub-goal <subject> itself has a, <noun phrase>, which further has two sub-goals <article> and <noun>. The second sub-goal of the main goal <sentence> also has two sub-goals <verb> and <object>. And <object> has one sub-goal, which further has two sub-goals <article> and <noun> as mentioned above. The sub-goal <article> may be satisfied by alternatively finding the goal 'a' or the goal 'the'.

If a goal (or a sub-goal) has more than one sub-goal, the sub-goals are attempted from left to right. For example, to find the main goal <sentence>, the sub-goal <subject>, which is to the left should be achieved first, and then the second sub-goal <predicate>, which is to the right. All such sub-goals must be achieved in the correct order. If a goal has alternative sub-goals only one of them need be satisfied. They are still, however, tried from left to right until one of them is recognised.

The top-down parsing method attempts initially to find a goal called the start symbol. In this case the start symbol is <sentence>. This leads us to look at the sub-goals necessary to achieve this main goal. These sub-goals will involve finding further sub-goals and so on. Eventually a sub-goal will require matching of the group of words (constituting the sentence to be parsed) with some terminal words.

In the example sentence 'the cat sees a mouse' the top-down process could be described as follows. The goal is to find out if this group of words is indeed a <sentence>. A <sentence> consists of a <subject> followed by a <predicate>. The intermediate sub-goals are therefore to find a <subject> and then a <predicate>. The <subject> consists of a <noun phrase> which has two intermediate sub-goals, an <article> and a <noun>. An <article> is either the word 'the' or the word 'a'. The group of terminal words starts with 'the', so that 'the' matches correctly in the given sentence. Will the next part of the group of words be a <noun>? A <noun> is either the word 'cat' or the word 'mouse'. The first of these appears next in the group of words, so both <article> and <noun> have been recognised and that is a <noun phrase> and hence <subject> is recognised.

Therefore <subject> could be matched in the group of words as 'the cat'. As <sentence >consists of a <subject> followed by a <predicate>, can <predicate> match the remainder 'sees a mouse'? The sub-goals of predicate would eventually be obtained in a similar manner and finally <sentence>, the original goal, would be recognised as <subject> followed by <predicate>.

The example above is so trivial that the only place that non-recognition takes place is at the level of terminal words. However, in general, non-recognition may occur at any level and alternatives tried if there are any.

1.3.3.3 Bottom-Up Parsing

Considering the same English sentence 'the cat sees a mouse' once again. Bottom-up parsing proceeds as follows. The goal is to produce a non-terminal <sentence> given a particular group of words. First consider the possible rules to see if a rule exists which starts with the word 'the', the first

word in the sentence to be parsed. There is only rule starting with the word 'the':

<article> → the, a

(See Figure - 1.1). So the sequential form now becomes

<article> cat sees a mouse

The only rule starting with <article> is <noun phrase>. However <noun phrase> is not yet completed as it needs a <noun> as well. Instead of continuing with the goal, <sentence>, this is now stored and a new goal <noun> is set up to match the group of words 'cat sees a mouse'. The word at hand is 'cat' and so the non-terminal <noun> is recognised and therefore <noun phrase>. At this point the original goal <sentence> is retrieved and by the rule

<subject> → <noun phrase>

we now proceed to the state:

<subject> sees a mouse

In order to achieve the goal <sentence>, the remainder 'sees a mouse' should be recognised as <predicate>.

Once again the parsing is done by starting from 'sees', the word at hand, and looking for a rule which starts with it. The only rule is

<verb> → sees

We now look for a rule starting with the <verb>. The only rule is

<predicate> → <verb><object>

As has been explained above the remainder of the group of words 'a mouse', can easily be recognised as <noun phrase>, which by the rule

<object> → <noun phrase>

is recognised as <object> and therefore 'sees a mouse' is recognised as <predicate>. And finally the group of words 'the cat sees a mouse' is recognised as <sentence> by the rule

<sentence> → <subject><predicate>

This is a simplified version of the bottom-up parsing.

1.3.4 Code Generation

We shall now go on to a brief description of code generation.

This phase of compilation produces object program. On a logical level the output of the syntax analyser is some representation of a parse tree. The code generatinn phase transforms this parse tree into an intermediate language representation such as an assembly language - either for a real machine or an ideal hypothetical one. Some compilers do not produce a parse tree but rather go to intermediate code directly as syntax analysis takes place.

As syntactic structures are recognised in the source program, codes are produced which are determined by the semantics. These semantics may be embedded in the syntax of the source language.

The nature of this part of the compiler depends on the source language, target machine and the quality (run time efficiency, compactness, etc) of object code desired.

1.3.5 Optimization

In many situations it is desirable to have compilers produce object program that runs efficiently. Code optimization is the term generally applied to attempt to make object program more efficient, e.g. fast running or more compact.

1.4 Errors

We have so far assumed that the input to the compiler is a well-formed program and that each phase of compiling can be carried out in a way that makes some sense. In practice, this will not be the case in many compilations. A compiler has an opportunity to detect errors in a program in at least three phases of compilation - lexical analysis, syntactic analysis, and code generation. When it is encountered, it is a difficult job, bordering on an application of "artificial intelligence", for the computer to be able to look at an arbitrary faulty program and tell what was probably meant. However, in certain cases, it is easy to make a good guess. Whether compilers should "correct" code after appropriate warning is a point of ethics which we shall not discuss here.

1.5 Semantics

The term syntax refers to a "relation" which associates with each sentence of a language a syntactic structure. The term semantic we shall take to refer to a 'mapping' which associates with the syntactic structure of each correct input a string in some language (possibly the same language) we consider the "meaning" of the original sentence (Pol-72).

1.6 The Compiler Type Considered in this Thesis

We shall consider a compiler model in which syntax analysis and code generation are combined into one single phase separate from the book-keeping. We can now view such a model as one in which code generation operations are interspersed with parsing operations. The book-keeping part if independent of description of the language which can be "plugged in" in one of the two ways described below. The term syntax-directed

compiling is often used to describe this kind of compilation (Aho-73).

1.7 Syntax-Directed Compilation

In practice, there are two methods of syntax-directed compilation: program-driven and table-driven. We shall explain them briefly in the following sub-sections.

1.7.1 Program-driven Syntax-directed Compiling

In this method a grammar-assembly-program is pre-constructed from the description of the language and is executed (either directly or by interpretation) to carry out the parsing and code generation. This stored form of the grammar is a list (or an array) whose elements are instructions for the parsing machine. Note that a second level of compilation (meta-compilation) has been introduced here. We have to compile the grammar description into the grammar-assembly-program.

1.7.2 Table-driven Syntax-directed Compilation

In this method a syntax-table is constructed. In this way the control mechanism is implemented by looking up this tabular form of the grammar. The analyser performs a syntactic analysis of the source material using this tabled specification as data (Che-64).

Both methods require a formal syntax and semantics of the language for the construction of the program or syntax-table.

1.8 Reeves' Translator

This thesis describes the implementation of a particular syntax-directed translator of the first type, described by Reeves (Ree-67). Briefly, the translator is program-driven and has a top-down back-tracking mechanism of parsing (we shall explain 'back-tracking' later). The implementation is done in ALGOL W on an IBM 360/44 computer.

1.9 Structure of the rest of the Thesis

Chapter 2 discusses some other syntax-and-table-directed translators which are in some way or other related to our specific problem. Chapter 3 describes the construction of a meta syntactic language. Chapter 4 describes a top-down back-tracking mechanism of parsing, for determination whether a source text in a language is grammatical. Chapter 5 describes the generation of output and adding of the semantic actions to the parser machine. It also describes an editor machine which is used as a post processor for manipulation of symbols. Chapter 6 describes how a recogniser can be produced automatically when the syntax of the language is specified in our meta language form.

In Chapter 7, a large problem has been tried namely - the translation of the source statements in the SASL language (Tur-76a) into SASL machine instructions (Tur-76b).

CHAPTER 2

SOME SYNTAX-AND TABLE-DIRECTED TRANSLATORS

2. In this chapter we shall describe some syntax-directed translators and compiler-compilers. Some of these are related to the work described in this thesis and others are discussed for completeness sake. But first we shall have a brief introduction to Translator Writing Systems and Compiler-Compilers.

2.1 Translator Writing Systems

A translator writing system (TWS) is any program or set of programs which aids in writing translators, compilers, interpreters, assemblers and the like. The purpose of a translator writing system is to simplify the implementation of translators and for this purpose a TWS contains primitives for the types of operations most translators must perform (Fel-68).

Much of the work in TWS has been specifically directly towards the problem of writing compilers, and the term compiler-compiler (Bro-62) (Bro-63) has been coined for this type. The term arose because a TWS designed for writing compilers is a program which compiles compilers (Gri-71).

Most TWS have the following two components in common:

1. A meta-syntactic language for describing the syntax of the source programs to be translated.

2. A meta-semantic language in which the semantic routines are written.

Syntax description and semantic routines are first compiled by the TWS into machine language or into some internal tabular form or language for interpretation. This process occurs at metacompile-time, that is, when meta descriptions of syntax and semantics of programs are compiled. The resulting translator is then executed or interpreted and this occurs at (ordinary) compile-time. Its execution is controlled by the syntax description and by the source program it is translating. Semantic routines are executed as the syntactic constructs with which they are associated are recognised. Alternatively two phases may be used - a syntactic analysis which builds a parse tree and a semantic synthesis which produces output from the tree.

A large part of a TWS is a set of basic subroutines, which any translator will automatically use. There is a library of Input/Output, and standard mathematical routines and functions used in algebraic languages.

The input to a TWS is usually BNF-like and for that input the TWS produces a parsing algorithm in the form of a program (possibly a tree) to be executed or interpreted, or else a set of tables which are used by basic routines in the library to perform the parse of source programs.

The semantic language in a TWS has the conventional constructions one has in other procedural languages and in general they are:

1. Data types, INTEGER, BOOLEAN, STRING
2. Simple variables, arrays, tables, records
3. Assignment, conditional, and iterative statements
4. I/O Statements

In addition, it may have other constructs such as pointers, arrays, stacks, primitives for searching tables and generating codes, etc.

Most TWS allow only single pass compilers (in which input is scanned only once and equivalent codes are generated), but there are some multipass ones like Cheatham's (Che-65).

2.2 Compiler-Compilers

A compiler-compiler (cc) is a program which is generally used for writing compilers. Historically, the existence of compiler-compilers is a result of using syntax-directed techniques in order to structure the compiler. The syntax of a language becomes a language itself in which parts of the compiler may be written, and the concept is extended to semantics by including a compatable programming language. Compiler-compilers very much depend on the language definition techniques used. It may even be said that a compiler-compiler is a tool which takes formal language definitions together with machine description and produces a compiler (Fel-68). A compiler-compiler is a TWS itself.

The first of a large number of compiler-compilers was described by Brooker and Morris (Bro-62). Most of them have certain characteristics:

(i) They are based on a particular method of syntax analysis - e.g. LR(1), LL(1), precedence etc.

(ii) Calls may be made during syntax analysis to semantic functions, which are procedures in some programming language. These calls appear in tables used by the analyser, or in the analysis program produced by the compiler-compiler.

(iii) Some compiler-compilers use a specially created production language (for example, that of Floyd and Evans' (Flo-61) and Feldman's FSL (Fel-66) see Section 2.6).

Whatever analysis method (top-down or bottom-up) is used, a compiler-compiler often includes a certain number of sub-programs which will perform the following tasks:

(a) Test that the grammar is well-defined and clean.

(b) If the grammar does not conform to type try to transform it so that it does.

(c) Produce an analyser in the relevent form, together with the interface which provides the "hooks" for the semantics.

2.3 A syntax-directed Translator for ALGOL 60 (Iro-61)

This syntax-directed translator described by E.T. Irons is one of the earliest translators developed. Irons' meta-language is an extension of the meta-syntactic language used in the ALGOL 60 report (Nau-60). The meta-language defines input and output of the source language and target language explicitly. Following is a description of the meta-language.

2.3.1. Irons' Meta-language

The translator specification consists of a series of sentences (grammatical rules) each one consisting of a syntax formula followed by a string of symbols designating semantics of that syntax formula. The meta-language defines a language A in terms of another, B, by listing a series of "definitions", each definition listing:

(a) A string of syntactic units in language A

(b) One syntactic unit of A which is the syntactic class being described.

(c) The meaning, described in language B, associated with the syntactic structure or, if meaning has been assigned to one or more syntactic units of (a), modification or amplification of these meanings.

Each sentence of the "specifications" has the following form:

SSS .... SS =:: S → {PPPP .... P}

components subject definition

where S is a syntax unit: either a metalinguistic variable (non-terminal) or a symbol of the input language; and P denotes a semantic unit: either a symbol of the output language or a designator of a string of such symbols.

The syntactic unit S following the meta symbol "=::" in any sentence is the "subject" of the sentence. The units to the left of the meta symbol "=::" are the "components" of the sentence. The string PPPP .... P between metasymbols { and } is the "definition" of the sentence. Specifically P may have one of the following forms:

1. Any symbol, p, of the output language. The output language alphabet may contain any symbol, but when the alphabet contains the symbols

$$\{P_n \; £ \; [\;] \; , \; \}$$

special conventions will hold in the cases described below.

2. An output string designator of the form

$$P_n \; [p \leftarrow PP \; .... \; P \; , \; p \leftarrow PP \; .... \; P \; , \; .... \; , \; p \leftarrow PP \; .... \; P]$$

where P and p are defined as above and n is an integer designating one particular string on the left of"=::" . The output designator string may be empty.

If a string designator is of the form

$$P_n [p \leftarrow PPP \; .... \; P \; , \; ....]$$

it denotes the same string but with substitutions made as indicated: namely, with the symbol p replaced by the symbols PPP .... P at every occurrence from left to right.

3. An output function designator of the form

$$£_n \; [PPP \; .... \; P \; , \; PPP \; .... \; P \; , \; .... \; , \; PPP \; .... \; P]$$

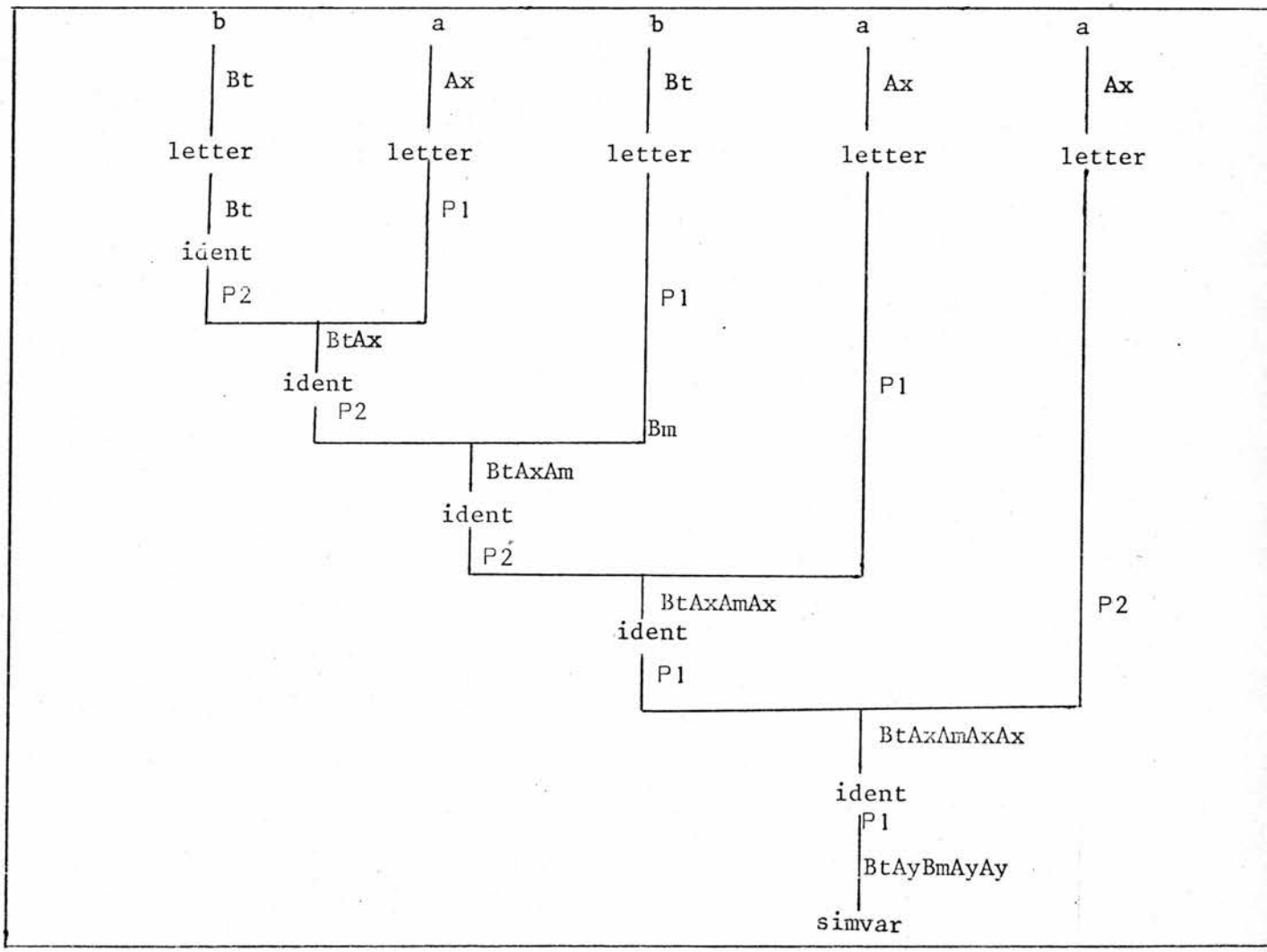
The output function designator may be empty. Each $£_n$ designates a different semantic housekeeping function - e.g. entering or looking up an identifier in a table.

2.3.1.1 Example

Let us now consider an example of the use of string designators in the five "sentences", where the input alphabet consists of letters "a" and "b", and the output alphabet is composed of "A", "B", "x", "y", "t", and "m".

```
                a =:: letter → {Ax}
                b =:: letter → {Bt}
          letter =:: ident  → {P1}
    ident letter =:: ident  → {P2P1 [t ← m]}
          ident =:: simvar → {P1 [x ← y]}
```

Thus, for an input string "babaa", it can be seen that the output produced will be "BtAyBmAyAy", and this is shown in Figure - 2.1



2.3.2 The Translator

The translator program operates on a set of specifications and on a string of symbols in the subject language for those specifications. It produces a string of symbols in the output language.

The translator program "diagrams" (produces a parse tree for) a string of input symbols by referring to the specifications. The output of this program is a linked list connecting various definitions together. In the operation of the translator, this output is a list which serves as input to a second program which forms the output string from the indicated sequence of definitions.

Given the name of a syntactic unit and the index of a symbol in the input string the program will parse the syntactic unit from the largest possible string of symbols following the one indicated. If it cannot, a failure is reported. The translator is table-driven and the parsing process is performed in a top-down manner.

2.3.3 Analytic Differentiation Using a Syntax-directed compiler

Herbert Schorr (Sch-62) (Sch-63) presents a method for performing analytic differentiation which differs from a method described by Hanson (Han-62) in that it makes use, not of a language for symbol manipulation, but instead of a syntax-directed compiler (Iro-63b).

The syntactic definition of the language is given in BNF and a translation is then added to the definition so that the derivative of any algebraic expression written in the language can be obtained by using a syntax-directed compiler. Normally the object string is in some form of assembly language; in the present instance it is the derivative of

the source expression. Algebraic expressions can be written using numbers, independent variables, dependent variables and normal arithmetic operators. In addition, for convenience, parentheses and certain standard functions, for example, Sine and Cosine are introduced.

The particular translation to be combined with BNF specifications can be obtained by using the rules of differentiation and observing that the derivatives of any algebraic expression is formed from:

(i) the components (or sub-expressions) of an expression, and

(ii) the derivatives of these sub-expressions.

For example, A * B has as its sub-expression A and B and its derivative is given by A * B' + B * A', where A' and B' are derivatives of A and B respectively. For the above expression two outputs are produced A * B the original expression, A * B, and A * B' + B * A', the result.

Let us consider a particular case as described by Schorr in syntax specifications

Sin (<expression>) =:: <expression>

for differentiation purposes this is written as

Sin (<expression>) =:: <expression>{Sin(ρ_2)|($\rho_{2.2}$)*Cos(ρ_2)}

the left of the vertical bar in braces is the original expression and the right its derivative. Thus Sin (x ↑ 2) is the input expression, the output produced are (i) Sin (x ↑ 2) and (ii) 2x Cos(x ↑ 2); where x ↑ 2 is equivalent x^2 and $\rho_{2.2}$ stands for the derivative of ρ_2. We may also

write

<independent variable>↑<number>=::<expression>

$$\{\rho_3 \uparrow \rho_1 | \rho_1 \rho_3 \uparrow \rho_{1.2}\}$$

where $\rho_{1.2}$ stands one less than the number. This, given the inputs x ↑ 1 and x ↑ n produces the outputs x ↑ 1 and 1 (if the argument x is independent variable - otherwise 0 if constant); and, x ↑ n and nx ↑ (n-1) respectively.

The full specification of syntax allows differentiation of functions expon, arcsin, log, etc., which may have as argument <number>, <expression>, <constant>, <independent variables>, etc. Derivatives of higher order, partial derivatives and the derivatives of implicit functions and complex functions can be obtained.

2.4 "Practical construction of syntax-directed Translators"

Rekdal (Rek-74) describes a technique with the above title for automatic generating of translators from syntax-directed description of inputs and outputs. The purpose of the translator is to devise a unified way of specifying both input and output and make a compiler which can generate a general translator, e.g. Rekdal and Wessell's (Rek-72).

The system has one read-only input file and one write-only file. These files are sequential and can move only in the forward direction and hence there is no backtracking. The translators produced are program driven and perform a bottom-up parse.

2.4.1. The specification Language "TEXT"

"TEXT" is the language developed by Rekdal to describe the structure of the contents of files, that is, texts. There is one description of the input file and one for the output file. The same language is used to describe both input and output specifications. TEXT is based on BNF, but has been extended to make it possible to describe texts completely. A brief introduction of TEXT is as follows.

Let us consider the following definitions in which TEXT is used to describe its own syntax:-

```
text.definition = <1:>  *  subtext.definition ;
```

This definition says that "a text definition consists of a set of 1 or more subtext definitions". Names or identifiers cannot contain spaces, so the symbol "." is used as a connection character. The symbols "=", ";", ":", "<", ">", "/", "(", ")", etc. are meta symbols used for the description of TEXT.

```
subtext.definition = left.hand.side '=' right.hand.side ';';
```

A subtext definition or production rule has a left hand side and a right hand side separated by an equal sign and terminated by a semicolon. Literals are denoted within quotes.

```
left.hand.side = text.name;
right.hand.side = list.of.alternatives;
list.of.alternatives = concatenation * ('/' concatenation);
```

The slash is used to separate alternative.

```
concatenation = * (text / <0:1>  * limits '*' text);
```

The asterisk is used to denote replication of the succeeding text. This is used as an alternative to left or right recursion. The number of times a text may be replicated, may be limited by an expression in angled brackets. If these are absent <0:> is assumed.

```
text = literal.text / text.name
     / consecutive.alternative
literal.text = quote basic.symbol quote ;
quote = '''' ;
```

one quote within a literal text is denoted by two quotes. Basic symbol encompasses all characters in ASCII character set.

```
text.name = <1:>  * name.symbol ;
```

A text name must consist of at least one symbol.

```
name.symbol = '.' / '0' - '9' / 'A' - 'Z' / 'a' - 'z' ;
consecutive.alternative = literal.symbol '-' literal.symbol ;
limits = '<'  lower.limit ':' * upper.limit '>' ;
```

If the upper limit is missing, it means that text may be replicated any number of times.

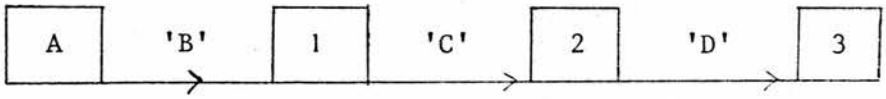
```
lower.limit = unsigned.integer ;
upper.limit = unsigned.integer ;
unsigned.integer = <1:3> * digit ;
digit = '0' - '9' ;
```

2.4.2 Graphical representation of Automata

TEXT is the source language of specifying input and output from which the text compiler generates the translator. The translator can be viewed as some kind of automaton. One way of visualising an automaton is by drawing a directed graph. A graph is very clearly represented as a linked list in a computer. The graphs and their construction also serve to visualise and document the implementation. Given a definition

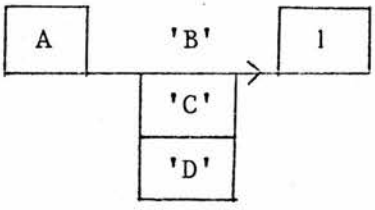
```
A = 'B'  'C'  'D' ;
```

a suitable graph representation is



Alternatives are expressed by giving the corresponding node one branch for each alternative.

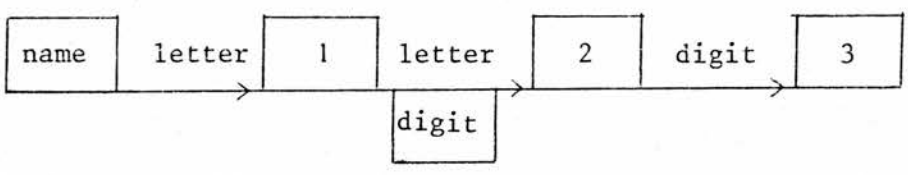
```
A = 'B' / 'C' / 'D' ;
```



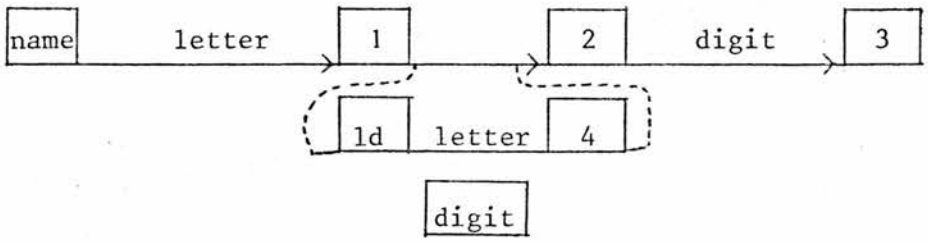
Given the text definition

```
name = letter ld digit;
  ld = letter / digit ;
```

If the right hand side of the second definition is substituted for ld in the first definition, a new definition is obtained which is equivalent to the former. Then the definition name has the graph



or



The linking is denoted by dashed lines.

From this general framework basically two types of automata, parsers and generators, can be constructed. In the translator both types are needed. When constructing the translator the general procedure is as follows. First a parser is constructed from input text definitions and a generator from the output text definitions. Finally the parser and the generator are merged. A reverse translator can be obtained by merely interchanging the definitions of the input and output texts.

In Rekdal's translator the parser is based on De Ramer's parser (De-69) (which we shall not discuss here). The parser is bottom-up and needs no backtracking, as explained earlier that the input and output files only move in the forward direction.

2.4.3. Example

Let us consider an example of input and output texts.

```
Input:
          clause = subject predicate ;
         subject = 'I';
       predicate = 'am' ;
Output:
          clause = subject predicate ;
         subject = 'Ich' ;
       predicate = 'bin' ;
```

The definition above specifies that if the clause "I am" is the input then the clause "Ich bin" should be the output. For example if the clause "I am" is to be changed to "am I", the input and output texts can be specified as follows:

```
Input

          clause = subject predicate ;
         subject = 'I' ;
       predicate = 'am' ;

Output

          clause = predicate subject ;
         subject = 'I' ;
       predicate = 'am' ;
```

2.5 A Parametrized Compiler

Metcalfe (Met-64) described a syntax-directed compiler called "A Parametrized Compiler Based on Mechanical Linguistics", which accepts as parameters the specifications of the languages between which it is to translate. In other words a single compiler which could translate languages (which may be problem-oriented) into target languages (which may be machine-oriented), by insertion of a set of parameters which themselves rigorously define two languages and their relationship. Metcalfe's compiler can be said the father of the one implemented here due to Reeves.

The compiler is composed of two machines, a syntax machine and an editor machine. The syntax machine has thirteen instructions which help the compiler in checking the validity of input text, according to the

syntax specifications, and produce output which is determined by the semantics of the specifications. This output is then interpreted by the editor machine which has another three instructions.

In this compiler the parser is top-down and back-tracking, that is, as soon as it fails to recognise the input at a point, it backtracks and attempts the same goal in a different way, according to the syntax specified.

More is not said here about this method as it is so similar to the one due to Reeves which is implemented in this work.

2.5 Feldman's Compiler-compiler (Fel-66)

The syntax-directed compiler of Irons' (Iro-61) and the compiler-compiler of Brooker and Morris (Bro-63) led to the speculation that the entire process of compilation could be automated. The program was to develop a single program which could act as translator for a class of languages differing from each other in a substantial way. To solve this so called compiler-compiler problem one must find appropriate formalisation of syntax and semantics of computer languages.

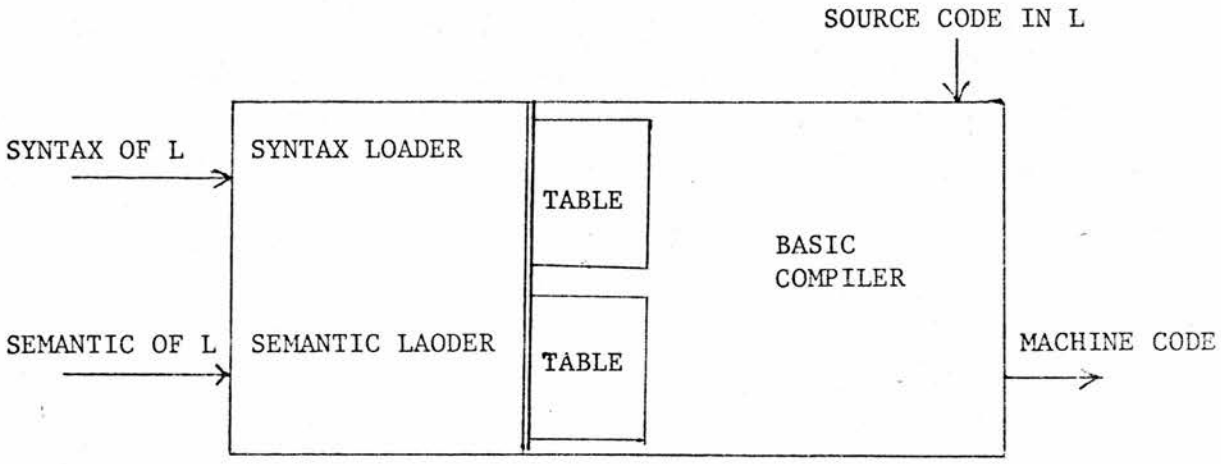
The formalization of semantics of some language L will involve representing the meanings of statements in L in terms of an appropriate meta-language.

In this manner, when a compiler for some language L is required, the following steps are taken (see Figure 2.2). First the formal syntax of L, expressed in a meta-syntactic language, is fed into a syntax loader. This program builds tables which control the recognition and parsing of programs in the language L. Then the semantics of L, written in a meta-

semantic language, is fed into the semantic loader. This program builds another table; this one containing a description of meaning of statements in L. Finally every thing to the left of the double line in the Figure is discarded, leaving a compiler for L.

2.6.1 Syntax and Semantics

The syntax phase of the system is based on a model of a sequential recogniser with a push-down stack. The meta-syntactic language describes



(Figure - 2.2)

(Feldman's compiler-compiler)

the behaviour of the recogniser and is thus a recogniser oriented meta-syntactic language. The formal properties of this type of meta language have been discussed in (Eve-63) (Fel-64)(Flo-64), and are not treated here.

The syntax of a source language written in meta-syntactic language specifies the operation of a recogniser as it scans a piece of text in that language. When a character is scanned, it is brought into the stack of the recogniser. The symbol at the top of the stack is compared with the meta-syntactic specification of the language being translated. When the character in the stack matches one of the specified configurations, certain changes are made in the stack. If a match is not attained, the stack is compared with the next statement in the meta-syntactic specifications. This process is repeated until a match is found or until the stack is discovered to be in an error state.

The form of a rule (production or statement) in meta-syntactic language is given below:

Label Ln .... L2 L1 | → Rm .... R1 | Action Next

In this, the first vertical bar from the left represents the top of the stack of the recogniser. The symbols Ln .... L1 in a given rule will be syntactic classes which could be encountered in translating the source language. The symbols in position L1 is on the top of the stack.

A match occurs when the classes actually in the stack, during the translation, are the same as Ln .... L1 in the rule being compared. If a match occurs, the remainder of the information in the matched rule is used. The symbol "→" specifies that the stack is to be changed. The symbols in positions Rm .... R1 name syntactic classes which must occupy the top of the stack after the change. If no "→" occurs, then the stack will not be changed and Rm .... R1 should be blank. If any "action" is specified after the second vertical bar, this semantic is carried out.

Let us consider the following definitions in ENF:

<term> ::= <factor> | <term> <op> <factor>

<op> ::= * | /

<factor> ::= <identifier>

for these defintions the production syntax will be

```
T1    T   *   F   <SG> |→ T  <SG> | EXEC 10    T2
      T   /   F   <SG> |→ T  <SG> | EXEC 11    T2
              F   <SG> |→ T  <SG>               T2
```

As described earlier, here T1 and T2 are labels. T and F are term and factor respectively. <TD> stands for type of the multiplication operator * or /. EXEC10 is an "action" field. And finally <SG> represents any character and is always matched. Considering the above statement labelled T1, this production is matched when the characters of the top of the stack are

T * P <SG>

The execution of the production T1 will leave the stack in the configuration T<SG>. The statement "EXEC10" in the action field of the production T1 is a call for the semantic routine associated with multiplication. The other possible actions are "ERROR n" which prints an error message, "HALT", which ends compilation, and "SCAN" which causes a new character to be read into the top of the stack. The next field of the production contains the symbol "T2". This specifies that the next production to be compared with the stack is the one labelled "T2".

The formal semantic language (FSL) can be considered as a problem oriented computer language. The problem involved is the representation of meaning in computer language. A complete description of the FSL is given in (Fel-64). The discussion here will only be to present basic features and general nature of the formal semantic language.

The basic unit of a formal semantic language program is a labelled statement or sentence. As mentioned above a production may include a statement of the form "EXEC n" in its "action" field. The semantic routine labelled "n" will be executed each time the production is matched. A semantic routine may generate code, change the state of the translator or both.

The distinction between run time and compile time operation is an important one in FSL and is represented explicitly. The means of doing this is the use of paired code brackets "CODE(" and ")". Any statement enclosed in code brackets specifies an action to be taken at run time, while any statement not so enclosed describes action occurring at translate time. One of the principal tasks of the compiler is the recording and use of housekeeping information about the program being translated. This is done by various operations involving cells, tables, push-down stacks and tag bits.

The resulting compiler is table-driven based on a bottom-up recogniser using push-down stacks. Each element in the stack consists of two machine words - one for syntactic construct and the other holding the semantic of the construct. When a particular construct is recognised, its semantic word and semantic table determine the action the translator will take.

A more detailed discussion of the implementation, along with documentation may be found in (Fel-64). The input-output is similar to that described in (Per-65) for all languages defined in the FSL system. This compiler-compiler has been used for the production of a compiler for an extremely complex language, Formula ALGOL (Per-65), using a formal semantic language.

2.7 YACC - Yet Another Compiler-Compiler (Joh-72)

YACC provides a general tool for controlling the input to a computer program. The YACC user describes the structure of his input, together with the code that is to be invoked when any such structures are recognised. The user prepares a specification of the input process; this includes rules which describe the input structure, code which is to be invoked when these structures are recognised. YACC then produces a subroutine to do the input procedure; this subroutine (the parser) calls a user-supplied low-level routine (the lexical analyser) to pick up the basic items (tokens) from the input stream. These tokens are organised according to the input structure rules (grammar rules). When one of these rules has been recognised, then the user code supplied for this rule is invoked.

Input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month-name day ',' year;
```

Here, date, month-name, day, and year represent a structure of interest in the input process; presumably month-name, day and year are defined elsewhere . The comma "," is quoted by single quotes; this implies that the comma is to appear literally in the input and is treated as a constant. The colon and semicolon merely serve as punctuation (meta symbols) in the rule and have no significance in controlling input. Then the input "July 4, 1978" might be matched by the rule above.

Frequently, the input being read does not confirm to the specifications due to error in the input. The parsers produced by YACC have the very desirable property that they will detect the input errors at the earliest place at which this can be done with a left-to-right scan. Error facilities can be entered as part of the input specification.

YACC requires those names which will be used as token names to be declared as such. In addition it is desirable to include the lexical analyser as part of the specification file. Every specification file consists of three sections: the declarations, (grammar) rules and programs for semantic subroutines and the lexical analyser. The specifications are separated by a double "%%" marks. A full specification file looks like as following

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover if the program section is omitted, the smallest legal YACC specification is

```
%%
rules
```

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

As described earlier, "A" represents a nonterminal name, the rule being defined, and the BODY represents a sequence of zero or more names and literals. Several grammar rules may be constructed with the same nonterminal name. For example

```
A :  B C D ;
A :  E F ;
A :  G ;
```

these rules can also be combined into one and can be given to YACC as

```
A : B C D | E F | G ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable and easy to change. If a nonterminal symbol matches the empty string, this can be indicated by the following rule:

```
EMPTY :  ;
```

The user may associate an action (the semantic) to be performed in a grammar rule. An action produces output,call other subprograms or return a value. An action in a rule is specified by an equal sign "=" at the end of the rule followed by one or more statements enclosed in curly braces "{" and "}". For example a grammar rule with an action in c is

```
X = '(' B ')' = {printf ("message");}
```

which prints "message" if the rule is recognised. The semantic actions are written in the systems programming language C(Rit-72).

To return a value, an action may set the pseudo-variable "$$" to some integer value. For example, an action which does nothing but returns a value 1 is

```
={$$=1;}
```

To obtain the values returned by previous actions and the analyser, an action may use integer pseudo-variables $1, $2, $3, ... which refer to the values returned by the components (syntactic constituents) on the right hand side of a rule, reading from left-to-right. Thus if the rule is

```
A : X Y Z ;
```

then $1 has the value returned by X and $3 has the value returned by Z. A grammar rule of the form

```
A : B ;
```

need not have an explicit semantic. The value returned is always $1.

The user may wish to get semantic control before a rule is fully parsed, as well as at the end of the rule. There is no explicit mechanism in YACC to allow this; the same effect is obtained, however, by introducing a new symbol which matches the empty string and inserting an action for this empty string. Let us consider the following

```
Statement : IF '(' expr ')' THEN statement
```

Suppose we wish to get semantic control after seeing the right parenthesis in order to generate some code. We might accomplish this by the rule:

```
Statement : IF '(' expr ')' actn THEN statement = {call action1}
```

```
actn : /* matches the empty string */ = {call action2}

                (/* .... */ denotes a comment)
```

Thus, the new terminal symbol matches no input but serves only to call "action2". The above rule can also be written in the following way:

```
statement : ifpart THEN statement = {call action1}

   ifpart : IF '(' expr ')' = {call action2}
```

The algorithm used by YACC parser allows left recursive grammar rules, that is, the rules of the form

```
name :  item | list ',' item ;
```

and

```
sequence :  item | sequence item ;
```

YACC provides a simple feature for error handling. It is generally not acceptable to stop all processing when an error is found; we wish to continue scanning input to find any further syntax errors. This leads to the problem of getting the parser restarted after an error. The general class of algorithm to do this involves reading ahead and discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user to some control over this process, the token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected and recovery might take place. The parser attempts to find the last time in the input when the special item "error" is permitted. The parser then behaves as though it saw the token name "error" as an input token and attempted to parse according to the rule encountered. The token at which the error was detected remains the next input token after this error token

is processed. If no special error rules have been spccified, the processing effectively halts when error is detected.

The user might include a rule of the form

```
statement : error ';' ;
```

Here, when there is an error in the input, the parser will attempt to skip over the next ";". All the tokens after the error and before the next ";" give syntax errors and are discarded.

To summarise, YACC can be used to construct parsers. These parsers can interact in a fairly flexible way with the lexical analysis and output phases of a large system. The system also provides an indication of ambiguities in the specifications. Because the output of YACC is largely tables, the system is relatively language independent. In the presence of reasonable applications, YACC could be modified or adapted to produce subroutines for other machines and languages.

YACC has been used in the construction of a C (Rit-72) compiler for the Honeywell 6000, a system for typesetting mathematical equations, a low level implementation language for PDP 11 computer, APL and BASIC interpreters to run under UNIX system and a number of other applications.

2.8 A Description of the Revised Compiler-Compiler System (Fis-77)

The Revised compiler-compiler (RCC) provides a syntax machinery together with a high-level language in which the user may explicitly describe the semantics of the language he wishes to implement. RCC is designed as a high-level and machine independent language for use in system programming in general, and for compiling compilers in particular.

The basic language of RCC is a one-pass language. The parsing algorithm used by the RCC system is top-down, left to right and with backtracking. Restrictions apply to grammars parsed by such an algorithm e.g. no left-recursion must be used to the syntax defined in the RCC meta-language.

The overall structure of an RCC program is a set of program units (sub-programs). Each unit is preceeded by a "master word" (a key word). Most commonly they are: ROUTINE, FORMAT, GLOBAL or CLASS. GLOBAL heads declaration of global identifiers, FORMAT heads a set of routine specifications, CLASS heads a declaration of a formal syntax class, and ROUTINE heads a routine definition.

2.8.1 The RCC Meta-Language

Simple class definitions are analogous to BNF. The trivial differences are that the meta symbols

< and >	are replaced by [and],
::=	is replaced by =
|	is replaced by ,

and that the name of a class be an upper case name. Thus, for example the BNF definition

<integer> ::= <digit><integer>|<digit>

becomes in RCC

[INTEGER] = [DIGIT][INTEGER] , [DIGIT]

It is sometimes desirable to give a class name which is other than a sequence of upper case letters. This is allowed in RCC by replacing the brackets [and] by [and]. Thus one may write class names for example [digit-0-9] or [0:9].

A meta symbol may be used as a terminal symbol by using ?NB as pre-editing. For example

[LISTSEPARATOR] = ,& , ,AND, , ? NB ,=,

states that a LIST SEPARATOR is either ",&" or ",AND" or ",".

The RCC meta language has another two meta symbols - "NOT" and "BUT". The use of these meta symbols is shown in the following example

[OCTAL DIGIT] = NOT 8, 9, BUT [DIGIT]

which may be read as an "OCTAL DIGIT" is defined to be any digit, but 8 and 9.

It is possible to define a class to be a "list class", that is, a sequence of one or more occurences of a given alternative may be separated by a second alternative, for example

[INTEGER] = LIST OF [DIGIT]

and

[MULTIPLE CONDITION] = LIST OF [IF][COND], SEPARATED BY; [ANDOR]

The second definition is equivalent to the infinite definition

[MULTIPLE CONDITION] = [IF][COND],[IF][COND];[ANDOR][IF][COND],
[IF][COND];[ANDOR][IF][COND];[ANDOR][IF][COND],

(Note: semicolon is not a meta symbol).

If a class name ends with "*" and is found undeclared, it is implicitly declared to be a list class, for example

[LABEL] = [DIGIT*]

is encountered, the class [DIGIT] has not been previously defined. Then the implicit definition

[DIGIT*] = LIST OF [DIGIT]

is inserted.

Because of backtracking involved, the RCC parser can be inefficient in certain circumstances. To avoid it (or to make the parser rather efficient) "restricted forms" UC, LC, UC, LC, RD, R may be used. For example,

[SMALL LETTER] RD = a, b, c, ... x, y, z

In this, "RD" ensures that average recognition or non-recognition may take (in this particular example) 3 or 4 comparisons (attempts of matching) in all rather than 13 comparisons for success and 26 for failure. The explanation of other "restricted form" may be found in (Fis-77).

CHAPTER 3

A META-SYNTACTIC LANGUAGE

In this chapter we shall describe a Meta-Syntactic language which we shall use to describe the syntax of programming languages.

Meta-Syntactic Language

A language in which another language can be defined is termed a meta-language and must be uniquely distinguishable from the language being described. According to the official document which defines the ALGOL 60 language [Bac-63] a special notation, the so called Backus-Normal-Form or Backus-Naur-Form (BNF), is used for defining the grammar or rules of syntax of ALGOL. This meta language is simple, powerful and general. It is the basis of most formal specifications of current programming languages. We shall take it as our starting point. The following adds some extensions to it.

3.1 Definitions

Backus notation represents the grammar of a language as a set of definitions of grammatical structures. Each definition has three components:

First the name of the structure being defined,

Second the symbol "::=" which is read as "is defined as",

Third a specification of the permitted forms of the structure.

The symbol "::=" belongs to the meta language, not to the language whose grammar is being defined, and is termed a meta symbol.

The names of the structures being defined may be spelt using symbols which do belong to the defined language, and so in order to avoid any possible confusion, these names, whenever they occur, are enclosed between the angled brackets "<" and ">". These brackets are further meta symbols.

3.2 Specifications

Now we come to the specification part of the definition. This is an expression which is a list of alternative permitted forms of a structure, pairs of alternatives being separated by the meta symbol "|" [Reeves uses "↑"] which is read as "or". For example

<factor> ::= <variable>|<term>

has three components:

(i) <factor>, which is on the left of the definition, is the name of the structure being defined,

(ii) the symbol "::=", and

(iii) <variable>|<term>, as a meta-expression.

The above expression has two elements <variable> and <term> separated by the meta symbol "|", and hence they are alternative permitted forms of <factor>. The above can be read as "a factor is defined as a variable or a term".

3.3 Alternatives

Each alternative is a list of elements written one after another. The sense of an alternative is obtained by reading "followed by" between adjacent pairs of elements. There may be zero alternatives or elements in an alternative; this signifies emptiness. Finally an element is one of the two types, a constant or a variable. A constant is any symbol belonging to the character set or alphabet of the defined language, and it represents itself. A variable has the form of the name of a structure in angled brackets and represents any of the permitted forms specified by a definition of the type just described. This definition is presumably to be found elsewhere in the description of the language. As

an example we find in the ALGOL report

<digit> ::= 0|1|2|3|4|5|6|7|8|9

which reads "a digit is defined as the symbol '0' or the symbol '1' or ..... or the symbol '9'. And

<unsigned integer>::=<digit>|<unsigned integer><digit>

which reads "an unsigned integer is defined as a digit or an unsigned integer followed by a digit".

Thus in parsing the structure 798, we recognise 7 as a digit and taking the first alternative as an unsigned integer also. 9 is also a digit and hence from the second alternative, 79 is an unsigned integer. Similarly 8 is a digit and from the second alternative 798 is an unsigned integer.

3.4 Recursion and Iteration

A definition in which the structure being defined appears within the definition expression is known as recursive. The ALGOL report makes widespread use of recursive definitions. An alternative to recursion may often be obtained by an extension of Backus' meta language. We introduce another meta symbol "*" (Metcalfe use "ʃ" [Met-64]) which may precede an element in an expression and is read as "a sequence of none or many". And so

<unsigned integer> ::= <digit>*<digit>

should be read as "an unsigned integer is defined as a digit followed by a sequence of none or many digits". [Note: In some usage "*" is used after the element, not before, e.g. <digit><digit>*].

3.5 Sub-expressions

Let us look back again at the ALGOL report. We find

<identifier>::= <letter>|<identifier><letter>|<identifier><digit>

The repetition of <identifier> in the expression part of the definition can be avoided by introducing the meta symbols "(" and ")" whose effect when enclosing an expression is to produce a meta sub-expression. Thus we can write

<identifier>::= <letter>|<identifier>(<letter>|<digit>)

or yet more compactly, using the meta symbol "*", as

<identifier>::= <letter> * (<letter>|<digit>)

which reads "an identifier is defined as a letter followed by a sequence of none or many letters or digits".

3.6 Self-description

Our meta language is now complicated. We have described it formally in English but it would be convenient to have some more formal and compact definition. To invent a meta-meta-language in which to describe our meta-language would be "embark on a journey that has no end". This is quoted directly from Reeves.

3.6.1 Constants

Let us instead consider the much more satisfying task of using our meta language to describe itself. As things stand this is impossible because our meta symbols have explicitly been excluded from the character set of the defined language, such as <, >, |, *, (, etc. This obstacle is removed by introducing a meta symbol "=" with the significance that

whatever symbol follows it is to be interpreted as a constant character of the language being defined and which for consistency must be placed in front of every constant. So that "|" on its own is a meta symbol but "=|" denotes the actual symbol "|" and also "=*" denotes the actual symbol "*".

3.6.2 Universal Symbol and Negation

We have so far discussed the various symbols forming the character set of the defined language without in any way specifying what they are. Further we introduce two meta symbols "U" and "¬" (Reeves uses "$\underline{U}$" and "≠") where

U	denotes "any symbol in the alphabet of the language being defined", and
¬	preceding an element denotes "any sequence of symbols other than that element". (This meta symbol may precede a whole sub-expression)

This is easily understood when the element is a constant. Thus ¬=9 means any symbol other than digit 9. It is possible to have more complex forms, such as, the meta sub-expression:

¬(=A=B|=A=C)

which denotes

(i) any symbol other than "A"

or

(ii) "A" provided it is not followed by "B" or "C"

The above expression can also be written as

$$\neg\,(=A\ (=B\,|\,=C))$$

The acceptability of "A" thus depends on its context.

All the symbols so far introduced are available in the character set of the IBM 360/44 computer. In Reeves' Syntax-directed translator all the symbols introduced (e.g. U, A, A, etc) were available in KDF9 paper tape code. In our case only upper-case letters are allowed, such as "A", "B", etc., since the implementation is done in the programming language ALGOL W (which has no convention of lower-case letters) on the IBM 360/44 computer.

3.7 Syntax of the Metalanguage

We can now define the syntax of our meta syntactic language, which is as follows, where we use the meta symbol ";" to separate definitions

```
    <grammar> ::= <definition>*(=; <definition>);
 <definition> ::= <variable>=:=:== <expression>;
   <variable> ::= = < * ¬ =>=;
 <expression> ::= <alternative>* (=| <alternative>);
<alternative> ::= *<element>;
    <element> ::= <variable>|<constant>|=*<element>
                  |=(<expression>=)|=U|=¬<element>;
   <constant> ::= == U
```

3.8 Example

Previously we described the syntax of an English sentence in a tree fashion [see Figure 1.2] . Now we are able to write the grammar of the

English sentence in our meta language using meta symbols defined to describe the meta language. Following is a grammar whose language include the English sentence "THE CAT SEES A MOUSE".

```
    <Sentence>::= <subject><predicate>;
     <Subject>::= <noun phrase>;
<noun phrase>::= <article><noun>;
  <predicate>::= <verb><object>;
     <object>::= =T =H =E| =A;
       <noun>::= =C =A =T | =M=O=U=S=E;
       <verb>::= =S=E=E=S
```

In the next chapter we shall describe a hypothetical machine for carrying out top-down parsing of sentences in languages described by such grammars.

CHAPTER 4

A PARSING MACHINE

This chapter describes a simple process for determining whether a given source text is grammatical in a language whose syntax is specified as has been described in the previous chapter. The process is performed by a hypothetical special purpose stored-program computer. The project has been to realize this machine by simulating an interpreter for it on a real machine - the IBM 360/44. We shall first describe the computer and then how to write programs for it.

4.1 Parsing Machine

The data for this parsing machine is an ordered sequence of symbols forming the source text. The program which the machine obeys is the parsing algorithm for the source language. The program is an ordered stream of instructions, and instructions are normally obeyed in sequence.

The data is scanned one element at a time and the point of scan is normally moved in one direction to the right - but the position of this point of scan may be recorded and subsequently reset if desired. This will happen when a sub-goal has not been achieved and an alternative sub-goal must be tried.

Besides storage space for data and program, the machine has,

(a) two pointer registers: (i) "input", the input pointer, and
(ii) "pcr", the parser control register for recording the current state of the machine - the instruction about to be obeyed.

(b) a Boolean register "flag", whose value true or false indicates the success or failure of different stages in the parsing process.

(c) A stack (see below) and a pointer to it.

Initially the points of scan are set to the positions of the leading elements of the two streams, that is, the "inpt", to the first element of the input sequence and "pcr" to the first element of the program sequence. The stack is empty and the flag is set to true. The stack is a vector. The base of the stack is the first position in the vector. A pointer is set to the top-most element of the stack. Insertion of an item consists of incrementing the pointer and inserting at the address pointed to by the pointer.

The control or interpretation cycle of the machine consists of inspecting the instruction of the program specified by the "pcr", carrying out the indicated operations which must include changing "pcr", normally to the position of the successor in sequence, and returning to commence a new interpretation cycle.

The instruction code is of single-address type and has the structure (F,A), where F, the function part specifies the operation, and A, the address part, the operand. The address part may be either a symbol of the defined language or a pointer value identifying the address of some particular instruction of the program, or it may be null.

There are nine computer like instructions CALL, RETURN, TRUE, FALSE, STOP, FLAG etc. The full purpose of these instructions will be made clear later when we describe them and their synthesis into complete programs. Briefly, however, CALL and RETURN are subroutine entry and exit instructions which use the stack to administer the return addresses: they allow sub-goals to be aimed at and the stack is used to remember which goals have to be returned to; TRUE, FALSE and FLAG are conditional jump instructions; STOP is an output instruction.

4.1.1. Available Instructions

The list of available functions F and their effect is as follows. Except where the contrary is explicitly stated it is implicit that their effect includes advancing "pcr" to the location of the next instruction in the program sequence.

MATCH A — This is used to recognise individual constants (the "ultimate constituents" described in Section 1.3.3 of Chapter 1). If the source symbol at the current point of scan is the same as that in the address part A, it sets flag true and advances inpt to the position of the successor symbol in the sentence to be parsed, otherwise sets flag false.

NEXT — This recognises any basic element. It is used to implement the meta symbol "U". It sets flag to true and advances inpt to the successor symbol. The address part is ignored.

CALL A — This is used as a subroutine call to recognise a whole sub-expression. It records the value of inpt in the stack together with, if address part A is null, then null otherwise the location of the next instruction in the program sequence. If A is not null then it sets A in pcr. It sets flag true. (If A is null the subroutine is "in-line". Details of the use of CALL in this context are described later).

RETURN — This returns control to the caller after a subroutine call. If flag is false then it restores inpt from the value in the top of the stack.

If the current register setting on the top of the stack is not null then it restores this value to pcr. This pair of values is unstacked.

NOT — This is used to implement the meta symbol ¬ (see Sections 3.6.2 and 3.7). This instruction accepts any symbol or string of symbols other than the one recognised by the instructions which it follows. This instruction normally follows a MATCH instruction or a CALL instruction (with null address) (See Section 4.3).

It reverses the setting of flag and restores inpt by popping it from the stack. If flag is now true inpt is advanced to its successor position.

TRUE A — This is used as a conditional branch instruction. If flag is true then it copies the address part A into pcr, otherwise it sets flag true and resotres inpt from the top of the stack leaving the stack unaltered. This allows alternative meta-sub-expression to be tried in a grammatical rule.

FALSE A — This is another conditional branching instruction. If flag is false then it copies the address part A into pcr. It is used to check that elements of an expression follow each other correctly in a grammatical rule.

FLAG A — This is used when a subroutine is called iteratively (when the meta symbol * is used). If flag is true it copies the address part A into pcr (this will normally be a backward jump). Otherwise it sets flag true .

STOP — This is an output instruction. It displays the value of flag and stops the machine. It thus indicates whether the main goal was achieved or not.

Before discussing the programming of the machine we shall describe informally the basis of the parsing process. The fundamental operation is to test whether any portion of the input text following the current point of scan satisfies the definition of the syntax structure which is currently of interest: if so the point of scan is advanced to the end of this portion. In making this test, each alternative is tested in turn in order listed, and whichever is first satisfied is accepted. In testing an alternative which may consist of more than one element each is then tested in turn until either one is rejected or the list is exhausted. If a rejection occurs the 'inpt' will be restored by a TRUE or FALSE command and another alternative tried.

4.1.2 Example

As an illustration, let us use the definition

<name>::=<letter><name>|<letter>;

To make our illustration smaller and easier to understand let us consider only three constants A, B and C in a second definition, that is,

<letter>::= =A|=B|=C

To test whether a sequence is acceptable as <name>, we use → to denote the point of scan. The process goes as follows:

	goal	input
(1)	?<name>	→AB;
(2)	?<letter>	→AB;
(3)	Match "A"	→AB;
(4)	Accept "A"	A→B;
(5)	Accept <letter>	A→B;
(6)	?<name>	A→B;
(7)	?<letter>	A→B;
(8)	Match "A"	A→B;
(9)	Reject "A"	A→B;
(10)	Match "B"	A→B;
(11)	Accept "B"	AB→;
(12)	Accept <letter>	
(13)	?<name>	
(14)	? <letter>	
(15) - (20)	test and reject each of three constants	
(21)	Reject<letter>	
(22)	?<letter> (Second alternative <letter> is being tried)	
(23) - (28)	same as (15) - (20)	
(29)	Reject<letter>	
(30)	Reject<name>	

At this point the first alternative <letter><name> of the test begun at step (6) has failed. We therefore try the second alternative <letter>, resetting the point of scan to its value at step (6), which is A→B;

(31)	?<letter>	A→B;
(32) - (35)	Same as (8) - (11)	AB→;
(36)	Accept <name>	AB→;

The process thus works successfully but is very inefficient. It would do better with the use of a null alternative in the first definition and we may write:

<name>::= <letter>(<name>|);

Using the above definition, it can easily be verified that the process is faster than the one illustrated above. It is because of the null alternative which enables us to accept the sequence →AB; without resetting the point of scan. An even better way to write the definition <name> will be using iteration and we will write

<name> ::= <letter>*<letter>;

this is more elegant and the process is yet more efficient, because once <letter> is rejected we do not try again but accept "*<letter>" and finally <name> is accepted.

4.2 Difficulties with Definitions

Metcalfe (Met-64) describes some major errors that can be made in constructing grammars for our syntax machine. Following are some of the difficulties in writing definitions in a grammar and their solutions in order to construct them in a more elegant fashion and make the parsing process efficient.

(i) Misordered Alternatives

A difficulty may arise from the ordering of alternatives in some definitions. Considering the same definition as we had used in illustrating the parsing process, suppose we had written

<name> ::= <letter>|<letter><name>;

If we test the input sequence "AB" as name it would accept "A" as <name> because the first alternative in the definition is a <letter>; and that is not what we intended to do. A solution to write the definition name correctly will be

<name> ::= <letter><name>|<letter>;

As mentioned above, a better solution will be using a null alternative or iteration in order to make the parsing process efficient, that is,

<name> ::= <letter>(<name>|);

or

<name> ::= <letter>* <letter>;

(ii) Left Recursion

A different problem would arise if we had written

<name> ::= <name><letter>|<letter>;

This definition is known as a left-recursive definition. This causes top-down parsing processes to go into a loop. Therefore in constructing definitions left-recursion should be avoided. The solution to the problem is to write the recursion on the right and then we get something similar to that explained above - "misordered alternatives". (Note: strangely enough, bottom-up methods tend to be more efficient when left-recursive definitions are given).

(iii) Common Constituent:

Let us consider the following definition

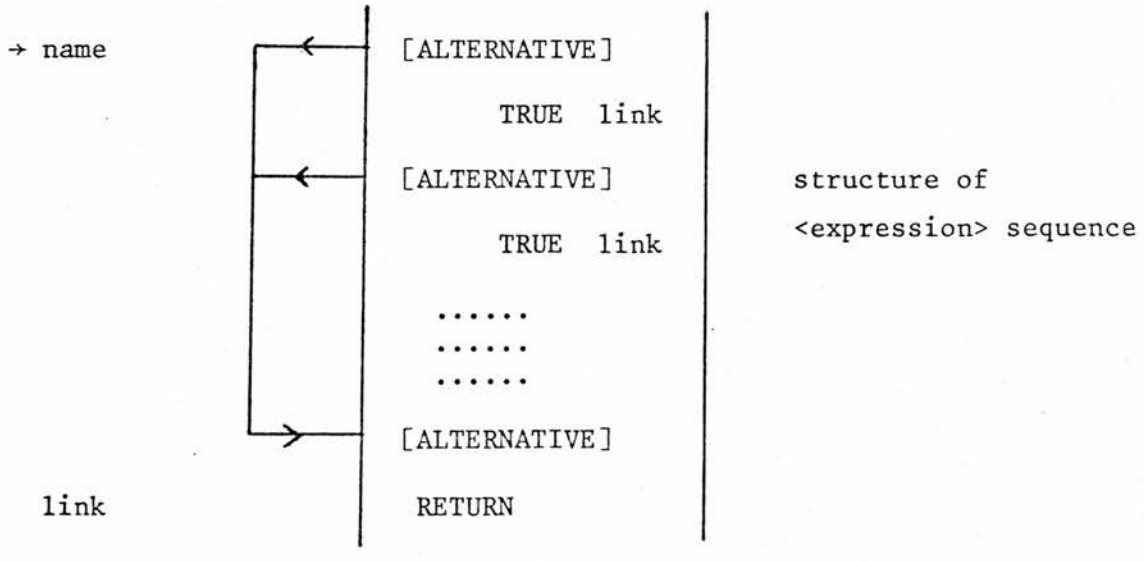
<number> ::= <integer><fraction>|<integer>|<fraction>;

This says, a <number> is defined as an <integer> followed by <fraction> or an <integer> or <fraction>. In this definition the alternatives have <integer> and <fraction> as common constituents. A better way will be to write it as

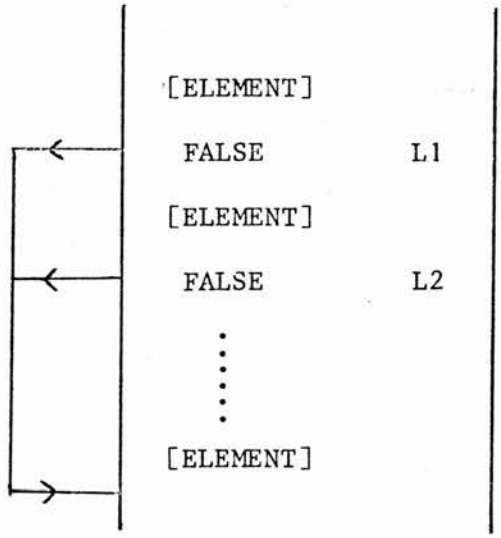
<number> ::= <integer>(<fraction>|)|<fraction>;

4.3 Implementation of the Parsing Process

We are now in a position to implement the parsing process on our computer. Each syntactic definition of the meta language is represented by a subroutine in the program for the parsing hypothetical computer. Our eventual aim is to produce these programs automatically. The label attached to the head of the routine corresponds to the name of the structure defined, and its instructions to the defining expression. These instructions are partitioned into sequences corresponding to each alternative. The function of each such sequence is to set the flag corresponding to acceptability of that alternative. Pairs of such sequences are separated by a TRUE instruction causing a conditional jump to the end of the sub-routine where the link instruction RETURN is placed. Thus as soon as an acceptable alternative is found, the rest are skipped, but when unacceptable ones are encountered, the "inpt" is reset and the next alternative is tried.



An alternative may consist of a number of elements and each element is examined in turn. The group of instructions for each element sets the flag according to the acceptability of that element. Between pairs of such groups is placed a FALSE instructions causing a conditional jump to the end of the alternative. Thus, if any element is rejected, the flag is set to false and the rest are skipped, so rejecting the whole group.



Structure of
<alternative> sequence

Let us consider the definition

<s> ::= <t><p>|<v><f>;

In this definition <s> is defined as <t> followed by <p> or <v> followed by <f>. Thus <s> has two alternatives (i) <t><p>, and (ii) <v><f>. The first has two elements, <t> and <p>. Similarly the second alternative has the elements <v> and <f>.

The instruction for the above definition <s> may now be expanded as following:

```
        code for <t>
        FALSE    L1
        code for <p>
L1      TRUE     L2
        code for <v>
        FALSE    L2
        code for <f>
L2      RETURN
```

In fact, if the element is of * type, we can omit the FALSE instruction since we know that it is always acceptable. The structure for *<element> sequence is written as

```
L1  | [ELEMENT]      |
    |  FLAG     L1   |
```

Let us consider a definition and write instruction for it.

Definition:

<identifier> ::= <letter>*(<letter>|<digit>);

Instruction:

```
        code for <letter>
        FALSE    L1
        ------------
L2      CALL       |
        code for <letter>
        TRUE     L3          is always accepted
        code for <digit>
L3      RETURN     |
        FLAG     L2
        ------------
L1      RETURN
```

Such a term (*) is called passive as opposed to those which require a test and which are called active. U is another passive element (see Sections 3.6.2 and 3.7). Since U accepts any symbol, the instruction sequence for U is

NEXT

The instruction sequence for other types of elements is given below. They are reasonable and self-evident

CALL name	, for <variable>, the address part is the location of the head of the relevant (calling) routine.
MATCH constant	, for <constant>.
CALL [EXPRESSION] RETURN	, for (<expression>).

Note that the address part of CALL is null. An in-line subroutine is generated. A meta sub-expression (<expression>) in the grammar is used if the sub-expression has more than one alternative. For example, let us consider the definition

<number> ::= <integer>(<fraction>|)|<fraction>;

which says, a <number> is defined as an <integer> followed by a <fraction> or nothing, or a <fraction>. In the above definition the first alternative has (<fraction>|) which is type (<expression>). For code corresponding to this, see below:

CALL [ELEMENT] NOT	, for ¬= <element>,

the address part of CALL is also null here as in (<expression>) above.

Finally the program is completed by placing at its head a call of the principal subroutine followed by a STOP command.

```
CALL    principal
STOP
```

For illustration, let us consider the following grammar and write the program in symbolic instructions.

```
<number>::= <integer>(<fraction>|)|<fraction>;
<fraction>::= =.<integer>;
<integer>::= <digit>*<digit>;
<digit>::= =0|=1|=2|=3|=4|=5|=6|=7|=8|=9
```

Following is the program corresponding to the syntax, where L1, L2, .., L7 are labels.

Label	Function (f)	Address part (A)	Items being programmed
	CALL	number	
	STOP		
Number	CALL	integer	<number>::=< integer>
	FALSE	L1	
	CALL		(
	CALL	fraction	<fraction>
	TRUE	L2	|
L2	RETURN		)

Label	Function (f)	Address part (A)	Items being programmed
L1	TRUE	L3	|
	CALL	fraction	<fraction>
L3	RETURN		
fraction	MATCH	.	<fraction>::= .
	FALSE	L4	
	CALL	integer	<integer>
L4	RETURN		
integer	CALL	digit	<integer>::=< digit>
	FALSE	L5	
L6	CALL		*
	CALL	digit	<digit>
	RETURN		
	FLAG	L6	
L5	RETURN		
digit	MATCH	0	<digit> ::= 0
	TRUE	L7	|
	MATCH	1	1
	TRUE	L7	|
	⋮		⋮
	MATCH	9	9
L7	RETURN		

The next illustration is the program corresponding to the syntax of the meta language itself as set out in the last chapter. This program will check the validity of the rules of syntax of an arbitrary language when they are expressed in terms of the meta language.

It will be the eventual aim to use this not only to recognise a grammar as correct but to generate a parsing program from it.

Label	Function (F)	Address Part (A)	Rule being programmed
	CALL	grammar	
	STOP		
grammar	CALL	definition	<grammar>::=<definition>*(=;
	FALSE	A1	<definition>);
A3	CALL		
	MATCH	;	
	FALSE	A2	
	CALL	definition	
A2	RETURN		
	FLAG	A3	
A1	RETURN		
definition	CALL	variable	<definition>::=<variable>=:=:
	FALSE	B1	==<expression>;
	MATCH	:	
	FALSE	B1	
	MATCH	:	
	FALSE	B1	
	MATCH	=	
	FALSE	B1	
	CALL	expression	
B1	RETURN		

Label	Function (F)	Address Part (A)	Rule being programmed
variable	MATCH	<	<variable>::=<*¬=>=>;
	FALSE	C1	
C2	CALL		
	MATCH	>	
	NOT		
	FLAG	C2	
	MATCH	>	
C1	RETURN		
expression	CALL	alternative	<expression>::=<alternative>
	FALSE	D1	*(=|<alternative>
D3	CALL		
	MATCH	|	
	FALSE	D2	
	CALL	alternative	
D2	RETURN		
	FLAG	D3	
D1	RETURN		
alternative,E1	CALL	element	<alternative>::=*<element>;
	FLAG	E1	
	RETURN		
element	CALL	variable	<element>::=<variable>
	TRUE	F1	|<constant>
	CALL	constant	
	TRUE	F1	
	MATCH	*	
	FALSE	F2	

Label	Function (F)	Address part (A)	Rule being programmed
	CALL	element	|=*<element>|=(<expression>
F2	TRUE	F1	=)|=U =¬== <element>;
	MATCH	(	
	FALSE	F3	
	CALL	expression	
	FALSE	F3	
	MATCH	)	
F3	TRUE	F1	
	MATCH	U	
	TRUE	F1	
	MATCH	¬	
	FALSE	F4	
	MATCH	=	
	FALSE	F4	
	CALL	element	
F1, F4	RETURN		
constant	MATCH	=	<constant>::===U
	FALSE	G1	
	NEXT		
G1	RETURN		

The above implementation allows sentences in the languages of given grammars (subject to the constraints of not being left-recursive etc) to be recognised. It merely gives a yes-no answer. We now proceed to extend the machine so that output is generated as the parse proceeds.

CHAPTER 5

PARSER OUTPUT AND THE EDIT MACHINE

5. Semantics

This chapter describes how output can be generated from the parser machine as the parse proceeds. For this purpose we intend to extend our machine in such a manner that as soon as a sentence or a string has been recognised the output is generated. In the last two chapters we have talked about the structure and analysis of the grammar of a language. Now we turn to its meaning or interpretation. Putting this objectively, we shall now consider how to provide our parsing machine with extensive output facilities. The kinds of output we may require are as flexible as the input types and we need similar means to specify them. Thus our meta-syntactic language must be extended to encompass a meta-semantic language. We shall henceforth call it a meta language and understand that it specifies the structure of both input and output streams. In describing these extensions it is convenient to have a specific problem in mind to illustrate the motivation.

5.1 Translation of Grammars

The specific problem is to enable the extended machine to read a description of the syntax and semantics of an arbitrary language written in the extended meta language, and to produce as output the corresponding program for the extended machine which when executed will read a sentence in the specified language and produce the proper translated output.

Thus we wish to produce a compiler which will spare the user the chore of programming in the machine code of the parser machine. To be yet more specific, we wish to produce a program, such as the one given at the end of the chapter 4, automatically from the meta grammar, mentioned in Chapter 3.

There is an evident similarity between the input and output texts. Each syntactic structure in the meta grammar gives rise to a characteristic sequence in the corresponding recognition program. This suggests that the parser output should be caused by output instructions which are obeyed at points in the program closely following recognition of the corresponding structure in the text input. Provision must be made for rearranging the output stream. For example the symbol "¬" generates an instruction "NOT" which follows the instructions generated by the structure <element> which the "¬" precedes. Now we shall describe Metcalfe's two storage mechanism for achieving this.

5.2 Parser Output

Let us suppose that the parser machine is provided with an output facility. This consists of an ordered sequence of positions or vector. This, like the input, can be traversed in either direction, the current index point being recorded in a register. We call the register "outpt", the output pointer. Each position may record a printable symbol or other single item used for editing control (see section 5.3.1).

An output instruction loads the position indicated by the current point of scan with an indicated item and advances the "outpt" to the successor position. Certain instruction cause "outpt" to be restored to an earlier value and have effect of erasing all intermediate items from the output stream. This is done when backtracking occurs in the parse. Initially the output stream is empty and "outpt" indicates the leading position.

5.2.1 Output Instructions

The basic output instructions of the parser are as follows:

PRINT A	Outputs the symbol, A, in the address of the instruction i.e. places A in the output vector, advancing "outpt" to its successor position.
COPY	Outputs the symbol just read in - that is, the one immediately preceding that specified by "inpt" as the next to be read. This is used for copying input to output without change.
NULL	Outputs a special null symbol. Note this occupies a space in the output stream and causes "outpt" to be advanced - unlike a null input. The reason for this will become apparent. But briefly, the objects in the output vector should be thought of as strings (mostly one character long). Later, when the output is further processed by an editor, these may be manipulated (chiefly by concatenation). The null symbol represents a string of length zero.
EDIT A	Outputs an edit code, or control character, A, specified by the address part of the instruction. As mentioned above (in NULL), the output undergoes a further stage of editing. These edit codes can be thought of as instructions for an editor machine whereas the other objects in the output vector (strings) are objects to be manipulated by the editor machine.

5.2.2 Addition to Previous Parser Instructions

We distinguish between symbols and edit codes on the parser output. PRINT, COPY and NULL produce symbols (or strings to be more exact). Before discussing the distinction, we list the effects of output introductions on the parser. The parser stack is augmented at each level by a location which records a value of "outpt". These are processed as follows:

CALL	Records the value of "outpt" in the stack (for possible backup)
RETURN	If "flag" is false then restores to "outpt" the value from the top of the stack and pops it from the stack
NOT	Pops the value on the stack to "outpt"
TRUE	If the "flag" is false restores to "outpt" the value from the top of the stack, leaving the stack unaltered.

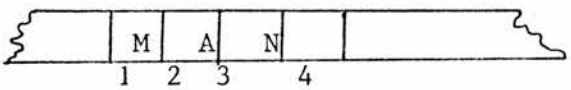
It can be seen that the effect of these added operations is to erase any output that may take place in the course of unsuccessful recognition attempts and thus to backtrack correctly.

5.2.3. Example

The use of the instructions PRINT, COPY and NULL can be seen in the program shown below in Figure 5.1. Let us consider the grammar

<word> ::= *<letter>;

which says that "a word is a sequence of none or many letters". Let us write the code for this grammar and some output instructions. For this example, let the input tape be



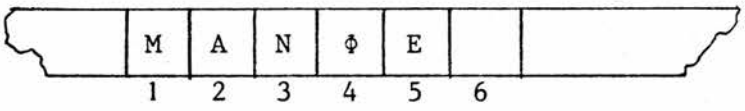
Input tape

We shall write the codes to copy the word "MAN" onto the output tape and then a null string followed by a printed letter say "E". Our code will be as follows:

Label	Function	Address
	CALL	word
	STOP	
L2, word	CALL	
	CALL	letter
	FALSE	L3
	COPY	(copies each letter to the outpu
L3	RETURN	
	FLAG	L2
	NULL	
	PRINT	E
	RETURN	

(Figure 5.1) Use of COPY and NULL instructions

In this case our output tape should look like this



We shall discuss the null symbol, Φ, and its use later in this chapter.

5.3 Editor Machine

We come next to the interpretation of the parser output stream (output tape). This consists of mixed symbols and so-called edit codes. In fact this stream acts as combined data and program for a second hypothetical computer, the editor machine.

The editor machine has its own independent push-down stack. At each level the elements of the stack are variable length ordered sequences or strings of symbols. There is also an output device capable of emitting an ordered stream of symbols.

The operation of the editor machine is initiated from the parser machine when a W edit code is encountered. It is placed in the output vector as other edit codes are but it also indicates that a position has been reached where there will be no backtracking and that the editor machine can now proceed to process the output vector produced so far. At each stage, if the element is a symbol it is placed on the top of the editor stack (as a string), pushing down the previous entries.

5.3.1 Editor Machine Instructions

If an element of the output vector is an edit code the action depends on the particular code. The basic codes are:

X Exchanges the entries in the top two levels of the editor stack

C Combines the two top entries in the stack into one by attaching the head of the top string to the tail of the next, that is, by concatenation.

W Prints the strings in the editor stack in a sequence level by level from the bottom of the stack (note that the null string does not get printed). "outpt" is reset to the beginning of the output vector and the editor stack is emptied. Control returns to the parser machine.

5.3.2 Example

Thus in order to change a string "TOBA" into "BOAT", we make up an input stream by inserting edit codes as follows

T O B X C A C X W

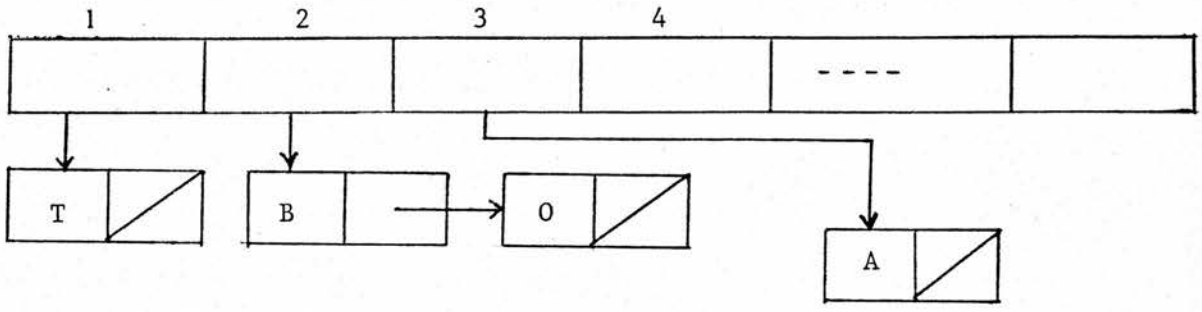
(Note that edit codes are underlined).

The state of editor stack, input (to the editor) and output is shown in Figure 5.2, where ',' is used to separate strings.

State	Input	Editor Stack	Output
0	...	...	
1	T	"T"	
2	O	"T", "O"	
3	B	"T", "O", "B"	
4	X	"T", "B", "O"	
5	C	"T", "BO"	
6	A	"T", "BO", "A"	
7	C	"T", "BOA"	
8	X	"BOA", "T"	
9	W		BOAT

(Figure 5.2) Different States of the Editor Stack

As an illustration, the editor stack, at the State 6 (see Figure 5.2) can be pictured as shown in Figure 5.3



(Figure 5.3) Editor State at Level 6

5.4 Meta-Semantic Language

In this section we link up the production of output from the parser with the meta language by defining additional meta symbols which correspond to the output instructions in the parser program. The correspondence is as follows:

Meta language	Program
£ ABC .... Z £	PRINT A
	PRINT B
	PRINT C
	⋮
	PRINT Z

where ABC .... Z is any sequence of symbols.

Metasymbol	Code produced
K	COPY
O	NULL
W	EDIT W
C	EDIT C
X	EDIT X

5.4.1 Example

Now we are able to translate an input stream into a desired output stream. Let us now write a program to reverse a string of characters. We shall write:

```
<string> ::= <reversed string> W;
<reversed string> ::= O * (<char>KXC);
<char> ::= =A|=B|=C| .... |=Z
```

The mnemonic codes for this grammar are the following:

→	CALL	string
	STOP	
string	CALL	rs
	FALSE	L1
	EDIT	W
L1	RETURN	
rs	NULL	
L2	CALL	
	CALL	char
	FALSE	L3
	COPY	
	EDIT	X

```
        |  EDIT      |  C
L3      |  RETURN    |
        |  FLAG      |  L2
        |  RETURN    |
        |            |
char    |  MATCH     |  A
        |  TRUE      |  L4
        |  MATCH     |  B
        |  TRUE      |  L4
        |   ...      |
        |   ...      |
        |            |
        |  MATCH     |  Z
L4      |  RETURN    |
```

If the input to this program is "RING" (say) the output from the parser will be:

$$\Phi\ R\ \underline{X}\ \underline{C}\ I\ \underline{X}\ \underline{C}\ N\ \underline{X}\ \underline{C}\ G\ \underline{X}\ \underline{C}\ \underline{W}$$

assuming that ϕ represents the null string. At this stage the editor machine is started and when the edit codes are performed we obtain the reversed string as "GNIR". Whenever a $\underline{C}$ operation is performed by the editor - the editor stack should have two (empty or non-empty) strings in order to perform this operation. So a null string is placed on the editor stack to start with to carry out the operation successfully. This also explains the requirement of the NULL instruction in our parser machine.

To reverse a string of any characters a better program could be

```
<string> ::= <reversed string> W;
<reversed string> ::= O * (U K X C);
```

5.4.2 Reverse Polish Example

Now, as another illustration, we consider the translation into Reverse Polish notation of a simple form of arithmetic expression in three variables P, Q, and R. We shall use three edit codes X C C in a sequence in order to convert a string of the form "P+Q" into "PQ+", where P and Q are terms (see below) and +, an operator. The program is as follows:

```
<data> ::= <arithmetic expression>W;
<arithmetic expression> ::= <term>* (<adding operator> K
                                <term>  X C C);
<adding operator> ::= = + | = - ;
<term> ::= <factor>*(<multiplying operator> K
          <factor> X C C);
<multiplying operator> ::= = * | = / ;
<factor> ::= <variable> K | = (<arithmetic expression>=);
<variable> ::= =P| =Q| =R
```

The corresponding program begins as follows:

```
                CALL    data
                STOP

data            CALL    ae
                FALSE   A1
                EDIT    W
A1              RETURN

ae              CALL    term
                FALSE   B1
B3              CALL
                CALL    ao
                FALSE   B2
                COPY
                CALL    term
                FALSE   B2
                EDIT    X
                EDIT    C
                EDIT    C
B2              RETURN
                FLAG    B3
B1              RETURN

ao              MATCH   +
                FALSE   C1
                MATCH   -
C1              RETURN

term            CALL    factor
                FALSE   D1
D3              CALL
```

```
                CALL    mo
                FALSE   D2
                COPY
                CALL    factor
                FALSE   D2
                EDIT    X
                EDIT    C
                EDIT    C
D2              RETURN
                FLAG    D3
D1              RETURN

mo              MATCH   *
                FALSE   E1
                MATCH   /
E1              RETURN

factor          CALL    var
                FALSE   F1
                COPY
F1              TRUE    F2
                MATCH   (
                FALSE   F2
                CALL    ae
                FALSE   F2
                MATCH   )
F2              RETURN

var             MATCH   P
                FALSE   G1
                MATCH   Q
```

	FALSE	G1
	MATCH	R
G1	RETURN	

(Figure 5.4) Translation of a simple arithmetic expression into its Reverse Polish

From this program it can easily be verified that for an input, say,

Q * P + (R - P/Q) + Q / (Q - R)

the output produced will be

QP * RPQ / - + QQR -/+

CHAPTER 6

FURTHER FEATURES

6.1 Integers and Labels

So far, in all our examples of programs for the parser machine we have used mnemonic codes for the function parts of instructions and have made free use of symbolic labels in the address part whenever one instruction referred to the location of another. If we are to succeed in compiling such programs automatically from their meta language form, we need to be more explicit over defining their structure.

We distinguish between the stored form of a program and the so-called assembly language form. These differ in that the assembly language permits symbolic labels and addresses whereas addresses are held in absolute form in the stored form of the program. We shall compile from the meta language form into assembly language form. For the present we discuss only this stage, leaving the description of the assembly process until later.

The assembly language form of a program is an output stream of signed integers rather than characters. The syntax may conveniently be expressed in terms of the meta language with slightly relaxed rules as follows, where explicit integers are within quotation marks indicating, for example, that "999" is one symbol rather than a sequence of three digits.

6.1.1 Syntax of Assembly Language Program in Internal Form

```
<program> ::= <routine> = "999";
<routine> ::= *<instruction>;
<instruction> ::= *<label><function><address>;
```

<label> ::= <internal reference>|<external reference>;

<internal reference> ::= <integer> (with -999≤integer≤-1);

<external reference> ::= <integer> (with integer≤-1000);

<function> ::= <integer> (with 1≤integer≤998);

<address> ::= <integer>;

(Figure 5.1) Informal syntax of programs in assembly language form

The convention is that a negative integer represents a label or symbolic address. If these are ≤-1000 they may represent routine names, otherwise internal labels. The mnemonic codes for the functions are replaced by integers (see Appendix 1) in the range 1 to 998 (in fact only 17 are used). Therefore, if, when reading an instruction, we encounter integers ≤-1000 or "999" before a function part, we know that we have reached the end of a routine or of the complete program respectively. Reeves use "0" or "999" to mark the end of a routine or the complete program. And for this reason he compiles an extra integer "0" at the end of each routine. In our case the encounter of an integer ≤-1000 gives information that a new routine is starting, and that is the reason in Figure 6.1 we changed the definition

<routine> ::= *<instruction>="0";

which Reeves uses, to the definition

<routine> ::= *<instruction>;

External references, which are at the head of routines, correspond to the variable names in the meta language while the internal references correspond to the labels in the parser programs. Both are used for cross references between routines.

An instruction may have several labels, and these precede the function part. A negative address part denotes a symbolic address corresponding to a location in the store with the corresponding label. It is assumed that all absolute addresses will be strictly positive so that zero denotes a null address in CALL or other instructions such as RETURN, COPY, NULL, STOP etc. Further each different symbol in the source and output streams must be identified by a unique integer. The address part of MATCH and PRINT instructions contain integers representing the characters directly.

6.1.2. Editor Instruction Sequences In Numeric Form

It is also necessary to specify the representation in terms of integers of the parser-editor interface.

<interface stream> ::= *(<symbol>|<edit codes>);

<symbols> ::= <integer> (with integer≥0);

<edit code> ::= <integer> (with integer≥0);

(Figure 6.2)

The mnemonic edit codes are replaced by integers (see Appendix I). The null symbol output from the parser by the NULL instruction is represented by the integer "-555555" (Reeves uses the integer 500 000 000 000 for the null symbol output). Now it can easily be seen that the program shown in Figure 5.4 can be written, with the aid of syntax defined above, as follows:

Symbolic Instructions			Numeric Instructions			Rule being programmed
Label	Function	Address Part	Label	Function	Address Part	
	CALL	data		1	-1000	
	STOP			17	0	
data	CALL	ae	-1000	1	-1001	<data>::=<arithmetic expression>W;
	FALSE	A1		4	-1	
	EDIT	W		15	1	
A1	RETURN		-1	2	0	
ae	CALL	term	-100	1	-1002	<arithmetic expression>::= <term>* (<adding operator>K <term>Xcc);
	FALSE	B1		4	-2	
B3	CALL		-4	1	0	
	CALL	ao		1	-1003	
	FALSE	B2		4	-3	
	COPY			13	0	
	CALL	term		1	-1002	
	FALSE	B2		4	-3	
	EDIT	X		15	3	
	EDIT	C		15	2	
	EDIT	C		15	2	
B2	RETURN		-3	2	0	
	FLAG	B3		5	-4	
B1	RETURN		-2	2	0	
ao	MATCH	+	-1003	11	78	<adding operator>::= = + |= - ;
	FALSE	C1		3	-5	
	MATCH	-		11	96	
C1	RETURN		-5	2	0	

term	CALL	factor	-1002	1	-1004	<term>::=<factor>*(<multiplying operator>K<factor>XCC);
	FALSE	D1		4	-6	
D3	CALL		-8	1	0	
	CALL	mo		1	-1005	
	FALSE	D2		4	-7	
	COPY			13	0	
	CALL	factor		1	-1004	
	FALSE	D2		4	-7	
	EDIT	X		15	3	
	EDIT	C		15	2	
	EDIT	C		15	2	
D2	RETURN		-7	2	0	
	FLAG	D3		5	-8	
D1	RETURN		-6	2	0	
mo	MATCH	*	-1005	11	92	<multiplying operator>::= = * | = / ;
	FALSE	E1		3	-9	
	MATCH	/		11	97	
E1	RETURN		-9	2	0	
factor	CALL	var	-1004	1	-1006	<factor>::=<variable>K |=(<arithmetic expression>=);
	FALSE	F1		4	-10	
	COPY			13	0	
F1	TRUE	F2	-10	3	-11	
	MATCH	(		11	77	
	FALSE	F2		4	-11	
	CALL	ae		1	-1001	

	FALSE	F2		4	-11	
	MATCH	)		11	93	
F2	RETURN		-11	2	0	
var	MATCH	P	-1006	11	215	<variable>::= =P|=Q|=R
	TRUE	G1		3	-12	
	MATCH	Q		11	216	
	FALSE	G1		3	-12	
	MATCH	R		11	217	
G1	RETURN		-12	2	0	
			999			

(Figure - 6.3) Assembly Language program for translating simple arithmetic expressions into their Reverse Polish.

The integers ≤ -1000 in the fourth column (label part) of the Figure - 6.3 are nothing but the starts of routines corresponding to where, in the first column, we have used ae, term, factor etc. for the symbolic instructions starting each subroutine. Moreover each integer (≤ -1000) is used uniquely to represent the head of a particular routine and the correspondence is shown in the following table. They are for global use between routines.

Name of head of routines	corresponding integer value
data	-1000
ae	-1001
term	-1002
ao	-1003
factor	-1004
mo	-1005
variable	-1006

(Figure - 6.4)

Table of head of routines and their corresponding integer value

The second and fifth columns in Figure-6.3 have the function parts of the instructions and their corresponding integer representations (see Appendix 1). All the negative integers, apart from the address part of an EDIT instruction are labels or internal references used within a routine. Integers 1, 2 and 3 as the address parts of EDIT instruction are the edit codes for W, X, and C respectively. Finally, a positive number in the address part of an instruction is the internal representation of some character in ALGOL W for the IBM 360/44. For example 215, 216, and 217 are the internal representations for the characters "P", "Q", and "R" respectively.

6.2 USE OF INTEGERS

6.2.1 Distinction Between Integers and Digits

The distinction between an integer and the digits of its representation is an important one and enters the meta language in two ways. Firstly, the compilation of PRINT instructions is extended to permit general integers as well as symbols to be specified as address parts, in the meta language. Thus £(12)£ is compiled as PRINT 12 whereas £12£ is compiled as PRINT 241 followed by PRINT 242 where, as it happens, 241, and 242 are the internal representations of digits 1 and 2 respectively, in the IBM 360/44 computer. In general the syntax of the sequence permitted within pound signs is

*(=(<integer>=)⊢=£)

6.2.2 Two New Edit Codes

The second reference to integers in the meta language is in the action of two new edit codes V and N. These compile the instructions EDIT V and EDIT N or rather their numerical equivalent (see Appendix 1). Their effects are as follows when interpreted by the editor:

V On the assumption that the top level of the editor stack contains a string of digits, it replaces the string by the integer which it represents in the decimal scale, the digit at the head of the stack being the most significant.

N on the assumption that the top level of the editor stack contains a single integer, it changes the sign of this integer.

The use of V code is seen in the full specification of an integer in the meta language

<integer>::= <digit>K*(<digit>KC)V

As successive digits are recognised they are copied and combined with the string already read, and finally the whole string is replaced by the value of the integer.

6.3 Generation of External Labels

We are now ready to discuss the generation of negative integers as labels for the cross references within the parser program. There are two sorts of labels which are called internal and external labels. Internal labels are for local use within a routine and external labels are for global use between routines. It will be seen from the syntax of the compiled program that the internal labels have values x in the range $-999 \leq x \leq -1$ and external ones in $x \leq -1000$.

6.3.1 Edit L

External labels correspond to the names of the variables in the meta language or, equivalently to the locations of the head of the routines in the parser program. This correspondence is set up by means of a further edit code, L in the meta language which compiles as EDIT L. This code is associated with additional storage in the editor machine. Independently of the editor stack is provided a list called "name". This is a list of sub-lists and acts as a dictionary. The first element of each sub-list is an external label (≤ -1000): the second sub-list is a list of symbols making

up a variable name. The list "name" is initially empty. There is a register "index" initially set to -1000. The action of code is as follows:

L On the assumption that the top level of the editor stack contains a string of symbols forming a variable name, scans the list "name" for this name. If found, replaces the entry in the stack, by the corresponding label. If not found, enters the name from the stack in the list "name" with the current value of "index" as external label and replaces the stack entry as before - also reduces the value of "index" by unity. (So that a different numerical label is generated for each name).

The use of the L code is seen in the full specification of a variable in the meta language.

<variable>::= =<UK*(¬=>KC)=>L

The syntax tree differs slightly from that in Chapter 2 but the change is not important.

6.4 Generation of Internal Labels

For generating internal references within routines, we introduce additional functions into the parser instruction code, together with an associated hardware register, "tag", to hold an integer, and a push down list "label" whose elements are values of "tag" at different stages during execution. The extra instructions are as follows:

SETLAB	clears label list
INCLAB	adds current tag setting to label list and increases tag setting by unity.
LABEL	puts out the top entry in the label list followed by N edit code (which is -4)
DECLAB	unstacks the top entry in label list and if flag is false copies into tag, otherwise discards.

For convenience, the four instructions are in fact selected in the program by the address part of a function "REFERENCE" (See Appendix 1). In Reeves' translator the function "REFERENCE" has five cases instead of four as we have in our case and they are the following:

SETLAB	sets tag to unity and clears label list
INCLAB	adds current tag setting to label list and increases tag setting by unity
LABEL	puts out the top entry in the label list followed by N edit code
DECLAB	unstacks the top entry in the label list and if flag is false copies into tag, otherwise discard
RESETLAB	clears label list

The functions SETLAB and RESETLAB are obeyed to clear the "label" list and reset "tag" to unity in order to output different labels in each routine. Each routine has labels in an ascending sequence starting from 1. The existence of common labels among routines may give rise to a confusion when making cross references between routines in the parser program. To avoid this, some changes are made in the grammar for the meta language and are shown in Appendix 2. Changes are made in the definitions of <subject> and <expression>.

SETLAB should be obeyed at the beginning and end of an instruction sequence putting out a routine. INCLAB and DECLAB should be obeyed at the beginning and end of an instruction sequence containing a LABEL instruction corresponding to a particular label value. LABEL is obeyed whenever a label is to be output. Note that the label value can not be put out from the parser directly as a negative number otherwise the editor would treat it as an edit code. It is the purpose of the N edit code to get round this difficulty by negating the integer at edit time . The requirement for one of these instructions in the program is denoted in the meta language by the construction "=R<digit>" as can be seen in the full specification of the meta language in Appendix 2. These four instructions are sufficient to ensure that the correct value of the label is output in the compiled program.

6.5 The Use of Markers:

Metcalfe describes a facility which is helpful in economising the description of a process in the meta language. As an illustration, let us consider the syntax

<group>::= <item><item>;

<item> ::= =A|=B|=C

and suppose that the required output when parsing a group is "D" if the first item of the group is "B", otherwise nothing. It is no good changing item to

<item>::= =A|=B£D£|=C

because this would cause an output whenever "B" appears in either first or second position in the group . We may write

<group>::= <first item><second item>;

<first item>::= =A|=B£D£ |=C;

<second item>::= =A|=B|=C

This solution is unattractive and further it doesn't help in economising the description of the meta language, though it solves the problem. Our preferred solution based on Metcalfe's facility is

<group>::= <item> T1 (£D£) <item>;

<item> ::= =A|=B M1| =C

where T and M are another two meta symbols. Now we shall describe the process and how it works.

The parser is provided with three new instructions and some extra hardware. There is a register "Mark" which can hold a sequence of 32 binary digits. The parser stack is augmented at each level so that the setting of "Mark" can be recorded together with "inpt" and, "outpt" and "pcr".

Existing functions are modified as follows:

CALL	Adds mark into the top of the stack and sets zeroes in mark
TRUE	If flag is false, resets zeroes in mark.
RETURN NOT	Removes records from head of the stack and copies it into mark.

The three new functions are the following:

MARK	Inserts a 1 digit in the bit position specified by the address part, in the record at the head of the stack.
SELECT	Copies the address part into a temporary location called "hold."
TEST	Transfers control (that is copies the address part into pcr) if the digit position of mark specified by "hold" contains 0.

Each activation of a routine has a set of marker bits associated with it in "mark". This record is cleared at entry to the routine. It may be varied by MARK instructions within routines which it calls by resetting to zeroes whenever an alternative is rejected, ready for a fresh start on the next alternative. The current values may be tested by SELECT and TEST instructions. Similarly the routine may itself contain MARK instructions which set values in the record associated with the routine which has called it.

The compiled form of the example at the head of the section is given on the next page.

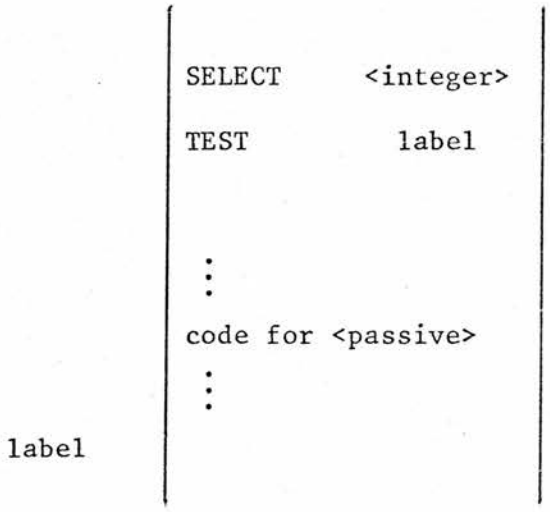
Label	Function (F)	Address (A)	Item being programmed
group	CALL	item	<group>::=<item>
	FALSE	B1	
	SELECT	1	T1
	TEST	B2	
	PRINT	D	£(D)£
B2	CALL	item	
	RETURN		
item	MATCH	A	<item>::= =A
	TRUE	L1	|
	MATCH	B	=B
	FALSE	L2	
	MARK	1	M1
L2	TRUE	L1	|
	MATCH	C	=C
L1	RETURN		

(Figure-6.5) Use of MARK, SELECT and TEST

In the meta language a mark is made by a sequence "=M<integer>". Thus "M3" compiles to "MARK 3". The analogous testing sequence has the syntax

=T<integer>=(*<passive>=)

where <passive> is used in the sense as mentioned in Chapter 3, to denote an element for which flag is known to be true. Such a sequence is compiled as shown in Figure 6.6 overleaf.



(Figure-6.6) Instruction sequence for =T integer =(*<passive>=)

So that the instructions generated by *<passive> are skipped unless the appropriate marker bit has been set.

The full specification of the compiler, written in the meta language itself and so capable of compiling itself, is given in Appendix 2. It will be seen (in Appendix 2) that a grammar is formally delimited by the sequence BEGIN and END and that the following initial (<variable>) is the name of the principal definition. It is this which causes compilation of the program call sequence. Comment facilities are also included. Any thing which fails to satisfy the definition of a definition is ignored upto the next semi-colon separator or the terminal END.

The grammar of the meta language, given in Appendix 2 was initially compiled by hand into assembly language form and punched manually on punched cards. This was checked by compiling itself on the IBM 360/44.

6.6 Assembling the Numerical Form of Programs

The recogniser produced by the editor machine is not quite adequate, yet to run on our hypothetical computer. The address parts of instructions, particularly CALL, TRUE, FALSE and FLAG, are not the actual addresses of instructions or head of routines in the store. In otherwords the address parts of instructions (which are negative integers) should be replaced by the actual addresses of instructions or routines in order to transfer the control at the current location during execution. For example, if we consider the Figure - 6.3, the address parts which are -1000, -1001, -1, -2, etc. (in column 6) are not the actual addresses of instructions which have these as their labels (in column 4).

The assembly of programs is performed by another subroutine written in Algol W, whose input is the code produced by the editor machine and the output is the proper recogniser program which can be run on our hypothetical computer.

This subroutine looks up the actual location of an instruction in the store. This actual location and the corresponding label (which represents the address) of an instruction are placed in a dictionary for further reference. Those instructions which do not have a label (external or internal) are ignored at this point of assembling the numerical form of programs.

The programs are assembled as follows. Firstly, the names of head of routines (in fact the integers ≤ -1000) and their proper location in the stored are entered in the dictionary and are held until the assembly process is finished. Once this done only one routine is considered at a time.

Further entries are made in the dictionary of the internal labels and their actual locations, for the first routine. Now the assembly process of this routine starts. The address parts (only those which are negative integers) of instructions are replaced by their actual addresses in the store. This is achieved by table look-up and using the dictionary.

The same process is repeated for the next routine and so on. Finally the assembly is terminated when at the end of a routine an integer "999" is found which indicated that no more codes are left for assembly. The program in the store is now ready to run on our hypothetical computer.

CHAPTER 7

SASL IMPLEMENTATION

7.1 The SASL Language

In this chapter, a larger problem than has been tried before is described namely - the translation of source statement in the SASL language (Tur-76a) into SASL machine instructions. The name "SASL" stands for "St Andrews Static Language". It was implemented for the purpose of teaching combinatory programming. This is a method of programming which relies on the descriptive features of programming languages (i.e. expressions) rather than their imperative features (i.e. statements or commands).

SASL is in fact a purely static language - it has no goto, no assignment command and no imperatives of any kind. A SASL program is an expression and its outcome is to print the value of the expression. Any properly formed expression is a program. Apart from a finite set of built in functions (+, - etc) SASL allows a user to define functions of his own. These user defined functions can be used to define further functions and so on. In this manner the SASL programmer can express arbitrary complex algorithms in a clean and modular way.

SASL is based on the λ-calculus (chu-51), which is a formal system of representing and manipulating functions. The λ-calculus as it stands is a purely formal language with few but basic facilities. SASL "sugars" this notation by allowing a more convenient syntax to be used by programmers, e.g.

$$\_\text{let } x = Y \quad \_\text{in } Z$$

is a "sugared" form of:

$$(\lambda x.Z)Y$$

where Y and Z are expressions. A function which transforms data objects, is itself a data object in its own right. It is possible therefore, for a function to return as its result another function. Functions-generating functions of this kind (called Combinators) raise programming possibilities that are not normally available to the users of conventional languages.

7.1.1 Examples

For illustrative purposes, some examples of SASL expressions and their values are given below:

Type	Expression	Value
Arithmetic	2 + 3	5
Definition	_let x=3 _in x+2	5
Functional definition	let fx=x+2 _in f3	5
Conditional	2 > 3 → 4; 6 (Equivalent to if 2 > 3 then 4 else 6)	6
List	_hd (1,2,3,)	1
	_t1 (1,2,3,)	2,3,
	1,(2,3,)	1,2,3,
	()	(), the null li
Recursive definition and function producing function	_let map f x = x = () → (); f (_hd x) : map f (_t1 x) _let successor x = x+1 _let modify = map successor (modify is now a function that adds 1 to every element of a list) _in modify (1,2,5,)	2,3,6,

7.1.2. SASL Syntax

In the description of the syntax of the SASL language (see Appendix 3), in the above quoted reference, left recursive definitions are widely used. As explained in Chapter 3, the use of left-recursive definitions, to describe the syntax of a language, should be avoided for top-down translators. For this purpose we shall need to modify the description of the SASL syntax so that it is acceptable by our parser machine. Apart from the modifications in the SASL syntax we shall also specify the output we want to generate - the instructions to a hypothetical SASL machine.

7.2 Modifications in the Description of the SASL syntax:

In general, the following changes are made in the description of the SASL syntax, in order to make it syntactically suitable for the parser machine.

(i) Definitions of the type

<mult-op> ::= *|/|_mod

are replaced by

<MULT-OP> ::= =*|=/|=_=M=O=D;

(ii) To avoid left-recursion and make the parsing process efficient, definitions of the type

<exp-3> ::= <exp-3>&<exp-4>|<exp-4>

should be written as

<EXP-3> ::= <EXP-4>*(=&<EXP-4>);

also the definition

<exp-1> ::= <exp-2>,<exp-1>|<exp-2>,|<exp-2>

may be written as

<EXP-1> ::= <EXP-2> (=,<EXP-1>|=,|);

or

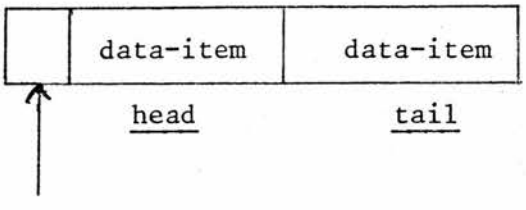
<EXP-1> ::= <EXP-2> (=,(<EXP-1>|)|);

For a complete listing of the altered form of SASL syntax, see Appendix 4.

7.3 Description of the SASL Machine (Tur-76b)

SASL text is first compiled to the machine code of an abstract "SASL machine". This code is then interpreted by a program that simulates the behaviour of the abstract machine.

Since SASL distinguishes between different data types at run time rather than at compile time the SASL machine has a "tagged" architecture. That is, each data item has some extra bits (a tag) in it which says whether it is, for example, an integer, a character or a function. The main memory of the machine consists of a large number of list cells, each of which contains two data items:

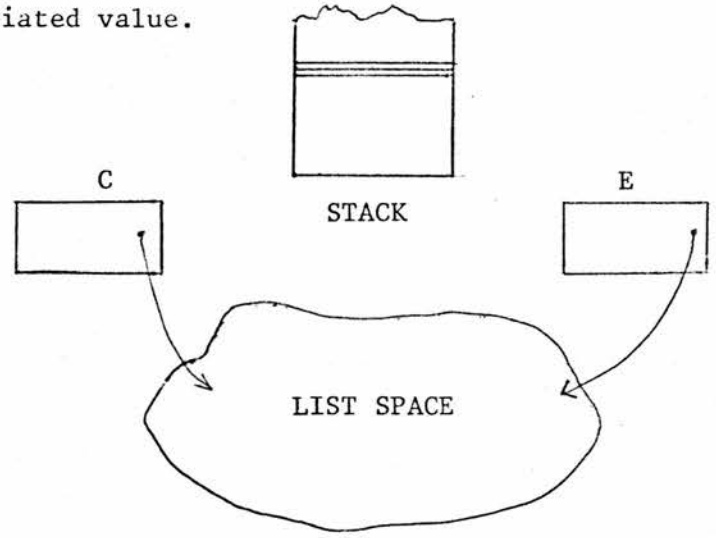


A LIST CELL

tag and extra bit

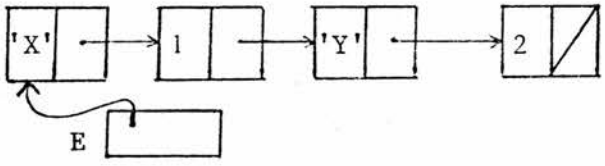
used by garbage-collector.

All data structures are built up from these cells. In addition the machine has a stack (for data items currently in hand) and two registers C (control) and E (environment). C points to the position in code which execution has currently reached. E points to a data structure containing all the names currently in scope and their associated value.



THE SASL MACHINE

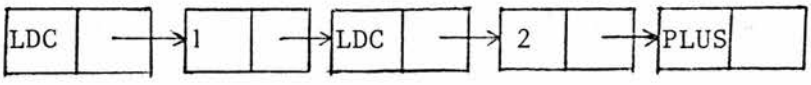
An Environment



where x has been declared with the value 1, and y with the value 2.

The code is also a list structure, in which successive instructions are placed in the heads of a chain list cells, each instruction being immediately followed by its parameters, if any.

Example



C

here the machine is about to execute the code for the expression '1+2'. This could be written in an "assembly language" form as

```
LDC  1
LDC  2
PLUS
```

At the beginning of execution the stack and the environment are empty, and the register C points to the beginning of the code for whole programs. At the end of execution the stack will contain one item, which is the result of the program and must then be printed.

7.4 SASL Machine Instructions

As described in the technical report "An implementation of SASL", a possible SASL machine has two types of instructions. The first type is zero address: PLUS, MINUS, TIMES, AND, OR, HD, TL etc. These instructions act on the top one or two elements of the stack of the SASL machine, removing them and placing the result of the operation on the top of the stack.

The second type of instruction has one address. For example LDC <constant>, where LDC is the instruction and <constant> is its parameter. Following are some of the functions with their description:

LDC <constant>	Places <constant> on the top of the stack.
LOOKUP <name>	Places on the stack the value associated with <name> in the current environment.
DECLGUESS <name>	Makes a new entry in the environment associating <name> with the "don't-know-yet" value, "GUESS".

TIEKNOT <name>	<name> is the current environment with the associated value "GUESS" - removes the top value from the stack and overwrites the "GUESS" with it.
FORK label	Immediately following this instruction the code splits into two branches. Removes the top value from the stack - if it is true "bear-left", if it is false "bear-right".
BLOCK <pointer to code-for block>	Saves the current value of C and E on the stack and transfers control to code for the block.
LOADEN <pointer to code for function body>	Loads the function onto the stack. The object loaded onto the stack is a pointer to a data structure indicating the code for the function body and the environment in which the function was defined.
APPLY	Removes two items from the stack, the first must be a function, the second is its argument. Saves the current values of C and E on the stack. Places the argument back on the stack and sets C and E to be the code and environment parts of the function.
RETURN	Not an explicit instruction, it is invoked automatically when control reaches the end of a list of instructions. This happens when the end of a block or function body is reached or at the end of the whole program.

	It retrieves the old value of C and E from the stack and carries on.
PLUS	On the assumption that the top two stack elements are integers, this performs the addition operation on them. Removes the operands from the stack and stacks the result. May give a run-time error if either element is not an integer.
HD	checks that the item on the top of the stack is a list and replaces it by its head.
TEST IF GREATER	Pushes true or false onto the stack, depending on the relationship between the top two stack elements.

There are many other zero-address instructions not given here explicitly.

7.4 Output of Instructions and Labels

In this section we shall describe the form of output we wish to produce if a source statement or a string is syntactically valid in the SASL language. Let us first consider a simple statement in SASL.

_LET A=2 _IN 3+A

which is a special case of the form

_LET <name>=<expr1> _IN <expr2>

Approximating to the technical report on "An implementation of SASL", (Tur-76b) the output from the parser machine would be the following:

```
        BLOCK  label
        code for <expr1>
        DECL  <name>
        code for <expr2>
        RETURN
    Label:
```

In the example this would be

```
        BLOCK   label
        LDC  2
        DECL  'A'
        LDC  3
        LOOKUP  'A'
        PLUS
        RETURN
    Label:
```

Following are the different forms of source statements in SASL and their equivalent SASL machine instructions as implemented here:

SASL expression <e>	Code for <e>
_LET <name>= <ea> _IN <eb>	BLOCK label code for <ea> DECL <name> code for <eb> RETURN label:

_LAMBDA <name>.<e>	LOADEN label DECL <name> code for <e> RETURN label:
<ea>→<eb>;<ec>	code for <ea> FORK label1 code for <eb> goto label2 label1: code for <ec> label2:
_REC <name>=<ea>_IN<eb>	BLOCK label DECLGUESS <name> code for <ea> TIEKNOW <name> code for <eb> RETURN label:
<constant>	LDC <constant>
<ea><eb>	code for <ea> code for <eb> APPLY
_HD <e> (a typical unary expression)	code for <e> HD

<ea> + <eb> (a typical binary expression)	code for <ea> code for <eb> PLUS
<ea>, <eb>	code for <ea> code for <eb> ,
<ea>,	code for <ea> LDC null ,

Table 7.1 Table of some SASL expressions and their codes

And finally at the end of the program we shall produce an "END", which will indicate the end of the production of instructions.

program ::= <e>	code for <e> . . . END

These represent slight modifications to the original SASL machine.

As an illustration, let us consider a small program in SASL and write SASL machine instructions for it:-

Program:

```
_LET XYZ=3 _IN
XYZ > 2 → 3 ; XYZ + 1
```

SASL Machine instructions:

```
            BLOCK     -1
            LDC       3
            DECL      'XYZ'
            LOOKUP    'XYZ'
            LDC       2
            >
            FORK      -2
            LDC       3
            GOTO      -3
-2:         LOOKUP    'XYZ'
            LDC       1
            +
-3:         RETURN
-1:         END
```

where -1, -2, -3 are labels.

The full specification of syntax is shown in Appendix 4, which accept SASL programs and produces output as SASL Machine instructions. Labels are output as negative integers in a descending sequential order, as has been described in the previous chapter.

CHAPTER 8

CONCLUSIONS

8. Conclusions

A study of syntax-directed translation (the method described by Reeves in particular) has been made where syntax analysis and code generation phases are combined into one. In other words the syntax of the source statements or strings in some source language is specified in extended BNF together with the description of the object codes to be generated when sentences or strings are recognised to be valid in that language. The object language may be the assembly language of some computer or the target language (or machine language) for some hypothetical computer.

The specification language can be extended by adding more semantic functions to it. The translator does not really produce error codes in order to locate syntax error in the source program, but an attempt has been made to discover some syntax errors which can be guessed easily; for example in the syntax and semantic specifications of any source language a name, <variable>, cannot appear on the left hand side of two different definitions; and this is checked at meta-compile time. Also the point of scan is checked to find out whether the input tape was scanned thoroughly.

The possibility of portability exists. And this can be achieved by producing assembly language (or target language) of another computer, and the translator may be used for this purpose. Alternatively the same instructions can be produced but an interpreter for them written for another machine.

There are both advantages and disadvantages for having uncommon labels between routines. It is helpful to produce different labels not

only within routines but within programs as well in order to avoid ambiguity in cross references. On the other hand not more than 998 different labels can be produced which is a limitation but for that to be exceeded the size of grammar would have to be fairly large; again it depends on the use of labels. It was observed, during compilation of the SASL grammar, that not more than 100 labels were used. In any case 998 is an arbitrary limit and can easily be changed.

The language ALGOL W in which the translator was written was found quite suitable for this kind of implementation. The translator program can be used under both the RAX and MFT operating systems available on the IBM 360/44 at St Andrews. Under RAX, a time sharing system, only small grammars can be tried and tested since it allows only 45K words (of 32 bit memory) where most of the storage is taken up by the translator. But under the (batch) MFT system, which allows a maximum of 200K bytes of storage, fairly big grammars (approximately twice as large as the SASL-grammar shown in Appendix 4) may be attempted. The translator takes about 9 seconds to compile itself and produces 404 machine instructions for our hypothetical computer.

A satisfying understanding of the operation of the translator can be attained by working through a few examples. A worthwhile experience may be achieved in learning grammars which can be expressed in extended BNF. We have given sufficiently detailed descriptions of some examples to permit the reader to write grammars of his own. The use of left-recursive definitions must be avoided as stated in section 4.2. The program can be used to learn how to construct grammars for languages in order to teach beginners their concepts.

Finally, it is hoped that some insight of the subject matter and of the implementation of syntax-directed translators has been conveyed in this work. A valuable learning and experience has certainly been achieved by the author.

APPENDIX - 1

PARSER FUNCTIONS AND EDIT CODES

1. CALL
2. RETURN
3. TRUE
4. FALSE
5. FLAG
6. NOT
7. MARK
8. SELECT
9. TEST
10. REFERENCE
    1. SETLAB (R1)
    2. INCLAB (R2)
    3. LABEL (R3)
    4. DECLAB (R4)
11. MATCH
12. NEXT
13. COPY
14. NULL
15. EDIT
    1. W (WRITE)
    2. C (COMBINE)
    3. X (EXCHANGE)
    4. N
    5. L
    6. V
16. PRINT
17. STOP

APPENDIX - 2
GRAMMAR FOR META-LANGUAGE

```
BEGIN
<GRAMMAR>;COMMENT-GRAMMAR-FOR-META-LANGUAGE;
<GRAMMAR>::==B=E=G=I=N<SUBJECT>*(=;(<DEFINITION>^
<COMMENT>))=E=N=D£(999)£W;
<SUBJECT>::=£(1)£<VARIABLE>£(17)(0)£W;
<COMMENT>::=* ¬(=;^=E=N=D);
<DEFINITION>::=R1<VARIABLE>=:=:==<EXPRESSION>
£(2)(0)£WR1;
<VARIABLE>::==<UK*( ¬=>KC)=>L;
<EXPRESSION>::=R2(=R=1£(10)(1)£C^=R=2£(10)(2)£C^0)
<ALTERNATIVE>C*(=^£(3)£R3CCM1<ALTERNATIVE>C)T1(R3C)
(=R=4£(10)(4)£CC^=R=1£(10)(1)£CC^)R4;
<ALTERNATIVE>::=R20*(<PASSIVE>C)(
<ACTIVE>C*(0*(<PASSIVE>C)<ACTIVE>C£(4)£
R3CXCCM1)
0*(<PASSIVE>CM1M2)T2(£(4)£R3CXC)CT1
(R3C)^)R4;
<ACTIVE>::=<VARIABLE>£(1)£XC^==£(11)£UKC
^=(£(1)(0)£C<EXPRESSION>C=)£(2)(0)£CC
^= ¬£(1)(0)£C<ACTIVE>C£(6)(0)£CC;
<PASSIVE>::=R2=*R3<ACTIVE>C£(5)£R3CC
^=£0*((=(<INTEGER>=)^ ¬=£K)£(16)£XCC)=£
^<CODE>£(15)£XC^=0£(14)(0)£C^=K£(13)(0)£C
^=U£(12)(0)£C^=R=3£(10)(3)£C^=M£(7)£
<INTEGER>C
^=T£(8)£<INTEGER>C£(9)£R3CC=(*(<PASSIVE>C)=)
R3CR4;
<CODE>::==N£(1)£^=C£(2)£^=X£(3)£^=N£(4)£
^=L£(5)£^=V£(6)£;
<INTEGER>::=<DIGIT>K*(<DIGIT>KC)V;
<DIGIT>::=0^=1^=2^=3^=4^=5^=6^=7^=8^=9
END
```

Note: The character "^" represents a vertical bar.

APPENDIX - 3

SASL SYNTAX

EXPRESSIONS

<program>::= <exp>

<exp>::= <block>|<λ-exp>|<conditional-exp>|<exp-1>

<block>::= let<defs>in<exp>|rec<defs>in<exp>

<λ-exp>::= λ<formal>.<exp>

<conditional-exp>::= <exp-2>→<exp>;<exp>

<exp-1>::= <exp-2>,<exp-1>|<exp-2>,|<exp-2>

<exp-2>::= <exp-2><or-op><exp-3>|<exp-3>

<exp-3>::= <exp-3>&<exp-4>|<exp-4>

<exp-4>::= ¬<exp-4>|<exp-5><rel-op><exp-5>|<exp-5>

<exp-5>::= <exp-5><add-op><exp-6>|<exp-6>

<exp-6>::= <add-op><exp-6>|<exp-6><mult-op><exp-7>|<exp-7>

<exp-7>::= <ex-op><exp-7>|<combination>

<combination>::= <combination><arg>|<arg>

<arg>::= <name>|<constant>|(<exp>)

DEFINITIONS ETC

<defs>::= <def>|<def>and<defs>

<def>::= <namelist>=<exp>|<function-form>be<exp>

<namelist>::= <formal>|<formal>,|<formal>,<namelist>

<formal>::= <name>|(<namelist>)|()

<function-form>::= <name><formal>|<function-form><formal>

VARIOUS OPERATORS

<or-op>::= |

<rel-op>::= >|>=|=|¬=|<=|<

<add-op>::= +|-

<mult-op>::= *|/|mod

<ex-op>::= hd|to|number|logical|char|list|function|letter|digit|digitval

APPENDIX-4
SASL GRAMMAR WITH SASL MACHINE INSTRUCTIONS

```
BEGIN<PROGRAM>;
<PROGRAM>::=<EXPR>W;
<EXPR>::=<BLOCK>^<LAMBDA-EXPR>^<COND-EXPR>^<E1>;
<BLOCK>::=R2(=_=L=E=T£BLK£CC£(1)£ R3 £(2)£ CC
<DEFS> £(2)£ £LDC£ £(1)£CX  £DECL£ £(1)£ C  CCX £(2)£
C =_=I=N £LK£C £(1)£C  <EXPR> £LDC£ £(2)£C  XCX ^
=_=R=E=C £BLK£CC £(1)£ R3 £(2)£ £DECGS£CCCC £(1)£<DEFS>
£(2)£X £TIEKN£ CCCC £(2)£ )
£RET£CC      R3   R4  ;
<LAMBDA-EXPR>::=R2 =_=L=A=M=B=D=A £LDFN£CCC £(1)£ R3
£DECL£CCC £(1)£ <FORMAL> =. £(2)£ <EXPR> £RET£CC
£(2)£  R3  R4  ;
<COND-EXPR>::=R2<E2>=-=> £(2)£ £FORK£CCC £(1)£ R3
£(2)£CCCCC  <E0> =; R3 £(1)£ <EXPR> £(2)£CCCX
R4;
<E0>::=R2<EXPR> £(2)££GOTO£CCC£(1)£ R3 £(2)£CCCCC
R3 £(2)£ C   R4 ;
<E1>::=<E2>(£(2)£=,K<E1>£(2)£CCX
^ =, K £(2)££(2)££LDC£CC  £(1)££NULL£CCC£(2)£
CCCCX ^ );
<E2>::=<E3>   *(=^<E3>);
<E3>::=<E4>*(=&K<E4>);
<E4>::= =¬ K<E4> ^<E5> (<REL-OP> £R-OP£ <E5> ^  );
<E5>::=<E6>*(<ADD-OP>  £(2)£ <E6> CCX £(2)£C );
<E6>::=<ADD-OP><E6>^<E7>*(<MULT-OP><E7>);
<E7>::=<EX-OP>  £EX-OP£CCCC  £(2)£C  <E7>£(2)£C X
^ <COMB>;
<COMB>::=<ARG>*( £(2)£ <ARG> £(2)£ £APPLY£CCCC
£(2)£ );
<ARG>::=<NAME>^£(2)£ £LDC£ £(1)£CCC <CONSTANT>
£(1)£ ^ =(<EXPR>=);
<DEFS>::=<DEF>(=_=A=N=D<DEFS>^ );
<DEF>::=<NAME-LIST>==<EXPR> ^<FUNC-FORM>=_=B=E<EXPR>;
<NAME-LIST>::=<FORMAL>(=,<NAME-LIST>^ =,^ );
<FORMAL>::= <NAME> ^=(<NAME-LIST>=) ^ =(=);
<FUNC-FORM>::=<NAME><FORMAL>*<FORMAL>;
<REL-OP>::= => ^ =>== ^ == ^ =¬== ^ =<== ^ =<  ;
<MULT-OP>::=  =*K ^ =/K  ^=_=M=O=D  £MOD£ ;
<ADD-OP>::= =+ K ^=- K ;
<EX-OP>::= =_=H=D ^ =_=T=L ^ =_=L=O=G=I=C=A=L ^
=_=C=H=A=R ^
=_=F=U=N=C=T=I=O=N ^ =_=L=E=T=T=E=R ^  =_=L=I=S=T  ^
```

```
=_=D=I=G=I=T ^ =_=D=I=G=I=T=V=A=L;
<CONSTANT>::= <NUMERAL> ^ <LOG-CONST> ^
<CHAR-CONST> ^ <STRING> ^ =(=) ;
<NUMERAL>::=<SP> <DIGIT>K *(<DIGIT>KC ) <SP> ;
<LOG-CONST>::= =_=T=R=U=E   ^   =_=F=A=L=S=E   ;
<CHAR-CONST>::=   =% U   ^ =_=N=L   ;
<STRING>::=   ='   *( ¬ =" ) =" ;
<NAME>::=<LETTER>K*(<LETTER>KC^<DIGIT>KC);
<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;
END
```

Note: The character "^" represents a vertical bar.

APPENDIX - 5

Mnemonic codes for grammar for metalanguage

```
        CALL    GRAM
        STOP
GRAM    MATCH   B
        FALSE   LO1
        MATCH   E
        FALSE   LO1
        MATCH   G
        FALSE   LO1
        MATCH   I
        FALSE   LO1
        MATCH   N
        FALSE   LO1
        CALL    SUBJ
        FALSE   LO1
LO2     CALL
        MATCH   ;
        FALSE   LO3
        CALL
        CALL    DEFI
        TRUE    LO4
        CALL    COMM
LO4     RETURN
LO3     RETURN
        FLAG    LO2
        MATCH   E
        FALSE   LO1
        MATCH   N
        FALSE   LO1
        MATCH   D
        FALSE   LO1
        PRINT   999
        EDIT    W
LO1     RETURN

SUBJ    PRINT   1
        CALL    VAR
        FALSE   S1
        PRINT   17
        PRINT   0
        EDIT    W
S1      RETURN

COMM    CALL
        CALL
        MATCH   ;
        TRUE    C2
        MATCH   E
        FALSE   C2
        MATCH   N
        FALSE   C2
```

```
        MATCH   D
C2      RETURN
        NOT
        FLAG    COMM
        RETURN

DEFI    REF     R1
        CALL    VAR
        FALSE   DE1
        MATCH   :
        FALSE   DE1
        MATCH   :
        FALSE   DE1
        MATCH   =
        FALSE   DE1
        CALL    EXPR
        FALSE   DE1
        PRINT   2
        PRINT   0
        EDIT    W
DE1     REF     R5
        RETURN

VAR     MATCH   <
        FALSE   V1
        NEXT
        COPY
V2      CALL
        CALL
        MATCH   >
        NOT
        FALSE   V3
        COPY
        EDIT    C
V3      RETURN
        FLAG    V2
        MATCH   >
        FALSE   V1
        EDIT    L
V1      RETURN

EXPR    REF     R2
        CALL
        MATCH   R
        FALSE   E1
        MATCH   1
        FALSE   E1
        PRINT   10
        PRINT   1
        EDIT    C
E1      TRUE    E2
        MATCH   R
        FALSE   E3
        MATCH   2
        FALSE   E3
```

```
        PRINT   10
        PRINT   2
        EDIT    C
E3      TRUE    E2
        NULL
E2      RETURN
        FALSE   E4
        CALL    ALTR
        FALSE   E4
        EDIT    C
E5      CALL
        MATCH   |
        FALSE   E6
        PRINT   3
        REF     R3
        EDIT    C
        EDIT    C
        MARK    1
        CALL    ALTR
        FALSE   E6
        EDIT    C
E6      RETURN
        FLAG    E5
        SELECT  1
        TEST    E7
        REF     R3
        EDIT    C
E7      CALL
        MATCH   R
        FALSE   E8
        MATCH   4
        FALSE   E8
        PRINT   10
        PRINT   4
        EDIT    C
        EDIT    C
E8      TRUE    E9
        MATCH   R
        FALSE   E10
        MATCH   5
        FALSE   E10
        PRINT   10
        PRINT   5
        EDIT    C
        EDIT    C
E10     TRUE    E9
E9      RETURN
E4      REF     R4
        RETURN
ALTR    REF     R2
        NULL
AL1     CALL
        CALL    PSV
        FALSE   AL2
        EDIT    C
AL2     RETURN
```

```
        FLAG    AL1
        CALL
        CALL    ACT1
        FALSE   AL3
        EDIT    C
AL4     CALL
        NULL
AL5     CALL
        CALL    PSV
        FALSE   AL6
        EDIT    C
AL6     RETURN
        FLAG    AL5
        CALL    ACT1
        FALSE   AL7
        EDIT    C
        PRINT   4
        REF     R3
        EDIT    C
        EDIT    X
        EDIT    C
        EDIT    C
        MARK    1
ALT     RETURN
        FLAG    AL4
        NULL
AL8     CALL
        CALL    PSV
        FALSE   AL9
        EDIT    C
        MARK    1
        MARK    2
AL9     RETURN
        FLAG    AL8
        SELECT  2
        TEST    L99
        PRINT   4
        REF     R3
        EDIT    C
        EDIT    X
        EDIT    C
L99     EDIT    C
        SELECT  1
        TEST    AL3
        REF     R3
        EDIT    C
AL3     TRUE    L88
L88     RETURN
        REF     R4
        RETURN

ACT1    CALL    VAR
        FALSE   AC1
        PRINT   1
        EDIT    X
        EDIT    C
AC1     TRUE    AC2
        MATCH   =
        FALSE   AC3
        PRINT   11
        NEXT
        COPY
```

```
        EDIT    C
AC3     TRUE    AC2
        MATCH   (
        FALSE   AC4
        PRINT   1
        PRINT   0
        EDIT    C
        CALL    EXPR
        FALSE   AC4
        EDIT    C
        MATCH   )
        FALSE   AC4
        PRINT   2
        PRINT   0
        EDIT    C
        EDIT    C
AC4     TRUE    AC2
        MATCH   ¬
        FALSE   AC2
        PRINT   1
        PRINT   0
        EDIT    C
        CALL    ACT1
        FALSE   AC2
        EDIT    C
        PRINT   6
        PRINT   0
        EDIT    C
        EDIT    C
AC2     RETURN

PSV     REF     R2
        MATCH   *
        FALSE   P1
        REF     R3
        CALL    ACT1
        FALSE   P1
        EDIT    C
        PRINT   5
        REF     R3
        EDIT    C
        EDIT    C
P1      TRUE    P2
        MATCH   £
        FALSE   P3
        NULL
P4      CALL
        CALL
        MATCH   (
        FALSE   P5
        CALL    INT
        FALSE   P5
        MATCH   )
P5      TRUE    P6
        CALL
        MATCH   £
        NOT
        FALSE   P6
```

```
        COPY
        RETURN
        FALSE   P8
        PRINT   16
        EDIT    X
        EDIT    C
        EDIT    C
P8      RETURN
        FLAG    P4
        MATCH   £
P3      TRUE    P2
        CALL    CODE
        FALSE   P9
        PRINT   15
        EDIT    X
        EDIT    C
P9      TRUE    P2
        MATCH   0
        FALSE   P10
        PRINT   14
        PRINT   0
        EDIT    C
P10     TRUE    P2
        MATCH   K
        FALSE   P11
        PRINT   13
        PRINT   0
        EDIT    C
P11     TRUE    P2
        MATCH   U
        FALSE   P12
        PRINT   12
        PRINT   0
        EDIT    C
P12     TRUE    P2
        MATCH   R
        FLASE   P13
        MATCH   3
        FALSE   P13
        PRINT   10
        PRINT   3
        EDIT    C
P13     TRUE    P2
        MATCH   M
        FALSE   P14
        PRINT   7
        CALL    INT
        FALSE   P14
        EDIT    C
P14     TRUE    P2
        MATCH   T
        FALSE   P2
        PRINT   8
        CALL    INT
        FALSE   P2
        EDIT    C
        PRINT   9
        REF     R3
        EDIT    C
        EDIT    C
```

```
        MATCH   (
        FALSE   P2
P16     CALL
        CALL    PSV
        FALSE   P17
        EDIT    C
P17     RETURN
        FLAG    P16
        MATCH   )
        FALSE   P2
        REF     R3
        EDIT    C
P2      REF     R4
        RETURN

CODE    MATCH   W
        FALSE   CD1
        PRINT   1
CD1     TRUE    CD2
        MATCH   C
        FALSE   CD3
        PRINT   2
CD3     TRUE    CD2
        MATCH   X
        FALSE   CD4
        PRINT   3
CD4     TRUE    CD2
        MATCH   N
        FALSE   CD5
        PRINT   4
CD5     TRUE    CD2
        MATCH   L
        FALSE   CD6
        PRINT   5
CD6     TRUE    CD2
        MATCH   V
        FALSE   CD2
        PRINT   6
CD2     RETURN

INT     CALL    DIG
        FALSE   IN1
        COPY
IN2     CALL
        CALL    DIG
        FALSE   IN3
        COPY
        EDIT    C
IN3     RETURN
        FLAG    IN2
        EDIT    V
IN1     RETURN

DIG     MATCH   0
        TRUE    D1
        MATCH   1
```

```
        TRUE    D1
        MATCH   2
        TRUE    D1
        MATCH   3
        TRUE    D1
        MATCH   4
        TRUE    D1
        MATCH   5
        TRUE    D1
        MATCH   6
        TRUE    D1
        MATCH   7
        TRUE    D1
        MATCH   8
        TRUE    D1
        MATCH   9
D1      RETURN
```

BIBLIOGRAPHY

Aho-72 Aho, A.V., and Ullman, J.D. (1972) The Theory of Parsing, Translation and Compiling, Vol. I - Parsing, Prentice Hall, Englewood Cliffs, N.J.

Aho-73 Aho, A.V., and Ullman, J.D. (1973) The Theory of Parsing, Translation and Compiling, Vol. II - Compiling, Prentice Hall, Englewood Cliffs, N.J.

Bac-63 Backus, J.W., et al (1964) "Revised report on the algorithmic language ALGOL 60", Computer Journal, Vol. 5, p. 349.

Bro-62 Brooker, R.A., and Morris, D. (1962) "A general translation phrase structure languages," JACM, 9, pp. 1-10.

Bro-63 Brooker, R.A., et al (1963) "The compiler-compiler", Annual Review in Automatic Programming," Vol. 3, pp. 229-275

Bro-67 Brooker, R.A., Morris, D., and Rohl, J.S. (1967) "Experience with compiler-compiler", Computer Journal, Vol. 9, No. 4, pp. 345-349.

Che-64 Cheatham, T.E., and Sattley, K. (1964) "Syntax-directed compiling", Proc. AFIPS, SJCC, Vol. 25, pp. 31-57.

Che-65 Cheatham, T.E. (1965) "The TGS-II translator-generator system", IFIP, pp. 592-593.

Cho-56 Chomsky, A.N. (1956) "On certain formal properties of grammars", Information and Control, Vol. 2, p. 137.

Chu-51 Church, A. (1951) "The calculi of lambda conversion", Princeton, V.P., Princeton.

DE-56 De Remer, F.L. (1969) "Practical construction for LR(k) Languages", Report, Massachusetts Institute of Technology, Mass.

Eve-63 Evey, R. (1963) "The theory and applications of pushdown store machine", Doctoral thesis, Harvard University.

Fel-64 Feldman, J.A. (1964) "A formal semantics and computer oriented language", Computing Centre, Carnegie Institute of Technology.

Fel-66 Feldman, J.A., (1966) "A formal semantics of computer languages and its application in a compiler-compiler", CACM, Vol. 9, No. 1, pp. 3-9.

Fel-68 Feldman, J.A., and Gries, D. (1968) "Translator writing systems", CACM, Vol. 11, pp. 77-113.

Flo-61 Floyd, R.W. (1961) "A descriptive language symbol manipulation," JACM, vol. 8, pp. 579-584.

Flo-64(a) Floyd, R.W. (1964) "Bounded context syntactic analysis", CACM, Vol. 7, pp. 62-67.

Flo-64(b) Floyd, R.W. (1964) "The syntax of programming languages - a survey", IEEE Trans. EC 13, No. 4, pp. 346-353.

Fis-77 Fisher, R. (1977) "Description of the Revised Compiler-Compiler System", St Andrews University, Scotland.

Gla-69 Glass, R.L. (1969) "An elementary discussion of compiler/ interpreter writing", Computing Surveys, Vol. 1, No. 1, pp. 55-76.

Gri-71 Gries, D. (1971) Compiler Construction for Digital Computers, John Wiley & Sons, Inc.

Han-62 Hanson, J.W., Caviness, J.S., Joseph, C. (1962) "Analytic differentiation by computer", CACM, Vol. 5, No. 6, pp. 349-355.

Iro-61 Irons, E.T. (1961) "A syntax-directed compiler for ALGOL 60", CACM, Vol. 4, pp. 51-58.

Iro-63(a) Irons, E.T. (1963) "Towards more versatile mechanical translators," Proc. of Symposium in Applied Mathematics, Amer. Math. Soc., Vol. 25, pp. 41-50.

Iro-63(b) Irons, E.T. (1963) "The structure and use of the syntax-directed compiler", Annual review in automatic programming, Vol. 9., pp. 207-227.

Joh-72 Johnson, C.S. (1972) "YACC - Yet another compiler-compiler", Bell Laboratories, New Jersey.

Ker-72 Kernighan, B.W. (1972) "Ratfor - A rational Fortran", Bell Laboratories, New Jersey.

Met-64 Metcalfe, H.H. (1964), "A parametrised compiler based on mechanical linguistics", Annual review in automatic programming, vol. 4, pp. 125-165.

Nau-60 Haur, P., et al. (1960) "Report on algorithmic language ALGOL 60" CACM, Vol. 3, pp. 299-314.

Per-65 Perlis, A.J., Iturriaga, R., and Standish, T. (1965) "A preliminary sketch of formula ALGOL", Computing Centre, Carnegie Institute of Technology.

Pol-72 Pollack, B.W. (1972) Compiler Techniques, Auberbach Publishers Inc.

Ree-67 Reeves, C.M. (1967) "Description of a syntax-directed translator", Computer Journal, Vol. 10, No. 3, pp. 244-255.

Rek-72 Rekdal, K., and Wessel, T. (1972) "Generator for SLR(1) - Analysatorer, Report, RUNIT (Norwegian).

Rek-74 Rekdal, K (1974) "Practical construction of syntax-directed translators", Report, RUNIT (Norwegian).

Rit-72 Ritchie, D.M. (1972) "Language C reference manuals", Bell Laboratories, Princeton, N.J.

Sch-62 Schorr, H. (1962) "A syntax-directed translator procedure," TR. No. 25, Department of Electrical Engineering, Digital System Lab., Princeton University.

Sch-63 Schorr, H. (1963) "Analytic differentiation using a syntax-directed compiler," Computer Journal, vol. 7, pp. 290-298.

Tur-76(a) Turner, D. (1976) "SASL language manual", Technical Report CS/75/1, St Andrews University, Scotland.

Tur-76(b) Turner, D, (1976) "An implementation of SASL", Technical Report CS/75/1, St Andrews University, Scotland