TR 82-33

A PARSER GENERATOR SYSTEM TO HANDLE COMPLETE SYNTAX

Thesis submitted in Fulfilment of the

Requirements for the Degree of

MASTER OF SCIENCE

of Rhodes University

by

HAROLD LEON OSSHER

January, 1981

To the memory of my mother

FREDA OSSHER

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to my supervisor, Professor M.H. Williams, for having originated the project here described, for his encouragement and assistance throughout, and particularly for his patience. It was truly a pleasure working under him.

I gratefully acknowledge Dr. J. Bishop's assistance in overcoming the limitations of Pascal. The implementation of strings used is based on a suggestion of hers.

My sincere gratitude is also due to the General Manager and staff of Infoplan (Pty) Limited for making computing facilities available to me and assisting me in their use. But for their kindness and generosity, this project would not have been completed.

ICL (SA) and the Council for Scientific and Industrial Research both assisted in financing the research reported in this thesis. I much appreciate and gratefully acknowledge their generosity.

Finally, I wish to thank my father, Mr Y.I. Ossher, for his invaluable assistance with the preparation of the first draft and the appendices, Bharti Vithal for bravely contending with my handwriting and typing the final draft, and Dr. P.D. Terry and Dr. A.G. Sartori-Angus for fighting the word-processing equipment used.

TABLE OF CONTENTS

1 INTRODUCTION 1

2 SYSTEM OVERVIEW 3

2.1 Complete Syntax 4

2.2 The Specification of Complete Syntax 6

2.3 Parser Requirements 7

2.4 Parser Structure 8

2.5 The Core Parser 10

2.6 The Syntax Interpretation Actions 19

2.7 Parser Error Messages 20

2.8 Parser Diagram 20

2.9 The Definition 20

2.10 The Parser Generator 23

3 THE DEFINITION LANGUAGE 26

3.1 Symbols and Symbol Declarations 27

3.2 Declarations 30

3.3 Types and Type Declarations 31

3.4 Variables and Variable Declarations 41

3.5 Procedures and Procedure Declarations 44

3.6 Expressions 44

3.7 Statements 48

3.8 Syntax Form Rules 60

3.9 Syntax Interpretation Rules 68

3.10 Extensions and Improvements 68

4 THE GENERATED PARSER 70

4.1 Symbols 72

4.2 Declarations 75

4.3 Types and Type Declarations 75

4.4 Variables and Variable Declarations 80

4.5 Procedures and Procedure Declarations 82

4.6 Expressions 82

4.7 Statements 83

4.8 Syntax Form Rules 85

4.9 Syntax Interpretation Rules 85

4.10 Communication between Passes 86

4.11 The Core Parser 86

4.12 The Lexical Analyser 94

4.13 The First Pass 97

4.14 Subsequent Passes 97

4.15 Extensions and Improvements 98

5 THE PARSER GENERATOR 99

5.1 Internal Structures 100

5.2 The Lexical Analyser 103

5.3 The Syntax Analyser 104

5.4 The CFSM Generator 108

5.5 The Macro Generator 116

5.6 The Generation of GLOBALTABLEFILE 126

5.7 TABLEFILE 126

5.8 SKELTNFILE 129

5.9 Extensions and Improvements 129

6 TESTING AND CONCLUSIONS 132

6.1 Two Pass Parser 132

6.2 One Pass Parser 134

6.3 Syntax and Semantics of Constructs 134

6.4 Conflicts 135

6.5 Parser Generator Error Handling 135

6.6 Definition Semantic Errors 135

6.7 Conclusions 136

REFERENCES 137

APPENDIX 1: DEFINITION LANGUAGE KEYWORDS 139

APPENDIX 2: BNF DEFINITION OF THE DEFINITION LANGUAGE 140

A2.1 The Definition 140

A2.2 Declarations 140

A2.3 Syntax Form Rules 141

A2.4 Syntax Interpretation Rules 142

A2.5 Statements 143

A2.6 Expressions 143

A2.7 Symbols 145

APPENDIX 3: ERROR MESSAGES 146

A3.1 Definition Syntax Errors 146

A3.2 LR Parser Generation Errors 151

A3.3 Definition Semantic Errors 151

APPENDIX 4: MACRO GENERATOR CALLS, OPERATIONS AND ARGUMENTS 153

A4.1 Calls Issued by the Syntax Analyser 153

A4.2 Operations 160

A4.3 Collections 161

A4.4 Generate Procedures 162

A4.5 Include Options 162

A4.6 Pseudo-operations 163

A4.7 Arguments 164

A4.8 Constants 164

A4.9 Variable Values 167

A4.10 Modifiers 169

APPENDIX 5: PROGRAMS Appended

A5.1 The Parser Generator

A5.2 TABLEFILESETUP

A5.3 SKELTNFILESETUP

APPENDIX 6: TABLEFILE Appended

A6.1 Source Form

A6.2 Final Form

APPENDIX 7: SKELTNFILE Appended

A7.1 Source Form

A7.2 Final Form

APPENDIX 8: TEST RUNS Appended

A8.1 Two Pass Parser

A8.2 One Pass Parser

A8.3 Syntax of Constructs

A8.4 Syntax and Semantics of Constructs

A8.5 Conflicts

A8.6 Parser Generator Error Handling

A8.7 Definition Semantic Errors

1. INTRODUCTION

The definition of programming languages and the automatic generation of compilers have long interested Computer Scientists. A good, formal definition system greatly facilitates the communication of language concepts between people, and is a prerequisite for automatic compiler generation. Compiler generators can eliminate much tedious and error-prone compiler writing, particular during the design stages of a new language.

To define a language completely, it is necessary to define both its syntax and semantics. If these definitions are in a suitable form, the parser and code-generator of a compiler, respectively, can be generated from them. This thesis addresses the problem of syntax definition and automatic parser generation.

As pointed out by Williams in [15], a complete syntax definition consists of rules of two types, viz:

(a) syntax form rules specifying the form of valid strings in a language; and

(b) syntax interpretation rules (also referred to as context-sensitive or static semantic rules) concerned with the interpretation of symbols within such strings.

Backus-Naur Form (BNF) [3] is an established and convenient notation for expressing syntax form rules. There is, however, no accepted formal notation for expressing syntax interpretation rules: they are usually phrased informally in English.

Various formal systems do exist for defining complete syntax, e.g. canonic systems [12] and two-level grammars [14], but both of these are highly complex and unsuited to parser generation. Work is in progress on attribute grammars [10], but at the time of writing, no

results are available. Hence, though many parser generators are in existence [7, 11, 13], as far as can be determined, none uses a formal specification of syntax interpretation rules. The present project was to develop one.

The definition language used is based on that designed by Williams [15, 16]. It consists of syntax form rules specified in BNF (with a few extensions), and syntax interpretation rules specified formally as actions to be taken by the parser on the recognition of program constructs.

A parser generator was developed which constructs parsers automatically from such definitions, and was successfully used to generate a few test parsers. Both the parser generator and the generated parsers are machine-independent Pascal programs of reasonable size and acceptable efficiency.

Of course, a parser generator alone is not nearly as useful as a full compiler generator. The present system was designed with the intention of coupling the parsers it generates and code generators produced by the code generator generator developed by Bulmer [4]. This remains a project for the future, but should be greatly facilitated by the flexibility of both the parsers' output and the code generators' input.

Chapter 2 of this thesis is an overview of the parser generator system. The definition language is described in detail in Chapter 3, the generated parser in chapter 4 and the parser generator program itself in chapter 5. Chapter 6 describes the main test runs performed, and the conclusions drawn from them.

2. SYSTEM OVERVIEW

The basic function of the parser generator is to accept a complete definition of the syntax of a programming language, and from it to generate a parser for that language, as illustrated in fig. 2.1(a). The listing is a copy of the definition together with messages indicating errors in it. The parser, in turn, will accept a program, PROG, in the language defined, parse it, and generate output related to the parse on GENFILE, for input to a code generator. This is illustrated in fig. 2.1(b). The listing in this case is a copy of the program, together with messages indicating errors in it.

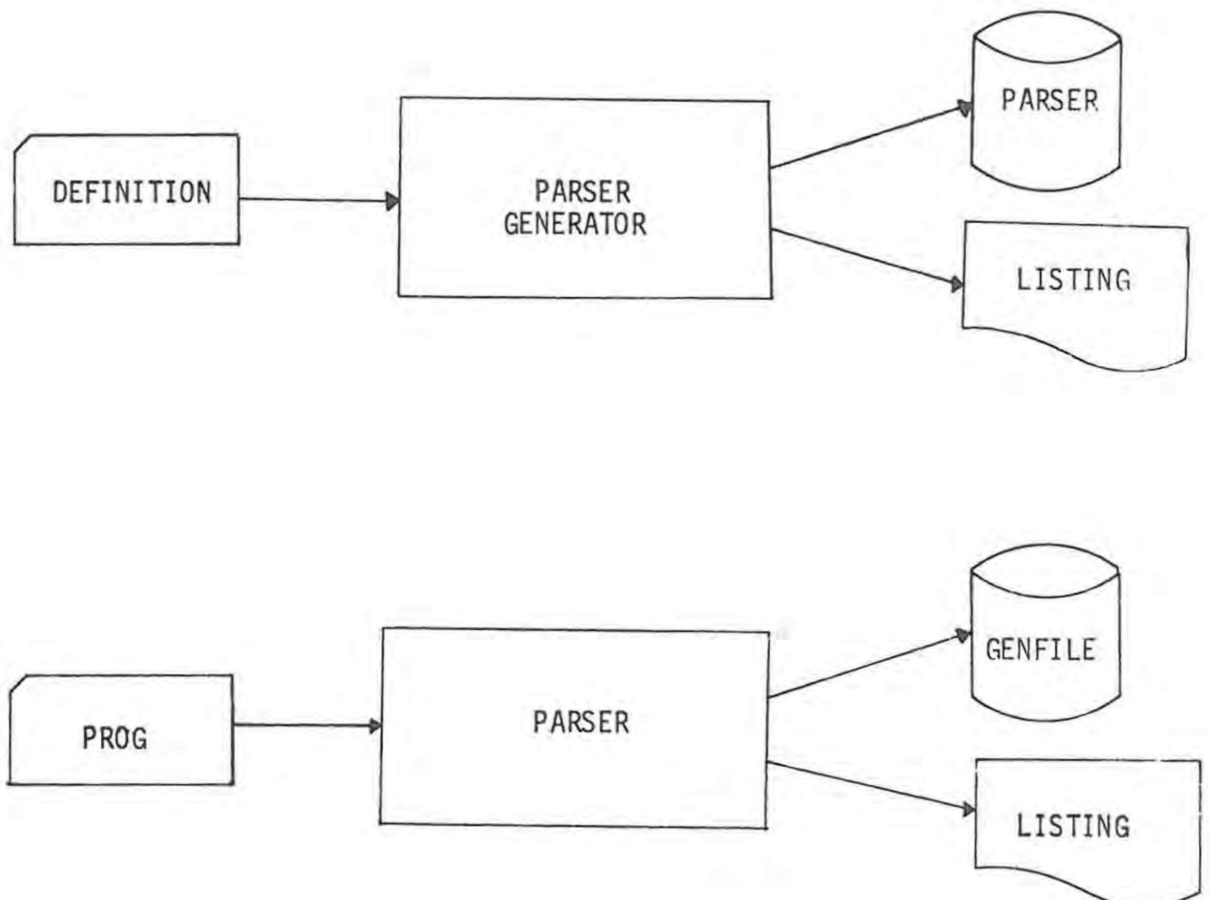

Fig. 2.1: (a) Parser Generator Overview; (b) Parser Overview

This chapter describes the concepts on which the parser generator system is based, and outlines its structure, under the following headings:

(2.1) Complete Syntax

(2.2) The Specification of Complete Syntax

(2.3) Parser Requirements

(2.4) Parser Structure

(2.5) The Core Parser

(2.6) The Syntax Interpretation Actions

(2.7) Parser Error Messages

(2.8) Parser Diagram

(2.9) The Definition

(2.10) The Parser Generator

The definition, parser and parser generator are described in detail in chapters 3, 4 and 5, respectively. Throughout, the reader is assumed to have a knowledge of context free grammars and their associated terminology [2], of Backus-Naur Form (BNF) [3], of Pascal [9] and of finite state machines and transition graphs [2].

2.1 COMPLETE SYNTAX

To perform its functions of parsing a program in a language, a parser must be able to recognise all strings which constitute valid programs in that language, and to determine their structure. It must also be able to reject all strings which are not valid programs in that language, and produce appropriate error messages.

It is therefore concerned, in the first instance, with the form of a program, i.e. with what symbols are valid in a program, and what arrangements of these symbols are permissible. For example, that an

assignment statement consists of a variable, followed by the symbol ":=", followed by an expression. Rules such as this are referred to here as *syntax form rules*, since they are syntactic rules describing the form of a program.

Syntax form rules are commonly expressed in BNF, for example:

<assignment statement> ::= <variable> := <expression>

Each rule specifies the form of each string in a particular class of strings which can occur within a program (in this case, assignment statements). Each string in such a class is called a *construct* and a construct described by means of a BNF rule with the symbol <*l*> on the left side is termed an *<l>-construct*.

Since each BNF rule is merely a context-free production, the body of rules describing the form of a program is a context-free grammar. Not all strings derivable from the BNF rules, and hence having the correct form, are necessarily valid programs, however. For example, in many languages, the above assignment statement would be valid only if the variable and all symbols in the expression were previously declared, and of acceptable and compatible types. Conditions such as this cannot be specified in BNF, as they are not context-free. They are usually stated as rules loosely phrased in English, and associated with the corresponding BNF rules. These rules are variously referred to as *static semantic rules*, *context sensitive rules* and *syntax interpretation rules*; the latter term is used throughout this thesis, as the rules are indeed syntactic, yet are concerned with the interpretation of constructs in a language rather than with their form.

Syntax form and syntax interpretation together are termed complete syntax, which any complete parser must be able to handle.

2.2 THE SPECIFICATION OF COMPLETE SYNTAX

A language definition used in a parser generator system which handles complete syntax must meet the following basic requirements:

(a) Both syntax form and syntax interpretation rules must be specified in a formal manner.

(b) The automatic generation of a parser from the definition must be possible, and preferably convenient and efficient.

In the light of these requirements, some alternative definition methods are now discussed.

The standard method of expressing syntax form rules is as a BNF grammar. This has the advantage that BNF is well known, and that several techniques exist for generating parsers automatically from BNF grammars [1, 5, 8].

There is currently no such standard method for expressing syntax interpretation rules, which are usually stated informally in English. Two methods which combine the definition of syntax form and syntax interpretation are two-level grammars [14] and canonic systems [12]. Both of these, however, are so complex that automatic parser generation based on them is extemely difficult, if not impossible. Furthermore, the relationship between the structure of a definition in one of these forms and the structure of the corresponding parser is obscure.

A more recent definition method is attribute grammars [10], and work is apparently being done on using them as the basis for parser generation. At present, however, no published results on this work are available.

The method used in the present system is based on that of Williams [15, 16]. Syntax form is defined in BNF as usual, but syntax interpretation rules are specified formally as actions to be performed on recognition of constructs. Further details of this method, and its advantages, are given in chapter 3.

2.3 PARSER REQUIREMENTS

The primary requirement of a generated parser is that it handle complete syntax. The manner in which this is done is described in the sequel. It is also desirable that it be readable and easily modifiable, to allow the user to study, augment and modify it, if he wishes. To be of general use, it should also be transportable.

To satisfy these secondary requirements, the parser must clearly be in a high-level language. Pascal was chosen, because it is particularly well-suited to clear, structured programming, and because of the wealth of data types it provides. Structure, modularity, meaningful names and comments serving as headings are used in the usual way to enhance readability and modifiability.

2.4 PARSER STRUCTURE

The structure of a parser generated by the present system is determined by the manner in which complete syntax is handled. The parser consists of three major components:

(a) A lexical analyser which recognises individual program symbols. It is not generated by the parser generator, however, and must be handwritten and included in the generated parser by the user.

(b) A core parser which accepts symbols from the lexical analyser and recognises constructs according to the syntax form rules.

(c) Syntax interpretation actions which check that constructs recognised conform to the syntax interpretation rules, and produce output for the code generator.

Most parsers, whether handwritten or automatically generated, have this structure, though the code for two or more of the components may be interwoven.

Syntax interpretation actions invariably involve building up and accessing symbol tables describing properties of various symbols. In the case of many languages, it is desirable, or even essential, to split the operation of the parser into phases, called passes, as follows:

(a) During the first pass, the core parser recognises all constructs, and the syntax interpretation actions build up all symbol tables.

(b) During subsequent passes, further syntax interpretation actions check the validity of the constructs recognised during the first pass by consulting the symbol tables, and generate output for the code generator.

Each pass, i, is dealt with by a separate routine, PASSi, in the parser. PASS1 contains the lexical analyser and core parser, since they are used only during the first pass, and syntax interpretation

actions. The other passes contain syntax interpretation actions only. In addition to these routines, there is a global area containing data structures and routines used during all passes, such as the symbol tables just mentioned. The parser therefore has the structure shown in fig. 2.2.

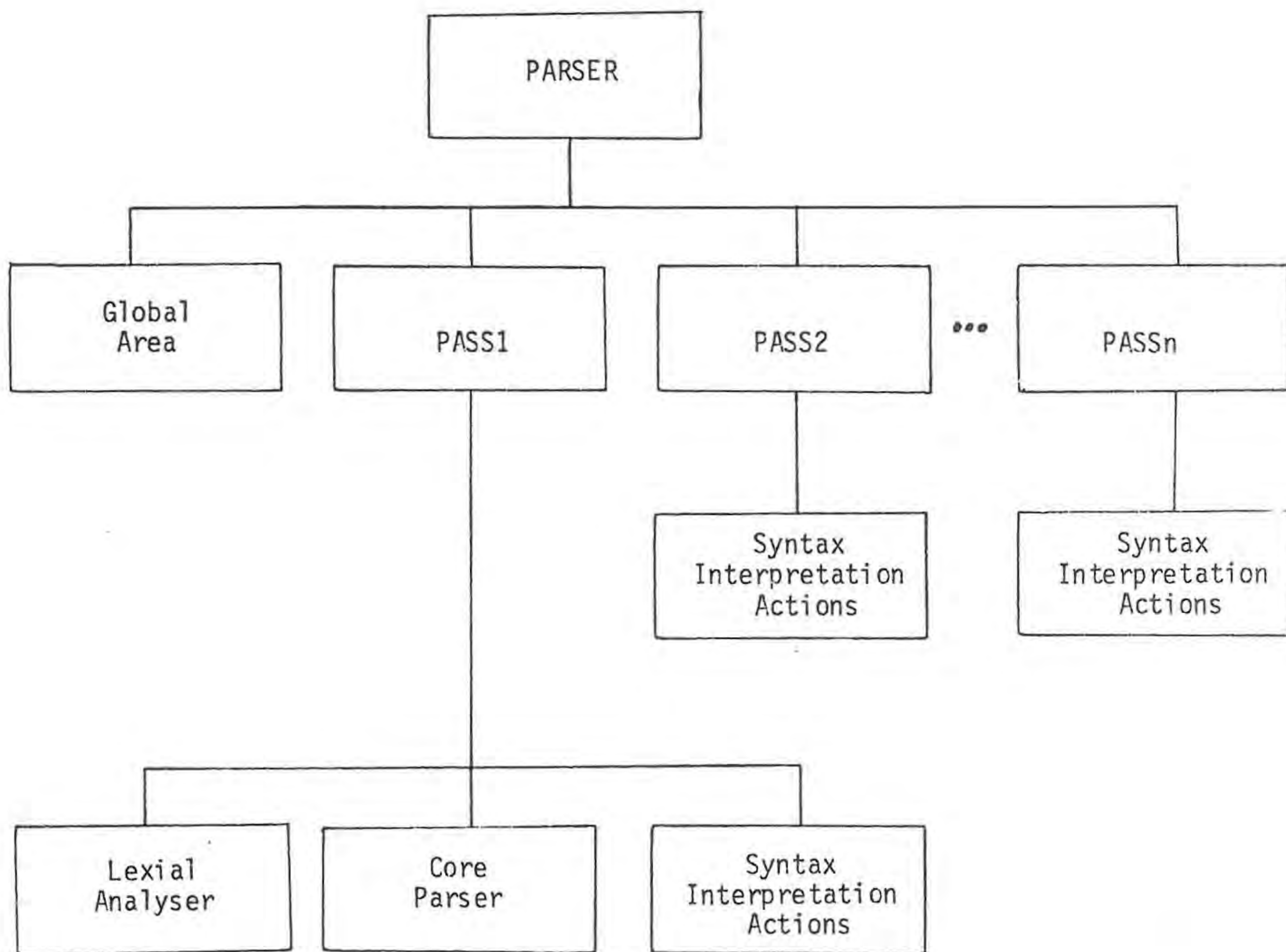

Fig. 2.2: The Structure of a Parser

Pass 1 syntax interpretation actions are invoked directly by the core parser; whenever a construct having an associated pass 1 syntax interpretation action is recognised, that action is performed. This is not possible in the case of subsequent passes, however, as all constructs are recognised during pass 1. Instead, the core parser outputs a partial parse containing details of each construct recognised which has a pass *i* syntax interpretation action, to the disc file PASS*i*PARTIALPARSE. By consulting this partial parse, PASS*i* determines what actions to perform.

The core parser and syntax interpretation actions are described briefly in the next two sections.

2.5 THE CORE PARSER

The function of the core parser is to receive symbols from the lexical analyser, and to group them into constructs according to the syntax form rules. Many standard types of parsers exist which can do this for various classes of languages, such as recursive descent, LL, LR, precedence and operator precedence parsers. For various reasons, listed in section 2.5.4, an LR parser was used in the present system ("L" stands for "left to right scan of the input", "R" for "producing a right canonical parse").

There is actually a whole class of LR parsers; the present system uses an extended version of the simplest one, viz. LR(0). The basic functioning of an LR(0) parser is described in section 2.5.1. For further details and the underlying theory, the reader is referred to [1,2,5,6]. Section 2.5.2 describes the extension used, and mentions the other types of LR parsers, which are also fully described in

[2,5]. Section 2.5.3 is an outline of how the core parser is implemented.

2.5.1 LR(0) PARSERS

An LR(0) parser consists of two elements:

(a) A modified finite state machine, referred to as the characteristic finite state machine (CFSM), consisting of states.

(b) A state stack. The state at the top of the stack is referred to as the current state.

The CFSM specifies for each state a parsing action to be taken on each permissible input symbol. These actions involve the state stack, which is used to keep track of paths followed through the machine, and are as follows:

(a) shift n:

Read the current input symbol and advance to state n, pushing n onto the state stack.

(b) reduce p:

Reduce by production p. If p has the form

$$\langle S \rangle ::= S_1\ S_2\ \ldots\ S_i$$

then pop i entries off the state stack, revert to the state now at the top of the stack, and undergo a transition from that state under the symbol <S> (such a transition will always be possible).

(c) accept:

Accept all symbols read as a complete program, and terminate parsing.

The CFSM of an LR(0) parser can be illustrated as a transition graph as follows:

(a) A shift action is shown as an ordinary transition.

(b) The action reduce p is shown as a transition under a special symbol, $\#_{\underline{p}}$, to a terminal state. Transitions under non-terminal symbols after reductions are shown naturally.

(c) An accept action is indicated by a transition under $\#_{\underline{0}}$ to a terminal state.

(d) The special terminal states are drawn as squares, the other states as circles.

(This transition graph is actually an acceptor for the so-called characteristic strings of the LR(0) grammar. See [5]).

To illustrate the above, consider the simple language consisting of variable length lists of As and Bs, separated by commas. A BNF grammar specifying the language is:

1.	<list> ::= <element>	(production 1A)
	| <list>, <element>	(production 1B)
2.	<element> ::= A	(production 2A)
	| B	(production 2B)

The corresponding CFSM is illustrated in fig. 2.3. The end-marker "⊣" represents the end of the list (i.e. the end of the parser file PROG).

The initial state of the CFSM is state 1. The parser starts off with just the initial state on the state stack, and the first symbol in the list to be parsed as the current input symbol. It then repeatedly performs the action specified for the current state and input symbol, until an accept action is performed. If at any stage no action is specified for the current state and input symbol, it takes an error action, which involves flagging the error and attempting to recover.

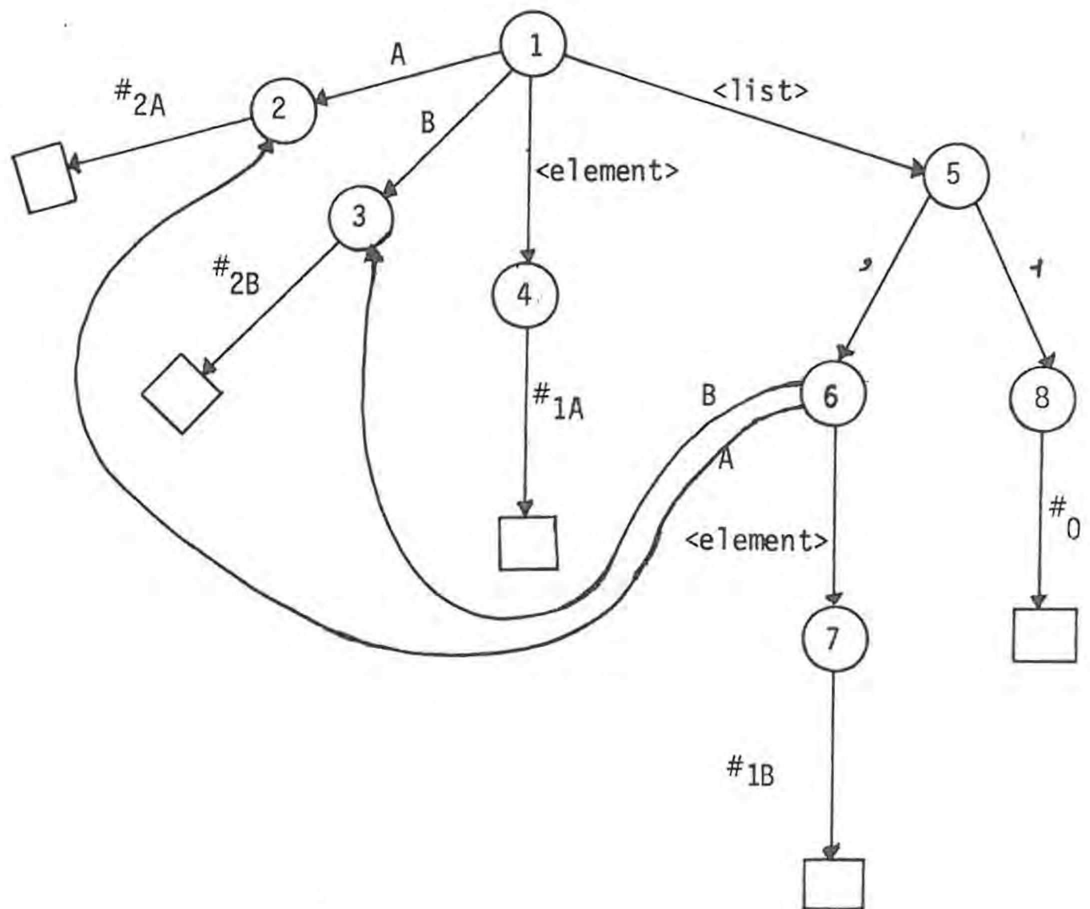

Fig. 2.3: A CFSM to Parse Simple Lists

Fig. 2.4 illustrates the parsing of the simple list

B,A

An LR(0) CFSM state is of one of two types, viz:

(a) A shift state having only shift actions, such as states 1, 5 and 6 in fig. 2.3.

(b) A reduce state having a single reduce or accept action as its only

CURRENT INPUT SYMBOL	STATE STACK	ACTION
B	1	shift 3
,	1 3	reduce 2B
,	1 4	reduce 1A
,	1 5	shift 6
A	1 5 6	shift 2
⊣	1 5 6 2	reduce 2A
⊣	1 5 6 7	reduce 1B
⊣	1 5	shift 8
	1 5 8	accept

Fig. 2.4: The Parsing of a Simple List

action, such as states 2, 3, 4, 7 and 8 in fig. 2.3. It is a property of LR(0) parsers that no state has more than one reduce action, or a combination of shift and reduce actions. This property has the important consequence that the current state of the CFSM alone specifies when a reduce action must be performed, and by which production. Neither the current nor any other input symbol need be consulted by the parser in choosing a reduce action.

In common parser terminology, if a parser consults _k_ input symbols in choosing a production by which to perform a reduction, it is said to use _k-symbol lookahead_. An LR(0) parser, therefore, uses 0-symbol lookahead, which is the significance of the "0" in its name.

Any language for which an LR(0) CFSM, having the properties described above, can be constructed, can be parsed by an LR(0) parser [5], and is termed an _LR(0) language_.

2.5.2 EXTENSIONS TO LR(0) PARSERS

Unfortunately, few languages are LR(0). When parsing programs in many languages, it is necessary to "look ahead" in order to determine whether a shift or reduce action must be performed, or which of several possible reduce actions applies. Such languages simply cannot be parsed by LR(0) parsers.

Even if a language is not LR(0), a CFSM can be constructed for it. However, it will violate the important property mentioned in the previous section: at least one state will have more than one reduce action, or a combination of shift and reduce actions. Such clashes of possible actions are termed conflicts, and the states at which they occur, inadequate or conflict states.

As an example, consider the following BNF grammar for simple expressions:

1.	<expression>	::=	<term>	(production 1A)
		|	<expression> + <term>	(production 1B)
2.	<term>	::=	<number>	(production 2A)
		|	<term> * <number>	(production 2B)

where <number> is assumed to be a basic symbol recognised by the lexical analyser (called a pseudoterminal). The corresponding CFSM is shown in fig. 2.5, in which state 3 is a conflict state.

Suppose this CFSM were used in an attempt to parse the expression

1*2+3

in the manner described in section 2.5.1. Parsing would proceed as illustrated in fig. 2.6, until state 3 were reached. The LR(0) parser would then be unable to decide between the possible shift 4 and reduce 1A actions. By inspection of the grammar and expression,

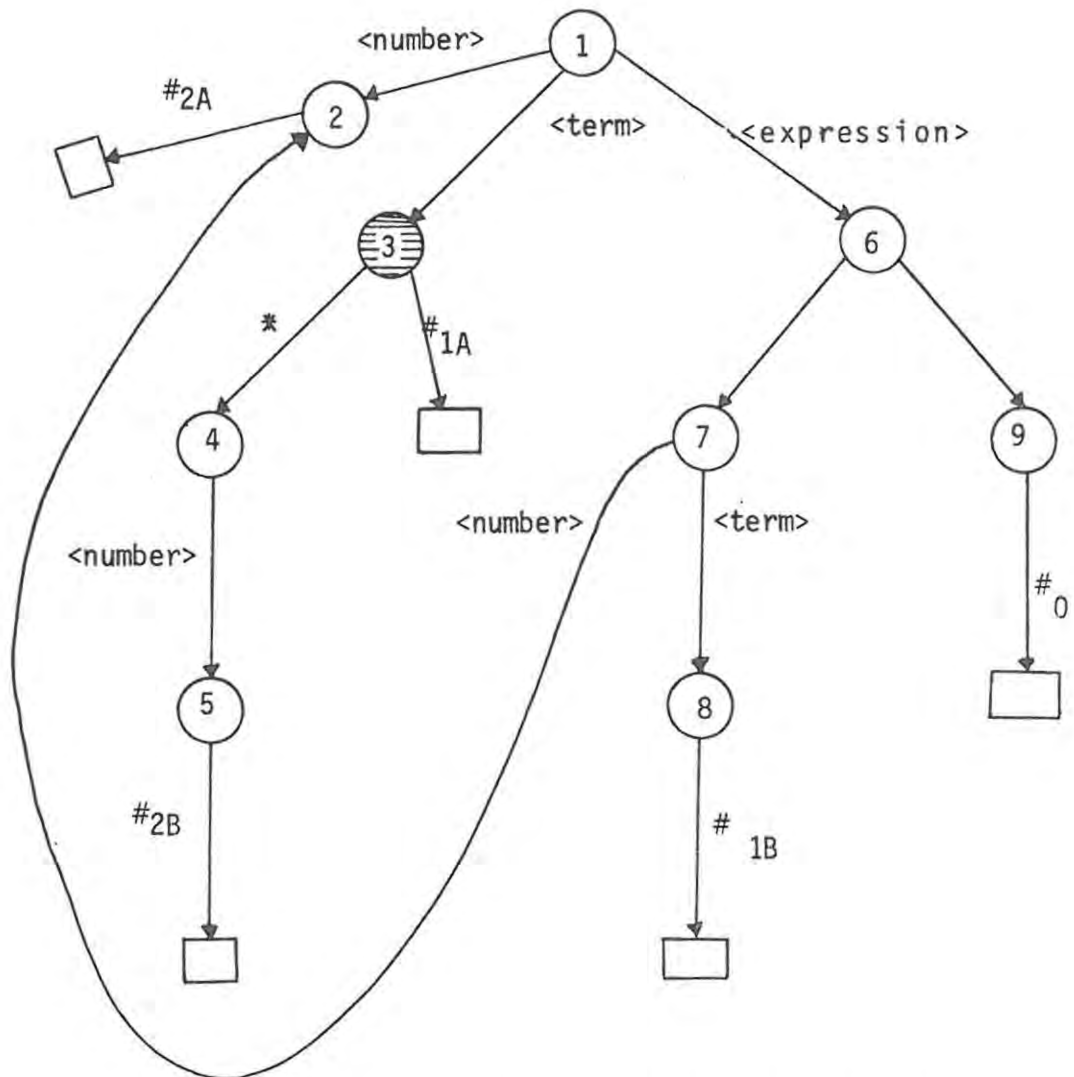

Fig. 2.5: A CFSM to Parse Simple Expressions

however, it can be seen that, since the current input symbol is "*", the shift action is the appropriate one. This illustrates the important principle that many conflicts can be resolved by means of lookahead.

CURRENT INPUT SYMBOL	STATE STACK	ACTION
1	1	shift 2
*	1 2	reduce 2A
*	1 3	?

Fig. 2.6: The Parsing of a Simple Expression

Modifying a CFSM to handle lookahead involves making reduce actions conditional on what input symbol or symbols are about to be scanned. What lookahead is appropriate for a particular reduce action, and hence will resolve a particular conflict, can often be determined by close examination of the grammar. Established techniques exist for doing this, which give rise to SLR(k) or LALR(k) parsers [1,2,5,6], depending on how narrowly they restrict the permissible lookahead for a reduction. _k_ is the number of lookahead symbols used, which must be fairly small and is most commonly one.

Some conflicts cannot be resolved by means of _k_-symbol lookahead, even for arbitrary _k_, and require _splitting_ certain CFSM states to create more distinct paths through the machine. (Whereas lookahead introduced examination of _right context_ to an LR(0) parser, this increases the parser's memory of _left context_). Once again, established techniques exist for determining what state-splitting is required from the grammar alone, and give rise to general LR(k) parsers [5].

Another approach to conflict resolution was adopted in the system being described: by means of _qualifiers_ embedded in the BNF grammar, the user specifies explicitly under what conditions reduce actions must be performed. Qualifiers are required only for productions involved in conflicts. The condition in a qualifier may be a _lookahead test_ on one lookahead symbol, or may involve tables built up during syntax interpretation actions, or any combination of these. Qualifiers are described in detail in sections 3.8.4 to 3.8.7.

2.5.3 IMPLEMENTATION

The core parser in the present system, then, is an LR(0) parser incorporating user-defined qualifiers. The CFSM is implemented, as is common, by means of two tables which together are equivalent to the transition graph representation. These tables are:

(a) The parsing action table specifying the actions associated with each state.

(b) The goto table specifying the state transitions immediately after reductions.

The goto table thus specifies all transitions under non-terminal symbols in the CFSM, and the parsing action table all other transitions.

In the present system, the parsing action table is built into the parser program as the routine PARSINGACTION. The goto table, however, is implemented as a data structure. The parser obtains goto table entries from the file PASS1TABLEFILE, created by the parser generator.

The error recovery part of an error action is by no means trivial, and no accepted, totally satisfactory method exists for handling it. The method used in the present system is described in section 4.11.7. At present, suffice it to say that error recovery tables are used, which are also set up on PASS1TABLEFILE by the parser generator.

2.5.4 ADVANTAGES OF LR PARSERS

LR parsers have the following advantages, which were instrumental in their choice as core parsers in the present system:

(a) They are fairly small and very efficient.

(b) The parsing action table can easily and naturally be expressed in

a high-level language in readable and easily modifiable form.

(c) Established, efficient techniques exist for generating LR(0) parsers automatically from BNF grammars.

(d) An established technique exists for adding computation of lookahead and state-splitting to the basic LR(0) generation algorithm, to facilitate the generation of SLR, LALR or full LR parsers in future.

2.6 THE SYNTAX INTERPRETATION ACTIONS

As already mentioned, the syntax interpretation actions check the validity of constructs recognised by the core parser, and this invariably involves building up and accessing symbol tables. Both the structure of the tables and the details of their manipulation are defined by the user in the definition, as will be described in detail in chapter 3.

Several data types are available for use in tables and related variables, viz. integers, booleans, flags, strings, records, stacks and trees, and a variety of operations are available for manipulating them. All are represented as Pascal data structures, and manipulated by means of Pascal statements and routines; each syntax interpretation action is itself a single Pascal routine. All declarations, statements and routines are built into the parser program, with the exception of string literals, which are obtained from the file GLOBALTABLEFILE created by the parser generator.

2.7 PARSER ERROR MESSAGES

Generated parsers detect and report four classes of errors:

(a) Illegal symbols occuring in the program.

(b) Syntax form errors, which arise when no parsing action exists for the current state and input symbol.

(c) Syntax interpretation errors, which arise when program constructs violate syntax interpretation rules. They are detected and flagged by the syntax interpretation actions, as specified in the definition.

(d) Definition semantic errors, which arise when a definition statement specifies an illegal or impossible operation. They are therefore errors in the language definition, whereas the other three types were errors in the program being parsed.

All errors are reported in the listing, together with some information identifying where they were detected.

2.8 PARSER DIAGRAM

The foregoing reveals fig. 2.1(b) to have been somewhat oversimplified. The complete diagram showing the overall effect of the parser is fig. 2.7.

2.9 THE DEFINITION

The definition language is an adaptation of that described by Williams; it is identical in concept and quite similar in detail. Its basic structure, which closely mirrors that of a generated parser, is described in section 2.9.1; details are given in chapter 3. The

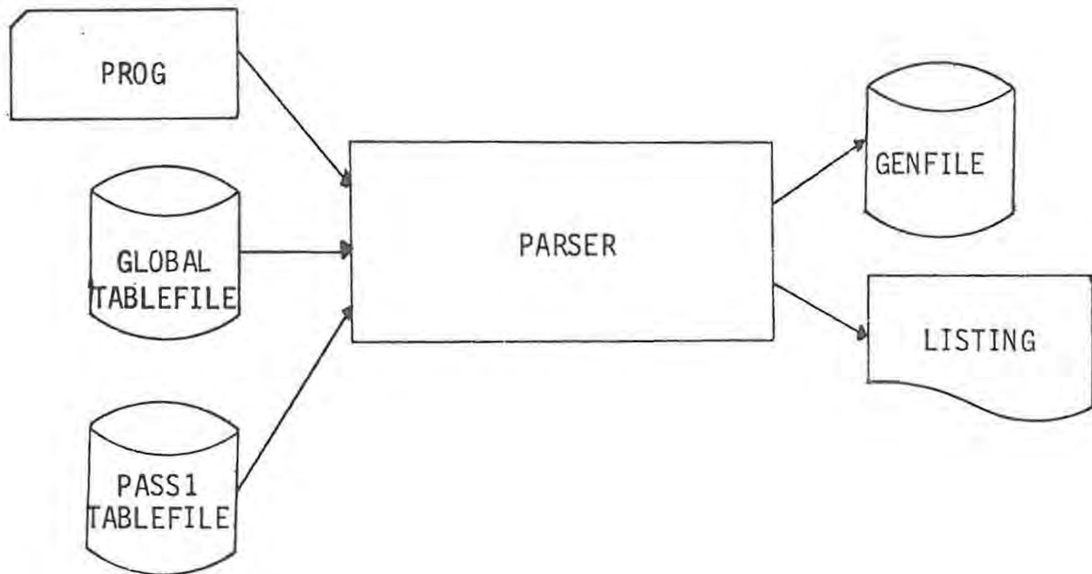

Fig. 2.7: Parser Overview

advantages of the definition language are listed in section 2.9.2.

2.9.1 DEFINITION STRUCTURE

A definition contains the following elements:

(a) Declarations of data types, variables (such as symbol tables) and routines to manipulate them.

(b) Syntax form rules, in BNF augmented by qualifiers.

(c) Syntax interpretation rules specifying manipulation of the symbol tables.

Like the associated parsers, the definition is divided into passes.

Declarations may be global, local to a pass or local to a single rule. In addition to declarations, the first pass consists of syntax form rules and syntax interpretation rules; the other passes just of syntax interpretation rules. The overall structure of the definition is thus as shown in fig. 2.8. The various blocks of declarations are optional, including the terminator and pseudoterminal symbol declarations which are described in section 3.

```
DEFINITION
global declarations;
PASS 1
     terminator symbol declaration;
     pseudoterminal symbol declaration;
     pass 1 declarations;
     syntax form rules;
     pass 1 syntax interpretation rules
     ENDPASS;
PASS 2
     pass 2 declarations;
     pass 2 syntax interpretation rules
     ENDPASS;
 .
 .
 .
ENDDEFN
```

Fig. 2.8: The Overall Structure of a Definition

2.9.2 ADVANTAGES OF THE DEFINITION LANGUAGE

Apart from the fact that it indeed specifies complete syntax in a formal manner, the definition system used has the following advantages:

(a) BNF is familiar and easy to use.

(b) Established techniques exist for generating parsers from BNF grammars.

(c) The syntax interpretation rules specify the manipulation of symbol tables explicitly, precisely in the way the parser works.

(d) The entire structure of the definition and parser are very similar, easing one's understanding of the parser and also greatly facilitating automatic parser generation.

2.10 THE PARSER GENERATOR

As illustrated in fig. 2.1(a), the task of the parser generator is to produce a parser of the type described in sections 2.4 to 2.8 from a definition of the form described in section 2.9. In addition to the parser program itself, it must produce GLOBALTABLEFILE containing string literals used in the definition, PASS1TABLEFILE consisting of the goto table and error recovery tables, information for the writer of the lexical analyser (on LATABLEFILE) and a listing of the definition.

The parser generator is thus a translator, responsible for translating a definition into a Pascal parser and associated tables. The basic structure and effect of the generator are described in this section; it is described in detail in chapter 5.

2.10.1 PARSER GENERATOR STRUCTURE

The parser generator consists of four major components:

(a) The lexical analyser, which recognises symbols in the definition.

(b) The syntax analyser, which receives these symbols, combines them into constructs and builds up internal tables representing these constructs.

(c) The CFSM generator, which, using these internal tables, generates an internal representation of the CFSM. Using this, it checks for conflicts and generates the goto and error recovery tables.

(d) The macro generator, which, using all the internal structures created by the syntax analyser and CFSM generator, generates the parser and the information on LATABLEFILE by expanding skeletons.

It thus has the structure shown in fig. 2.9.

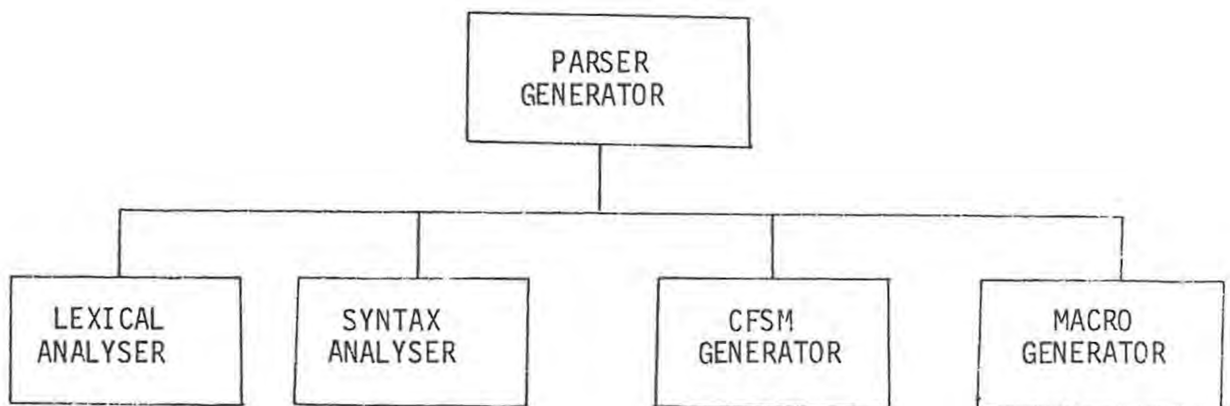

Fig. 2.9: The Structure of the Parser Generator

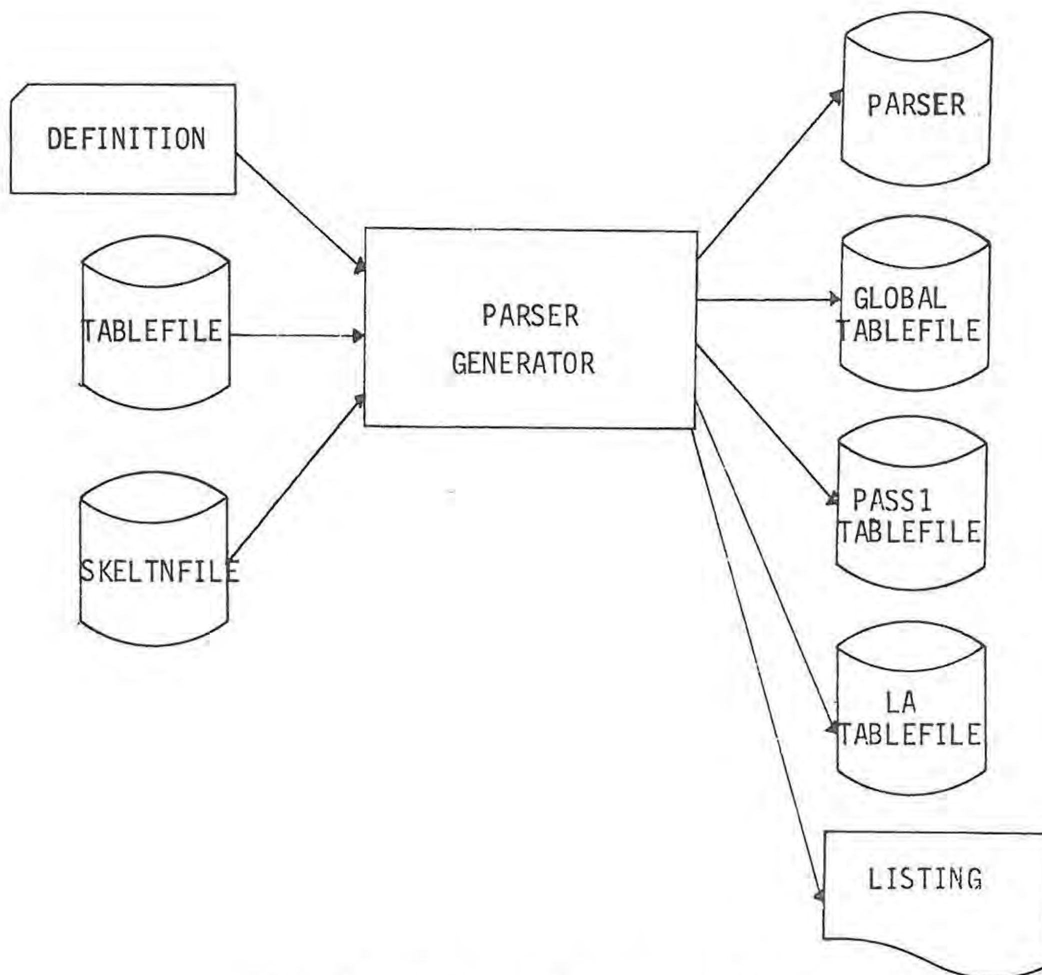

Fig. 2.10: Parser Generator Diagram

3. THE DEFINITION LANGUAGE

The overall structure of a definition was described in section 2.9 and illustrated in fig. 2.8, which is reproduced here as fig. 3.1.

```
DEFINITION
global declarations;
PASS 1
     terminator symbol declaration;
     pseudoterminal symbol declaration;
     pass 1 declarations;
     syntax form rules;
     pass 1 syntax interpretation rules
     ENDPASS;
PASS 2
     pass 2 declarations;
     pass 2 syntax interpretation rules
     ENDPASS;
.
.
.
ENDDEFN
```

Fig. 2.8: The Overall Structure of a Definition

The definition language, as implemented, is described in detail in this chapter. The sequence in which constructs are described is predominantly bottom-up, to minimise forward references, and is as follows:

(3.1) Symbols and Symbol Declarations

(3.2) Declarations

(3.3) Types and Type Declarations

(3.4) Variables and Variable Declarations

(3.5) Procedures and Procedure Declarations

(3.6) Expressions

(3.7) Statements

(3.8) Syntax Form Rules

(3.9) Syntax Interpretation Rules

In addition, a full BNF definition of the definition language is given in appendix 2. Most of the examples used in this chapter have been derived from the first test run, which is given in full in appendix 8.1.

Although the definition language as described and implemented is considered adequate for the definition of programming language syntax, certain extensions to it are desirable. These were borne in mind throughout the development of the parser generator system, but time did not permit their implementation. They are listed in section 3.10.

3.1 SYMBOLS AND SYMBOL DECLARATIONS

Most symbols used in the definition language are described in the appropriate contexts. Some, however, occur frequently in many contexts, and are best described here.

It should be noted at the outset that the character "$" is reserved for special purposes by the parser generator and may never appear in a definition. In particular, the term "any character" in the sequel always excludes "$".

3.1.1 WORD SYMBOLS

Word symbols begin with a capital letter, and consist only of capital letters and digits. They are divided into two classes:

(a) Keywords: special reserved symbols, highlighting or delimiting constructs in a definition, for example:

DEFINITION STACK IF

All keywords are listed in appendix 1.

(b) Identifiers: all names used in a definition for entities such as types and variables, for example:

INTEGER SYMTABSTACK

An identifier may be of any length, provided it fits onto a single line, and all its characters are significant. A keyword may not be used as an identifier. Two implementation restrictions apply:

(1) At most ten different, long identifiers which are identical over their first six characters are allowed in a definition.

(2) Due to various factors, such as indentation, a very long identifier may occasionally not fit onto a single line in the generated parser, in which case an error will be reported.

3.1.2 METALINGUISTIC SYMBOLS

All symbols which can appear after "::=" in a BNF rule are referred to as metalinguistic symbols. They may occasionally be used in other contexts as well, as described in the sequel. There are five classes of metalinguistic symbols:

(a) Metalinguistic Terminator: a single character, used in combination with other symbols to terminate constructs involving metalinguistic symbols. By default, it is the character "@" which denotes it throughout this text. It may be set explicitly to the

character *c* by means of the *terminator symbol declaration*

TERMINATOR c

at the head of the definition of pass 1. *c* may be any character other than space, "!" and "<", and should be chosen to be a distinctive character which never occurs in any other metalinguistic symbol.

(b) *Metalinguistic Or*: the character "!", used to separate alternatives in BNF rules.

(c) *The Empty Symbol*: the symbol <EMPTY>, denoting the null string in a BNF rule.

(d) *Metalinguistic Variables*: the familiar BNF symbols of the form <*l*>, where *l* is a string of letters and spaces; for example:

<ID> <ASSIGN STAT>

The string may be of any length, provided that the entire symbol fits onto a single line. Spaces may be used freely within *l* for readability, but are absolutely insignificant. Metalinguistic variables are generally non-terminal symbols, and are often defined as the non-terminal symbols of a BNF grammar. However, those which stand for single symbols, such as <IDENTIFIER> or <NUMBER>, may more conveniently be recognised and synthesised by the lexical analyser and treated as terminals by the parser. They are referred to as *pseudoterminals*, and must be declared by a *pseudoterminal symbol declaration* of the form

PSEUDOTERMINALS $mv_1, mv_2, \ldots, mv_n$ ENDPT

at the head of the definition of pass 1.

(e) *Metalinguistic Constants*: the remaining symbols in BNF rules, representing terminal symbols, for example:

BEGIN :=

A metalinguistic constant is, in essence, a string literal,

representing the characters of which it consists. It may contain any character other than space, "!", "<" and the metalinguistic terminator, any of which terminates it. It may be of any length, provided it fits onto a single line.

3.1.3 SPACES AND LINE CHANGES

Spaces may appear within metalinguistic variables, in which case they are ignored, or within string literals (section 3.3.3), in which case they are significant. They may not appear within any other symbols, including special compound symbols such as

    ->   <=

The normal use of spaces is to separate symbols. They are required between consecutive symbols that would otherwise appear to be a single symbol, such as metalinguistic constants or word symbols, and may be used freely between symbols, even when not required, to improve readability.

Line changes may also occur between any symbols, but never within a symbol.

3.2 DECLARATIONS

Declarations serve to define and name types, variables and procedures used in a definition; indeed, it is a general rule that no type, variable or procedure may be used in a definition before it has been

declared. Declarations appear in blocks of the form

```
DECLARATIONS
      type declarations;
      variable declarations;
      procedure declarations
      ENDDECL
```

where any, but not all, of the three kinds of declarations may be omitted, together with its preceding semicolon.

Declarations may appear in any or all of the positions indicated in fig. 3.1. Those at the start of a definition are _global declarations_, and their scope is the entire definition; those within a specific pass are _pass declarations_, and their scope is the pass concerned. In addition, variable declarations may appear within various constructs (e.g. qualifiers in syntax form rules), as described in the sequel. These are termed _local declarations_, and their scope is the construct concerned. There are thus three scope levels. Within their scope, low-level declarations override higher-level ones in the usual way.

Types, variables and procedures, and their declarations, are described in detail in the next three sections. Types and variables declared in examples in these sections are used in examples throughout the thesis.

3.3 TYPES AND TYPE DECLARATIONS

A _type_ is, in essence, a set of _values_, or items of information, of a particular kind. The definition language provides three implicit types: INTEGER, BOOLEAN and STRING. In addition, _type declarations_ allow the user to rename types or to declare new types within four classes, viz: _flags_, _records_, _stacks_ and _trees_. Records, stacks and

trees are referred to collectively as structures.

Individual types and type declarations are described below. Among the properties described for each type are the ordering of values of that type, used in comparisons, and the null value, assigned to a variable of that type whenever it is initialised. Finally, the form of a whole block of type declarations is described.

3.3.1 THE IMPLICIT TYPE INTEGER

An integer is a whole number in an implementation-defined range. A non-negative integer is denoted by an integer literal, which is a string of decimal digits, e.g:

    0   12345

Integers are ordered algebraically, and 0 is the null value.

The usual arithmetic operations of addition (+), subtraction or negation (-) and multiplication (*) are defined for integers. Division is handled by two operators DIV and MOD, defined as follows:

    a DIV b = a divided by b, truncated towards 0
    a MOD b = a - (a DIV b) * b

3.3.2 THE IMPLICIT TYPE BOOLEAN

A boolean value is one of the logical truth values denoted by the boolean literals TRUE and FALSE. The values are ordered with FALSE preceding TRUE, and FALSE is the null value.

The usual logical operators, NOT, AND and OR, may be applied to boolean values.

3.3.3 THE IMPLICIT TYPE STRING

A string is a sequence of 0 or more characters. A string s, not containing a quote, is denoted by the string literal

"s"

For example:

"THIS IS A STRING LITERAL" "*"

A quote itself is denoted by the special literal

QUOTE

Strings are ordered lexicographically according to the implementation-defined collating sequence of the underlying character set. Spaces, even trailing spaces, are significant. The null value is the null string consisting of no characters at all, and denoted by

""

The usual operation of concatenation (&) is defined for strings, and two implicit functions, LENGTH and SUBSTRING, are defined on string operands.

LENGTH(s)

returns the number of characters in the string s, an integer.

SUBSTRING (s,n,l)

where $n \geq 0$ and $l > 0$ are integers, returns the substring of the string s beginning at the character numbered n and of length l. Characters of a string are numbered from 0 at the left. No part of the substring specified may extend beyond the end of s. Some examples are given in fig. 3.2.

FUNCTION	RESULT
LENGTH ("XYZ")	3
LENGTH ("")	0
SUBSTRING ("XYZ", 1, 2)	"YZ"
SUBSTRING ("AB", 0, 3)	illegal
SUBSTRING ("PQR", 1, 0)	illegal

Fig. 3.2: Some Examples of LENGTH and SUBSTRING

3.3.4 THE IDENTITY TYPE DECLARATION

For the sake of clarity, it is often desirable to use more than one name for the same type. The identity type declaration

$y_1 = y_2$

specifies that the identifier y_1 names the same type as the previously declared (or implicit) type identifier y_2. For example:

NATURAL = INTEGER

TABLEINDEX = NATURAL

3.3.5 FLAGS

The flag declaration

$y = ('w_1', 'w_2', \ldots, 'w_n')$

where y is an identifier and each w_i a word symbol, declares a flag type, named y, consisting of flags denoted by the flag literals

$'w_1', 'w_2', \ldots, 'w_n'$

For example:

TYPEOFVALUE = ('INT', 'REAL', 'UNDEFINED')

Since flag literals are delimited by apostrophes, the word symbols $\underline{w_i}$ may be keywords, for example:

LOOPSTATEMENT = ('WHILE, 'REPEAT')

The same flag literal may not appear more than once in a single flag declaration, but may appear in more than one flag declaration. For example, assuming the above declaration of LOOPSTATEMENT, the declaration

COMPOUNDSTATEMENT = ('BLOCK, 'IF, 'WHILE', 'REPEAT')

is legitimate.

The order of the literals in the declaration specifies the ordering of the flags they denote, and the null value is the first flag.

Flags are intended as a natural and efficient way of storing single items of non-numeric information, such as tags, symbol types and status indicators. They are very similar to Pascal scalars.

3.3.6 RECORDS

A record is a structure consisting of a variety of named fields of arbitary types. The record declaration

y = RECORD fd_1; fd_2; ...; fd_m ENDREC

declares a record type, named $\underline{y}$, consisting of records with fields described by the field declarations $\underline{fd_1}$, $\underline{fd_2}$, ..., $\underline{fd_m}$. Each $\underline{fd_i}$ has the form

$f_{i1}, f_{i2}, \ldots, f_{in_i} : y_i$

and declares fields named $\underline{f_{i1}}$, $\underline{f_{i2}}$, ..., $\underline{f_{in_i}}$ to be of type $\underline{y_i}$, which

can be any previously declared type, except the record type _y_ itself.

For example, the record declaration

```
SYMTABENTRY = RECORD
                ID: STRING;
                TYPEOFID: TYPEOFVALUE
                ENDREC
```

declares records consisting of two fields, ID and TYPEOFID. The specific record of this type having ID "R" and TYPEOFID 'REAL' is illustrated in fig. 3.3(a).

Fig. 3.3: Records of Type SYMTABENTRY

The field identifiers within a single record must all be distinct, but they may be the same as any other identifiers outside the record. The scope of a field identifier is the record declaration itself. No ordering is defined for records, but the null value of a record type is the record containing the appropriate null values in all its fields. For example, fig. 3.3(b) illustrates the null value of type SYMTABENTRY.

3.3.7 STACKS

The stack declaration

y_1 = STACK OF y_2

declares a stack type named y_1 consisting of stacks of entries of type y_2, which can be any previously declared type except y_1 itself. For example, the stack declaration

TYPESTACK = STACK OF TYPEOFVALUE

declares stacks whose entries are flags of type TYPEOFVALUE. Two specific stacks of this type are illustrated in fig. 3.4. The symbol "◇" stands for the null link.

No ordering is defined for stacks. The null value is the empty stack having no entries.

A predicate EMPTY and a function POP are defined for stacks, as follows:

EMPTY(k) is TRUE if k has no entries, and FALSE otherwise;

POP(k) pops the top element off k and returns it; it is illegal if k is empty.

For example, if k denotes the stack in fig. 3.4(a), then POP(k) returns the flag value 'REAL' and leaves k as illustrated in fig.

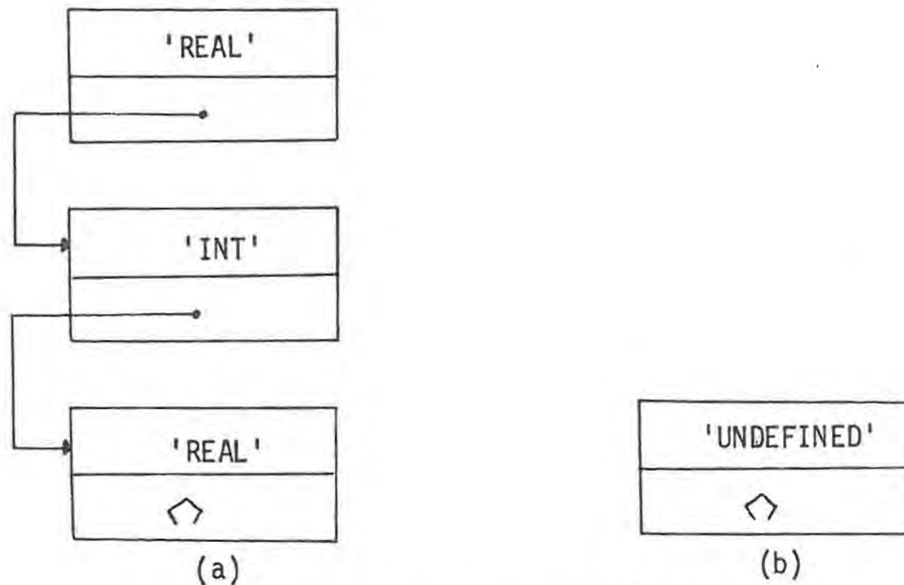

Fig. 3.4: Stacks of Type TYPESTACK

3.5. In addition, statements are provided for pushing, popping, combining, clearing and searching stacks. These are described in section 3.7.

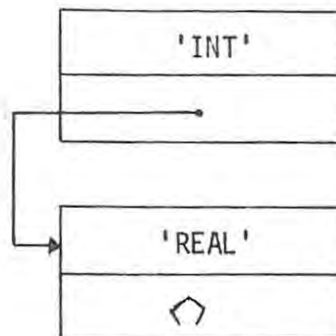

Fig. 3.5: The Stack in fig. 3.4(a) after an Application of POP

3.3.8 TREES

The tree declaration

y_1 = TREE OF y_2

declares a tree type, named y_1, consisting of binary, monkey-puzzle trees of entries of type y_2, which must be a previously defined record type. The first field of a y_2 record must be a string, and is called the search field.

A tree of the type y_1 declared above is defined recursively as follows:

(a) It is either empty, or consists of a y_2 record, the root, and two disjoint subtrees, termed the less and greater subtrees, which are y_1 trees.

(b) If its less (greater) subtree is not empty, then the search field of the root of this subtree precedes (succeeds) the search field of its own root.

For example, the tree declaration

SYMTABTREE = TREE OF SYMTABENTRY

declares trees whose entries are SYMTABENTRY records. A specific example of such a tree is given in fig. 3.6.

No ordering is defined for trees. The null value is the empty tree having no entries.

Though no tree operators or functions are defined, statements are provided for adding entries to trees, clearing trees and searching trees. Searching of trees is very efficient, due to their structure. All statements preserve the properties of trees described above, and are dealt with in section 3.7.

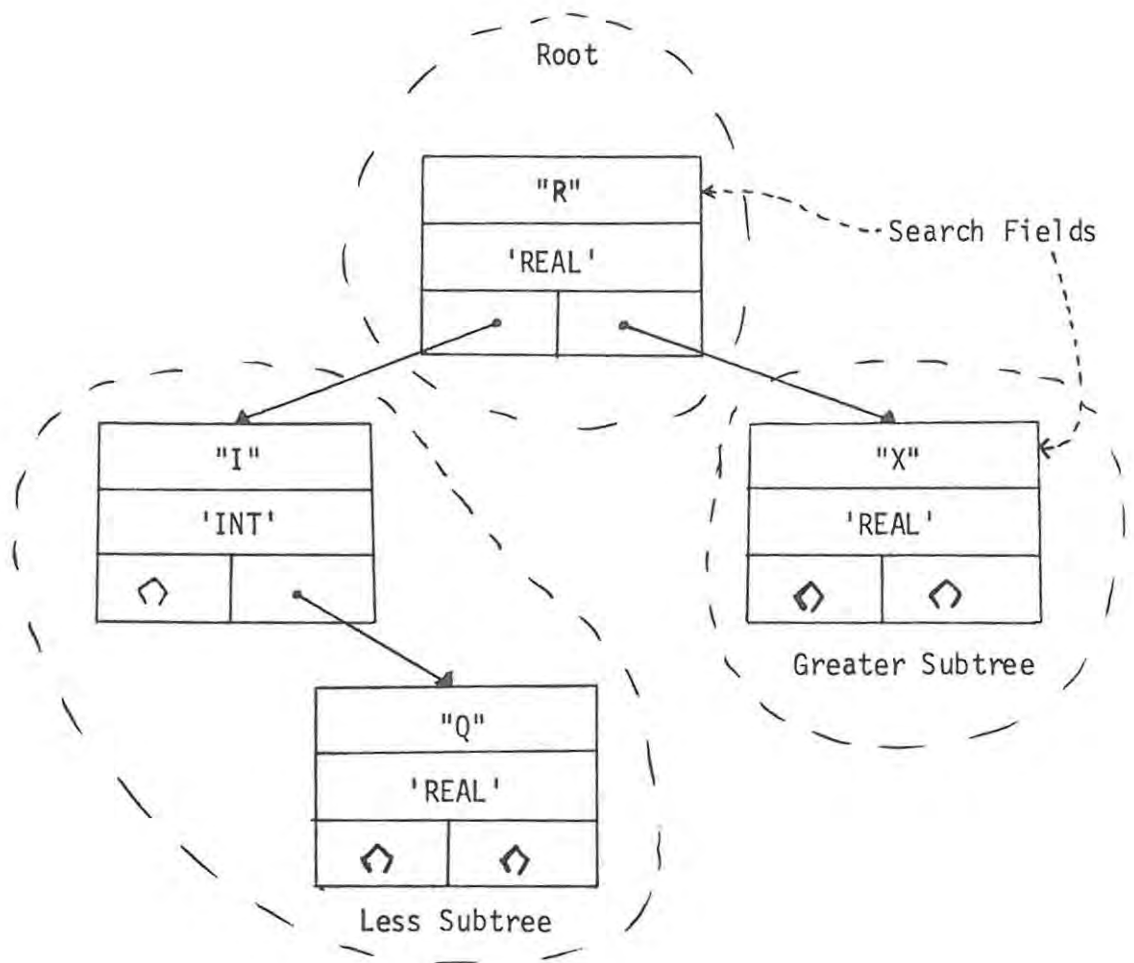

Fig. 3.6: A Tree of Type SYMTABTREE

3.3.9 BLOCKS OF TYPE DECLARATIONS

A block of type declarations within a block of declarations has the form

TYPE d_1; d_2;...; d_n ENDTYPE

where each d_i is one of the type declarations described above. For example:

```
TYPE
    SYMTABSTACK = STACK OF SYMTABTREE;
    ENTRYSTACK  = STACK OF SYMTABENTRY
    ENDTYPE
```

3.4 VARIABLES AND VARIABLE DECLARATIONS

3.4.1 SIMPLE VARIABLES

A simple variable is a named location capable of storing values of a particular type. The variable declaration

$v_1, v_2, \ldots, v_n : y$

declares n variables, named $v_1, v_2, \ldots, v_n$, capable of storing values of the previously declared type y. For example:

```
LHSENTRY, RHSENTRY: SYMTABENTRY
```

A variable declaration thus has the same form as a field declaration in a record.

A block of variable declarations within a block of declarations or a suitable construct has the form

VAR d_1 ; d_2, ...; d_n ENDVAR

where each d_i is a variable declaration. For example:

```
VAR
    EXPTYPESTACK, COMPLETETYPESTACK : TYPESTACK;
    SYMTABFORTHISLEVEL :              SYMTABTREE;
    TYPEOFID :                        TYPEOFVALUE;
    OPERATOR :                        STRING;
    ENTRYSTK :                        ENTRYSTACK;
    I, J :                            INTEGER
    ENDVAR
```

Every variable is initialised automatically at the start of its scope, to the appropriate null value for its type. For example, the variables declared above would be initialised to an empty stack, an empty stack, an empty tree, 'INT', "", an empty stack, 0 and 0 respectively.

3.4.2 FIELDS OF RECORDS

If r is a record variable, then the field f of the record stored in r is also a variable, and is denoted by

r.f

Examples are:

LHSENTRY.ID

and

LHSENTRY.TYPEOFID

which are illustrated for a specific record in fig. 3.7.

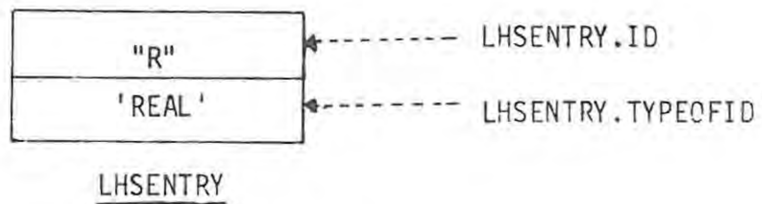

Fig. 3.7: Fields of a Record in LHSENTRY

3.4.3 TOP ENTRIES OF STACKS

If k is a stack variable, then the top entry of the stack stored in k is also a variable, and is denoted by

TOP(k)

An example is

TOP(EXPTYPESTACK)

which is illustrated in fig. 3.8. TOP(k) is undefined if k is empty, and its use in that case will, with the current implementation, lead to unpredictable results and probably disaster.

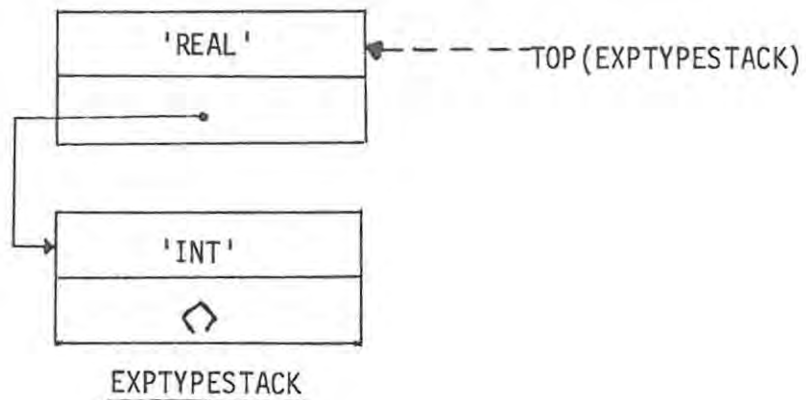

Fig. 3.8: The Top of the Stack in EXPTYPESTACK

3.5 PROCEDURES AND PROCEDURE DECLARATIONS

A procedure is a construct specifying an action, which can be invoked whenever that action must be performed. The procedure declaration

PROCEDURE p; v; sq ENDPROC

declares a procedure, named p, consisting of the statement sequence sq (see section 3.7). v is a block of local variable declarations, which may be omitted together with its semicolon.

A procedure may be invoked, causing its statement sequence to be executed, by referring to its name, as described in section 3.7.4. Since it may have no parameters, it can have an effect only by modifying global or pass variables, or producing some output.

A block of procedure declarations within a block of declarations consists simply of any number of procedure declarations, separated by semicolons. No delimiting symbols are required, as in the case of blocks of type or variable declarations.

3.6 EXPRESSIONS

An expression is a rule or formula for calculating a value, consisting of operands, such as literals and variables, combined by means of operators. Most of the permissible operands and operators have already been dealt with; only metalinguistic variables, lookahead tests and relational operators remain, and are described below. Finally, the order in which operators are applied during expression evaluation is described.

3.6.1 METALINGUISTIC VARIABLES

When a reduction is performed, any metalinguistic variables on the right hand side of the production involved represent specific strings within the program being parsed. The string represented by a pseudoterminal is always a single symbol. The same is true of a non-terminal which appears on the left hand sides of single productions only (i.e. of productions having solitary symbols on their right hand sides), provided the symbols on the right hand sides have the same property. In these cases, the symbol may be referenced in an expression, and is denoted by the metalinguistic variable itself. Any metalinguistic variable used thus in an expression anywhere in a definition is termed a <u>referenced metalinguistic variable</u>.

A metalingusitic variable is a variable only in the sense that, on reductions by different productions, or even on different reductions by the same production, it will represent different strings. No expression or statement in which it occurs, however, may change this string. Consequently, within an expression, a metalinguistic variable is effectively a string constant representing a symbol recognised in the program being parsed. Such access to program symbols is essential to the construction of symbol tables.

A metalinguistic variable may be referenced in any pass, but, in the case of a pseudoterminal, only after the pseudoterminal declaration, and in the case of a non-terminal, only after the syntax form rules. However, as stressed when appropriate in the sequel, it may be referenced only within actions associated with reductions by productions on whose right hand sides it appears. Elsewhere, it has no specific string associated with it, and the effect of referencing

it is therefore unpredictable.

3.6.2 LOOKAHEAD TESTS

Tests on the nature of the lookahead symbol (i.e. the program symbol on which the next shift will occur) are provided for use in expressions. Their chief use is in qualifiers, to resolve LR(0) conflicts by means of lookahead.

The "lookahead symbol is" test of the form

LSIS s_1 ! s_2 ! ... ! s_n @ENDLS

where each s_i is a metalinguistic constant or a pseudoterminal, and "@" is the metalinguistic terminator, as usual, is TRUE if the lookahead symbol is one of the symbols listed, and FALSE otherwise. Its negation

LSISNT s_1 ! s_2 ! ...! s_n ENDLS

is also provided for convenience.

A lookahead test may be used only in pass 1, as only then is lookahead meaningful. Also, any metalinguistic constant listed in a lookahead test must previously have appeared in a production, or it will be considered undefined by the present implementation.

3.6.3 RELATIONAL OPERATORS

The usual relational operators are available, and are denoted as follows:

=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>=	greater than or equal to
>	greater than

Integers, boolean values, flags of the same type and strings may be compared by means of these operators, yielding boolean results in the usual way. Records, stacks and trees may not be compared.

The first operand of a relational operator may not be a flag literal, as its type may be ambigious (the same flag literal may belong to more than one flag type). This restriction does not apply to the second operand, however.

3.6.4 ORDER OF EVALUATION

Operator procedences are as follows:

highest:	* DIV MOD &
	+ -
	= <> < <= >= >
	NOT
	AND
lowest:	OR

When an expression devoid of parentheses is evaluated, operators of high precedence are applied before those of lower precedence, and operators of equal precedence are applied from left to right. Parentheses may be used in the usual way to modify this order of evaluation.

3.7 STATEMENTS

A statement is a construct specifying an action to be performed. The following statements, divided into four categories, are provided:

(a) Simple statements: GEN, ERROR, transfer and procedure statements.

(b) Statements for manipulating structures: PUSH, ADD, POPUP and CLEAR statements.

(c) Conditional statements: IF and SEARCH statements.

(d) Iterative statements: WHILE, REPEAT and FORALL statements.

These are described in detail in this section.

In several contexts, including within conditional and iterative statements, long complex actions may have to be specified. This is done by means of statement sequences, consisting of any number of statements separated by semicolons.

3.7.1 THE GEN STATEMENT

A GEN statement causes one or more values to be output to GENFILE, presumably for later input to a code generator. It has the form

GEN e_1, e_2, ..., e_n

where each e_i is either:

(a) An integer or string expression, in which case its value is output, without any delimiting spaces or other characters, or

(b) NEWLINE, in which case the current line of GENFILE is terminated, and a new one begun.

An example is:

GEN <ID>, ":=", 6, NEWLINE

3.7.2 THE ERROR STATEMENT

The statement

ERROR e

specifies a syntax interpretation error, and causes the parser to print an error message on LISTING. The message consists of a prefix identifying the position reached within the program being parsed, followed by:

SYNTAX INTERPRETATION ERROR: e'

ON REDUCTION BY PRODUCTION p

where e' is the value of the integer or string expression e, and p identifies the production associated with the syntax interpretation action which detected the error.

An example is:

ERROR "UNDEFINED IDENTIFIER '" & <ID> & "' IN EXPRESSION"

The LISTING in appendix 8.1.3 contains several error messages produced by this statement.

3.7.3 THE TRANSFER STATEMENT

The transfer statement

e -> v

causes the value of the expression e to be assigned to the variable v, replacing its previous contents. The types of e and v must be identical.

If e is also a variable, and of a record, stack or tree type, then it is initialised to the appropriate null value, and the assignment is therefore destructive (it is, in fact, a transfer of the contents of the left hand variable to the right hand variable; hence the name of

the statement). In all other cases, the assignment is not destructive, as is common in programming languages.

Fig. 3.9 illustrates the transfer of SYMTABENTRY records. The record with ID the null string "" and TYPEOFID 'INT' is the null value of type SYMTABENTRY.

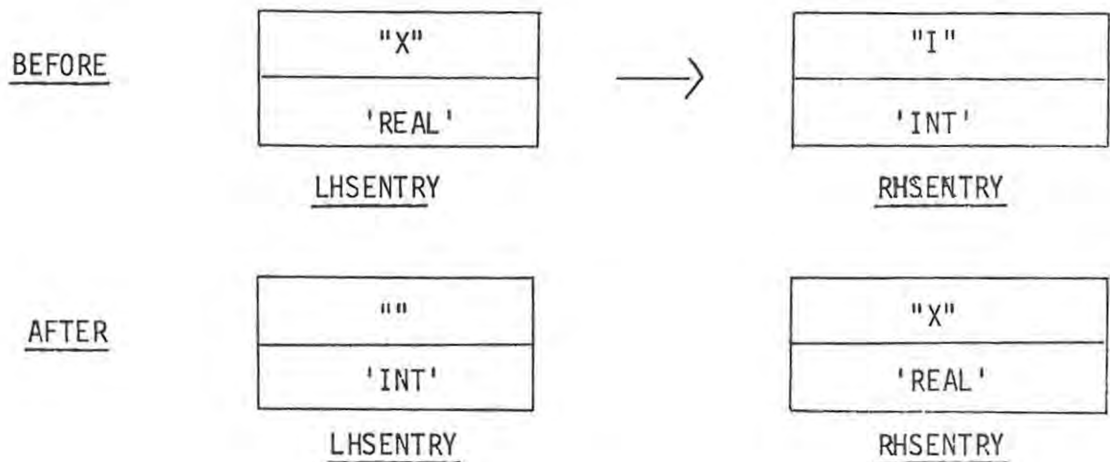

Fig. 3.9: The Effect of LHSENTRY->RHSENTRY

A transfer statement of the form

TOP(v_1) -> v_2

where v_1 is a variable containing a stack of structures, does not pop the stack in v_1, but does initialise its top entry, as illustrated in fig. 3.10.

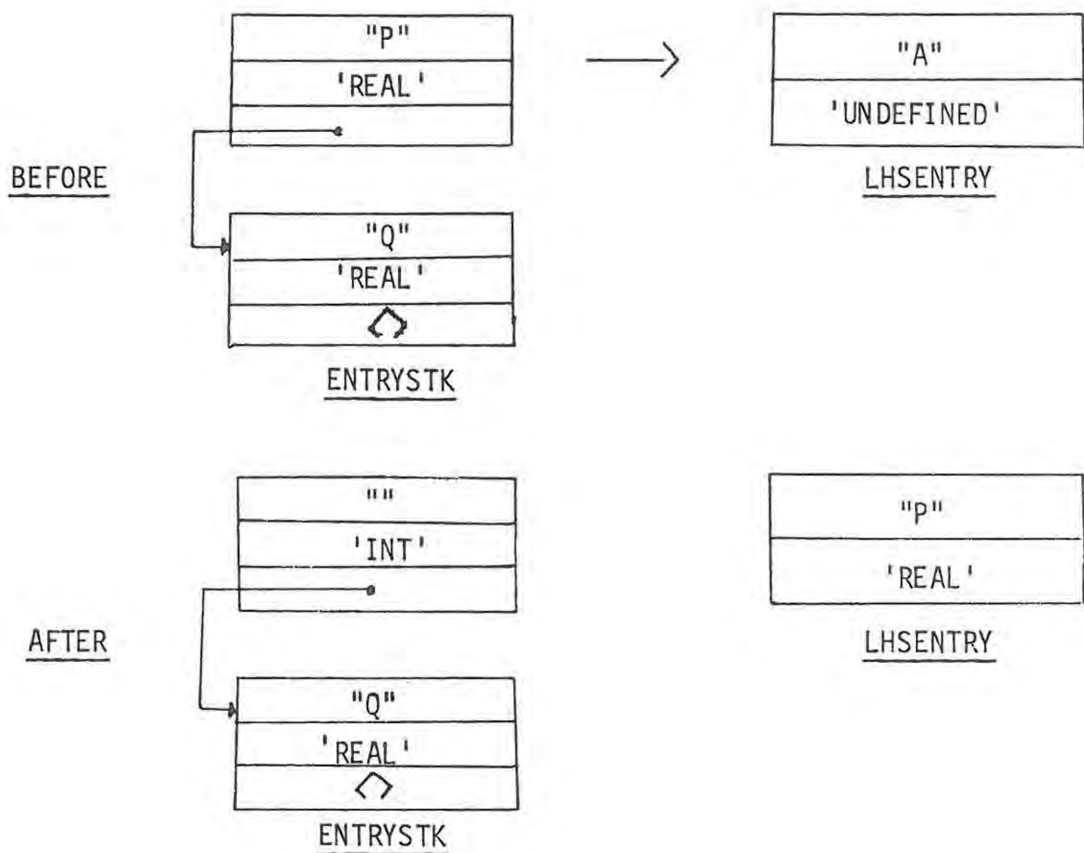

Fig. 3.10: The Effect of TOP(ENTRYSTK) —> LHSENTRY

Though destructive assignment is unusual in programming languages, and hence likely to prove confusing to new users, it is considered justified because it eliminates a great deal of unnecessary copying of complex structures. If copying is indeed required, it can be requested explicitly by means of the function COPY, which takes a structure variable as its only parameter, and produces an identical copy of its contents. The statement

COPY(v_1) -> v_2

leaves v_1 unchanged, since a copy of its contents is made and assigned to v_2. Transfer of values of type integer, boolean, string or flag are never destructive.

3.7.4 THE PROCEDURE STATEMENT

A declared procedure, p, may be invoked by means of the procedure statement

p

which causes its statement sequence to be executed.

3.7.5 THE PUSH STATEMENT

The statement

e PUSH v

where e is an expression of type y and v a variable of type STACK OF y or TREE OF y, causes e to be pushed onto v as a new entry. If v is a stack, e becomes the top entry of v, as illustrated in fig. 3.11. If v is a tree, however, the last operation performed on v must have been an unsuccessful search (section 3.7.10) on the search field of e. In that case, e will be added as a new entry in the appropriate place,

so as to satisfy the properties of trees given in section 3.3.8. Otherwise, the statement is illegal. Pushing onto trees is illustrated in fig. 3.12, which assumes that the appropriate search has been performed.

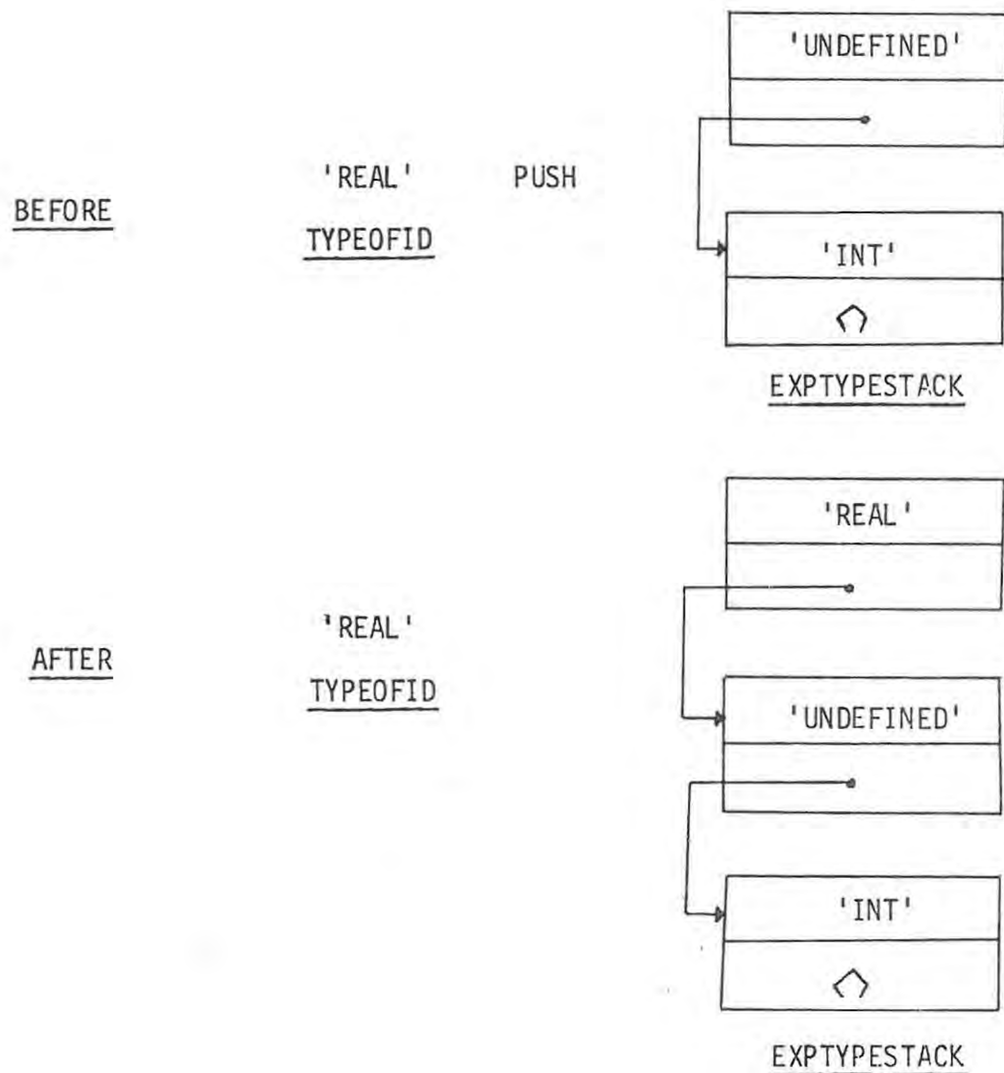

Fig. 3.11: The Effect of TYPEOFID PUSH EXPTYPESTACK

If e itself is a structure variable, it will be initialised, as in the case of a transfer statement. This is illustrated in fig. 3.12. The function COPY may be used to avoid this if desired.

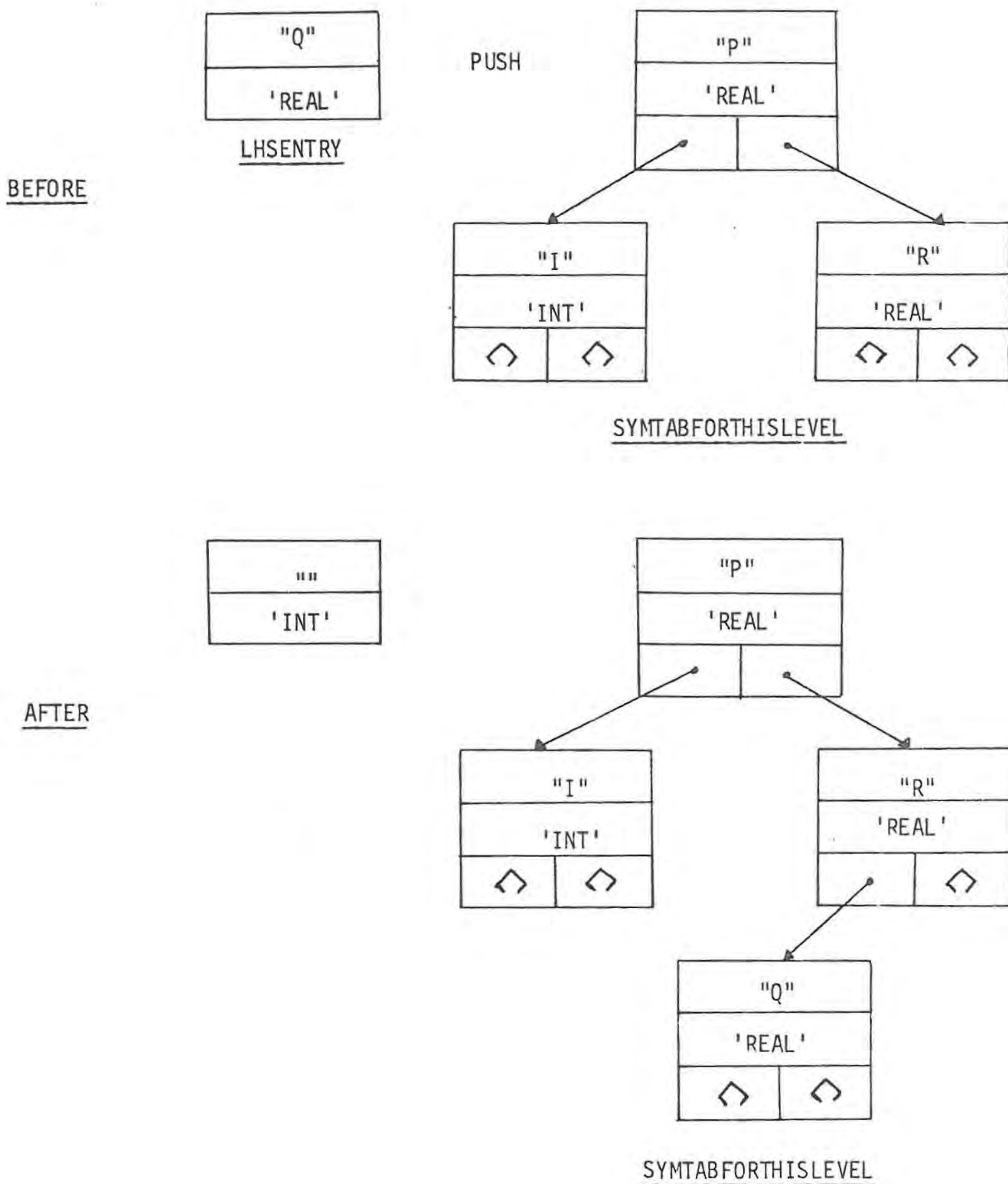

Fig. 3.12: The Effect of LHSENTRY PUSH SYMTABFORTHISLEVEL

3.7.6 THE ADD STATEMENT

The statement

e ADD v

where *e* is an expression and *v* a variable of identical stack types, causes all entries of *e* to be added to the top of *v*, in the same order as they occur in *e*. This is illustrated in fig. 3.13.

As indicated in the figure, if *e* is itself a variable, then it is initialised, as in the case of a transfer or PUSH statement. The function COPY may be used to avoid this, if desired.

3.7.7 THE POPUP STATEMENT

The statement

POPUP v

where *v* is a stack variable, causes the stack in *v* to be popped, and the top entry discarded, as illustrated in fig. 3.14. If *v* is empty, the statement is illegal.

3.7.8 THE CLEAR STATEMENT

The statement

CLEAR v

causes the entire contents of the stack or tree variable *v* to be discarded, and *v* to be initialised.

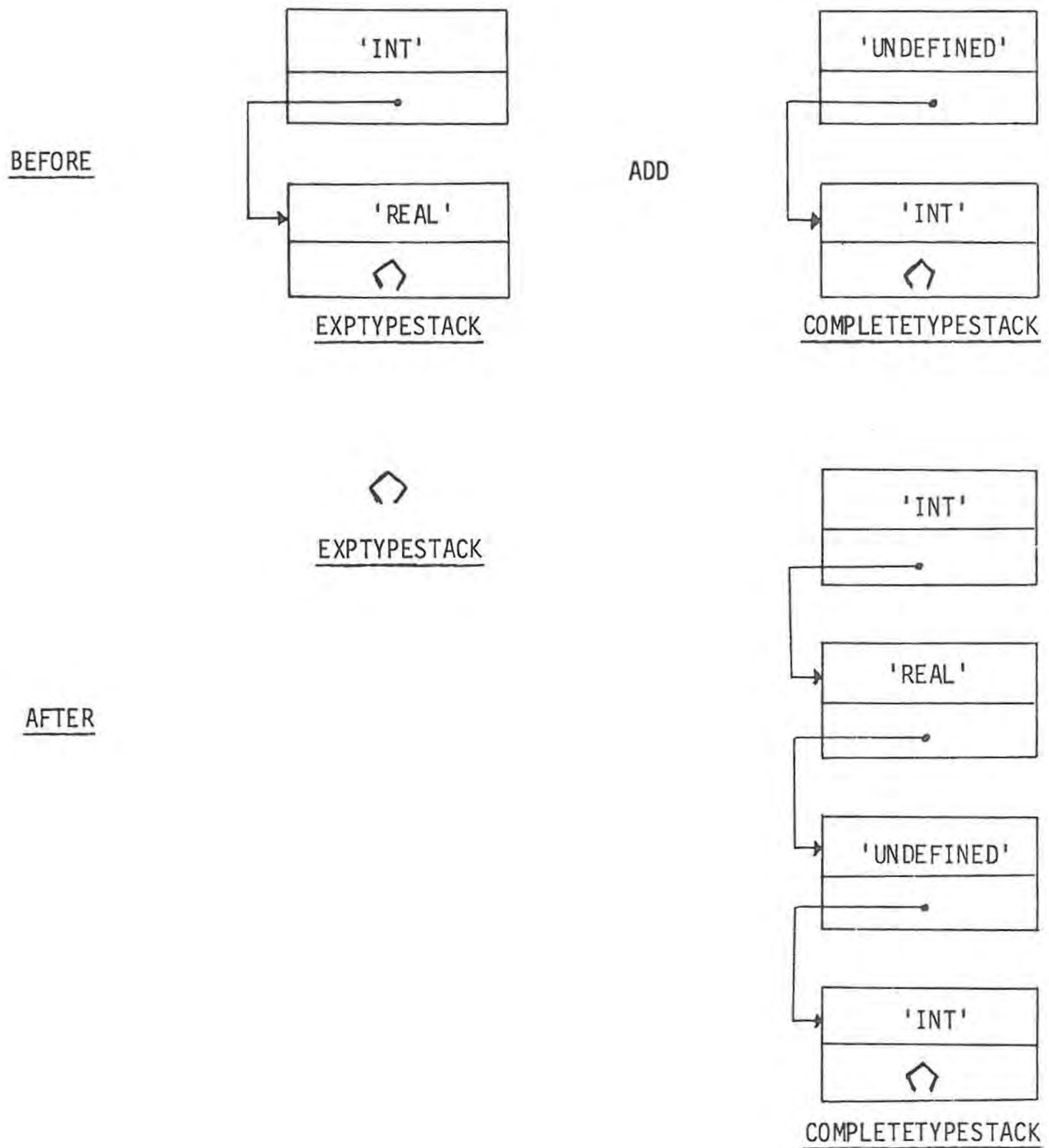

Fig. 3.13: The Effect of EXPTYPESTACK ADD COMPLETETYPESTACK

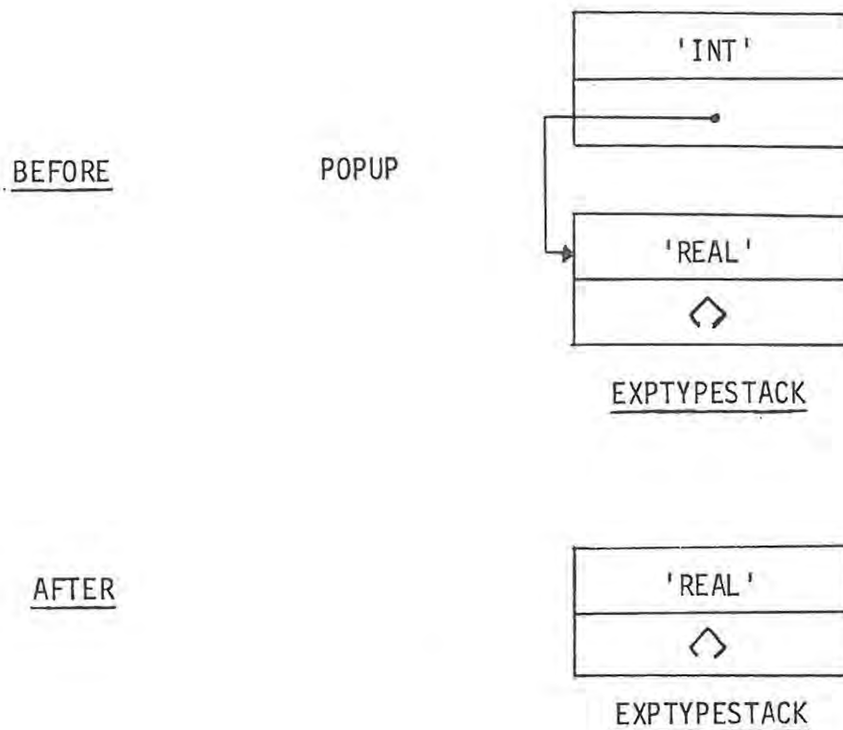

Fig. 3.14: The Effect of POPUP EXPTYPESTACK

3.7.9 THE IF STATEMENT

The statement

IF c THEN sq_1 ELSE sq_2 ENDIF

causes the boolean expression (condition) c to be evaluated, and the statement sequence sq_1 to be executed if it is TRUE, but statement sequence sq_2 if it is FALSE. The statement may have the simpler form

IF c THEN sq ENDIF

in which case no action is taken if c is FALSE.

Examples are:

```
IF OPERATOR = "+" THEN GEN "ADD" ELSE GEN "MULT" ENDIF
IF TYPEOFID <> 'UNDEFINED' THEN
     IF TOP(EXPTYPESTACK) <> TYPEOFID THEN ERROR 5 ENDIF;
     'INT' PUSH EXPTYPESTACK
     ENDIF
```

3.7.10 THE SEARCH STATEMENT

The full SEARCH statement has the form

SEARCH v FOR e FOUND sq_1 FAIL sq_2 ENDS

v is a variable whose type is a searchable structure, which is any of the following:

(a) a tree;

(b) a stack of records whose first fields are strings; or

(c) a stack of searchable structures.

The effect of the statement is to search for a record within v whose search (i.e. first) field is equal to the value of the string expression e. Trees are searched using a binary search; stacks are searched top-down for the first record with the desired search field. If a suitable record is found, the search succeeds, and statement sequence sq_1 is executed. Otherwise the search fails, and statement sequence sq_2 is executed. If the search succeeds, the scope of the found record is opened during the execution of sq_1 so that fields within it may be denoted simply by their names. If the found record is a tree entry, any attempt to change its search field value is illegal; otherwise, any field may be used or changed in any way. The found record as a whole, however, remains inaccessible.

An example of a SEARCH statement is:

```
SEARCH SYMTABFORTHISLEVEL FOR <ID>
    FOUND ERROR "IDENTIFIER '" & ID & "' ALREADY DECLARED"
    FAIL
        <ID> -> LHSENTRY.ID;
        TYPEOFID -> LHSENTRY.TYPEOFID;
        LHSENTRY PUSH SYMTABFORTHISLEVEL
    ENDS
```

The statement may have one of the simpler forms:

```
SEARCH v FOR e FOUND sq ENDS
```

or

```
SEARCH v FOR e FAIL sq ENDS
```

in which case no action is taken if the search is unsuccessful or successful, respectively.

3.7.11 THE WHILE STATEMENT

The statement

```
WHILE c DO sq ENDDO
```

causes the statement sequence *sq* to be executed repeatedly as long as the boolean expression (condition) *c* is TRUE. The test on *c* is performed *before* each iteration, so if *c* is FALSE initially, *sq* is not executed at all.

An example is:

```
WHILE I < J DO GEN I; I + 1 - > I ENDDO
```

3.7.12 THE REPEAT STATEMENT

The statement

```
REPEAT sq UNTIL c
```

causes the statement sequence sq to be executed repeatedly until the boolean expression (condition) c becomes TRUE. The test on c is performed after each iteration, so sq will always be executed at least once.

An example is:

```
REPEAT GEN I, I DIV J, I MOD J, NEWLINE; I + 1 -> I
UNTIL  I = 5 * J
```

3.7.13 THE FORALL STATEMENT

The statement

```
FORALL v DO sq ENDDO
```

where v is a stack variable and sq a statement sequence, is absolutely equivalent to

```
WHILE NOT EMPTY(v) DO sq ENDDO
```

sq should cause v to be popped, as this is not done automatically.

An example is:

```
FORALL ENTRYSTK DO
    GEN TOP(ENTRYSTK).ID, NEWLINE;
    POPUP ENTRYSTK
    ENDDO
```

3.8 SYNTAX FORM RULES

The syntax form rules consist of a BNF grammar and declarations of major terminal symbols, called beacons, to be used by the parser for error recovery. They have the form:

```
SFRULES
        productions;
        strongbeacon declaration;
        weakbeacon declaration
        ENDSF
```

Each beacon declaration may be omitted, together with its preceding semi-colon.

Productions and beacon declarations are now described.

3.8.1 PRODUCTIONS

The productions have the form

```
PRODUCTIONS
        BNF rules
        ENDPROD
```

Successive BNF rules are separated by semi-colons.

3.8.2 BNF RULES

Each BNF rule has the form:

n. nt ::= $r_A u_A$! $r_B u_B$! ... ! $r_X u_X$ @

where

n is an integer literal between 1 and 9999;

nt is a non-terminal metalinguistic variable;

r_1 is either the empty symbol, <EMPTY>, or a sequence of metalinguistic constants and metalinguistic variables;

u_1 is a qualifier (section 3.8.4), and may be omitted

and

x is any letter, say the ith in the alphabet.

The above BNF rule specifies i separate productions, viz:

nA: nt::= $r_A u_A$

nB: nt::= $r_B u_B$

⋮

nx: nt::= $r_X u_X$

nA, nB, ..., nx serve to identify these productions, and hence are termed production identifiers. They are used whenever the productions are referred to. If r_1 is <EMPTY>, u_1 is called an empty production.

An example without qualifiers is:

```
PRODUCTIONS
     1.  <LIST> ::= <ELEMENT> !  <LIST>, <ELEMENT> @;
     2.  <ELEMENT> ::= <EMPTY> !  A @
     ENDPROD
```

which specifies the following four productions:

```
1A: <LIST> ::= <ELEMENT>
1B: <LIST> ::= <LIST>, <ELEMENT>
2A: <ELEMENT> ::= <EMPTY>
2B: <ELEMENT> ::= A
```

Production 2A is an empty production.

3.8.3 RULES GOVERNING BNF RULES

(a) The rule numbers, n above, must be distinct, but need not be consecutive or in order.

(b) Every pseudoterminal declared at the head of pass 1 must appear on the right hand side of at least one production. Every other metalinguistic variable is a non-terminal, and must appear on the left hand side of precisely one BNF rule.

(c) The goal or sentence symbol of the grammar, usually <PROGRAM>, must be on the left hand side of the first BNF rule, and may, though need not, appear in right hand sides as well.

3.8.4 QUALIFIERS

The primary purpose of qualifiers is to resolve LR(0) conflicts. If a reduce action by a particular production never conflicts with any other action, that production requires no qualifier. Suppose, however, that a conflict CFSM state, q, has reduce actions by productions $n_1 l_1$, $n_2 l_2$, ..., $n_i l_i$, $i \geq 1$, and possibly (definitely if $i = 1$) shift actions as well. To resolve these conflicts, qualifiers u_1, u_2, ..., u_i are required for productions $n_1 l_1$, $n_2 l_2$, ..., $n_i l_i$ respectively.

Each qualifier, also termed an iff clause, has the form:

```
@IFF v; c AFTER sq ENDIFF
```

and specifies that a reduce action by the associated production may be performed if and only if the boolean expression (condition) c is TRUE after the statement sequence sq has been executed. v is a block of local variable declarations, which may be omitted together with its semi-colon. The AFTER clause

```
AFTER sq
```

may also be omitted if no action is required before the condition is tested. This occurs in the most frequent case when c is a lookahead test, causing conflicts to be resolved by means of lookahead alone.

The precise effect of qualifiers can be described in terms of the action taken by the parser on reaching the conflict state q above:

(a) It executes the statement sequence and then evaluates the condition of each of the qualifiers u_1, u_2, ..., u_i, in arbitrary order.
(b) If more than one condition is satisfied, the conflict remains and a definition semantic error results.
(c) If one condition is satisfied, a reduce action by the associated production is performed.
(d) If no condition is satisfied, a shift action is attempted. If none is possible, or none is specified for the state q, then a syntax form error results.

3.8.5 RULES GOVERNING QUALIFIERS

(a) Qualifiers are required for all productions involved in LR(0) conflicts, but are permissible (though usually pointless) in other productions as well.
(b) The conditions in the group of qualifiers resolving conflicts at a single state should be mutually exclusive in all circumstances.
(c) The locally declared variables, if present, may be used in c and sq. They will be initialised automatically before sq is executed.
(d) Variables not local to the qualifier may be accessed in c and sq, but should not have their values changed in any way. The reason for this is that sq may be executed and c evaluated frequently and

in a manner difficult to predict during the course of parsing, even if few or no reductions by the associated production are performed. Alterations to global variables in these circumstances are seldom sensible. In this connection, special care should be taken with transfer, PUSH and ADD statements, as these may cause source variables to be initialised.

(e) Metalinguistic variables may not be used in c or sq (except in lookahead tests), as their contents becomes defined only once a specific reduction has been decided on (see section 4.11.4).

3.8.6 DETERMINING WHICH PRODUCTIONS TO QUALIFY

It may be difficult for the user to determine in advance which productions will conflict, and hence will require qualifiers. When generating the LR(0) parser, therefore, the parser generator outputs messages to assist him.

If a conflict exists involving productions for which no qualifiers are specified, a conflict message is output, identifying:

(a) the conflict state;

(b) the conflicting actions; and

(c) the productions involved.

The user need then only qualify each of the productions identified. Examples of these messages appear in appendix 8.5.1.

If qualifiers are specified for some, but not all, the productions involved in a conflict, a qualifier error message is output, listing all the conflicting productions, and indicating that all must have iff clauses. To resolve the conflict entirely, the user need only supply qualifiers for the productions listed which do not already have them.

3.8.7 JUSTIFICATION OF QUALIFIERS

The use of explicit qualifiers for resolving LR(0) conflicts needs justification, as the more usual method is for the parser generator to deduce lookahead conditions automatically from the BNF grammar, and then to generate an SLR(k) or LALR(k) parser, or even to split states to produce a full LR(k) parser, for some $k \geq 1$. The disadvantage of qualifiers over this method is that the user is required to specify explicitly what is already implicit in his grammar.

The chief advantage, on the other hand, is flexibility. The test within a qualifier need not be a lookahead test alone, but may in addition or instead be based on the contents of variables and tables, allowing them to modify parsing. This is particularly useful when defining extensible languages, which allow new keywords, operators and constructs to be defined within a program (for example, Algol68 modes and operators).

The effect is that the LR(0) parser with qualifiers is at least as powerful as an LR(1) parser, without complex lookahead-computing or state-splitting algorithms being required in the generator. This simplification of the generator is the second advantage of qualifiers.

The third advantage has to do with language complexity. A language with conflicts which can be resolved only by lookahead or state-splitting computed in complex ways, is likely to prove confusing for programmers. The present system forces an awareness of such complexity on the language designer, and encourages him to eliminate it, for he will have to code the complex tests himself. The presence of explicit qualifiers in the definition also highlights conflict areas to readers of the definition, and should give them an added

insight into the structure of the language defined.

Ideally, the parser generator should be able to compute simple lookahead automatically, yet allow qualifiers for special and complex cases. This is one of the extensions to the parser generator listed in section 5.9.

3.8.8 BEACONS

Beacons are terminal symbols used by the parser during recovery from syntax form errors, in a manner described below. They may be specified by the user in two categories, strong and weak, by means of the optional declarations:

STRONGBEACONS m_1 m_2 ... m_s @ENDSB

and

WEAKBEACONS m_{s+1} m_{s+2} ... m_w @ENDWB

All the m_i must be distinct, previously used metalinguistic constants or pseudoterminals.

The parser's error recovery procedure, fully described in section 4.11.7, is based on the following assumptions:

(a) Only beacons are significant in error recovery; all other symbols are ignored.

(b) All strong beacons in a program are unconditionally assumed to be in context.

(c) A weak beacon is assumed to be in context if the introduction or removal of non-beacons would make it so. If other beacons have to be introduced or removed in order to establish a valid context for it, however, it is assumed to be out of context and is ignored.

Some corrolaries of this philosophy are:

(a) Any construct introduced by a strong beacon which is used exclusively to introduce such constructs, will always be parsed, whatever errors occurred before it.

(b) If too many symbols are declared as strong beacons, particularly symbols which can occur in many contexts, error recovery may become very artificial, and result in propagation of ridiculous errors.

(c) If too few symbols are declared as beacons, many program symbols will be skipped during error recovery. In the extreme case, if no beacons are declared at all, the entire program will be skipped after the first syntax form error!

Consequently, the following guidelines many prove helpful:

(a) Symbols introducing major constructs (e.g. PROCEDURE, BEGIN and CASE) should be declared strong beacons.

(b) Separators (e.g. "," and ";"), symbols terminating major constructs (e.g. END), operators and brackets should be declared weak beacons.

(c) Ordinary single symbols (e.g. identifiers and literals) should not be declared beacons at all.

These are only guidelines which are by no means proven. Trial and error is at present the only effective way of selecting beacons.

Unfortunately, weak beacons have been only partially implemented. At present, therefore, no WEAKBEACONS declaration may appear in a definition.

3.9 SYNTAX INTERPRETATION RULES

Syntax interpretation rules specify actions to be performed when constructs are recognised, the purpose of these actions being to test the validity of the constructs. They have the form

SIRULES actions ENDSI

Successive actions are separated by semi-colons.

3.9.1 ACTIONS

Each action has the form:

$p_1, p_2, \ldots, p_n$: ACTION v; sq ENDACT

where $\underline{p_1}, \underline{p_2}, \ldots, \underline{p_n}$ are production identifiers. It specifies that, when a reduce action by any of the productions listed is performed, statement sequence sq must be executed. v is a block of local variable declarations, which may be omitted together with its semicolon.

Any metalinguistic variables used in sq, or in any procedure invoked directly or indirectly from sq, must appear on the right hand side of each of the productions, $\underline{p_1}, \underline{p_2}, \ldots, \underline{p_n}$; otherwise, its value will be undefined and its use will have unpredictable effects.

3.10 EXTENSIONS AND IMPROVEMENTS

A variety of extensions to the definition language are possible and desirable, but unfortunately lack of time prevented their implementation. They are:

(a) Procedures with parameters.

(b) User-declared functions.

(c) Procedure and function forward references.

(d) A record construction function, taking as parameters the fields of a record and constructing and returning the corresponding record.

(e) Use of metalinguistic variables in qualifiers before they have appeared in BNF rules.

(f) Elimination of the special pseudoterminal symbol declaration; all metalinguistic variables not appearing on the left hand side of any BNF rule can be taken to be pseudoterminals.

(g) Removal of the restriction that the goal symbol must be on the left hand side of the first production.

(h) $\underline{k}$-symbol lookahead, $\underline{k} > 1$, in qualifiers.

4. THE GENERATED PARSER

As described in sections 2.3 to 2.8 and illustrated in fig. 2.7, the parser generated from a definition is a program which parses programs written in the language defined, producing a listing and generated output for a code generator. The listing contains a copy of the parsed program and all error messages produced during parsing; the generated output contains all output requested by means of GEN statements in the definition. The parser as generated does not contain a lexical analyser: one must be inserted by the user.

A generated parser is intended to be as readable as possible, to enable the user to modify, augment or optimise it easily if he so wishes. For this reason, it is a modular, structured Pascal program. More important, its structure and details mirror those of the definition as closely as possible.

Thus, in most cases, a construct <u>a</u> in the definition has a Pascal counterpart, denoted by <u>a</u>' , in the parser. Each symbol used in the definition appears in almost identical form in the parser. Type, variable and procedure declarations become Pascal type definitions, variable declarations and procedures. Each expression in the definition becomes an almost identical Pascal expression, each statement one or more Pascal statements, each qualifier and syntax interpretation rule a separate Pascal procedure. Only the syntax form rules have no direct counterpart in the parser; they specify the LR(0) core parser, which is implemented as a control routine which invokes a number of subsidiary routines (including the qualifier and syntax interpretation routines just mentioned), and goto and error recovery tables.

The overall structure of a parser also mirrors that of the corresponding definition, consisting of a global portion, and a routine PASS_i_ for each pass, _i_, in the definition. This structure is illustrated in fig. 4.1. The global portion contains everything derived from the global declarations, as well as certain service routines used throughout the parser (such as those to print error messages). PASS1 contains everything derived from the pass 1 definition, viz. declarations, the core parser and syntax interpretation actions. The lexical analyser must also be inserted into PASS1 by the user. Similarly, PASS_i_ contains everything derived from the pass _i_ definition, $i \geqslant 2$, viz. declarations and syntax interpretation rules.

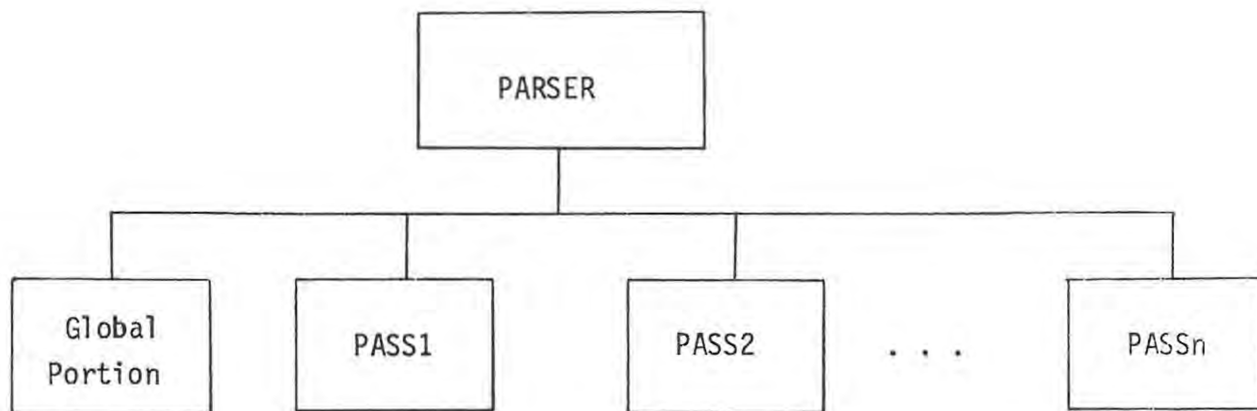

Fig. 4.1: The Overall Structure of a Generated Parser

This chapter gives some, though not all, details of a generated parser. (The skeletons in appendices 6.1 and 7.1 may be consulted for further details). Since the Pascal counterparts of definition constructs appear in many contexts, they are described first, in sections 4.1 to 4.9. The order of description and the notation used are the same as for the description of those constructs in sections 3.1 to 3.9. The structure and operation of the parser are then described in the following sections:

(4.10) Communication between Passes

(4.11) The Core Parser

(4.12) The Lexical Analyser

(4.13) The First Pass

(4.14) Subsequent Passes

Finally, section 4.15 describes desirable improvements to the parser.

Due to lack of space, this chapter contains few examples; the reader is referred to appendices 8.1 and 8.4, which between them contain examples of all constructs.

4.1 SYMBOLS

4.1.1 GENERAL STRATEGY

Many basic symbols in the definition, such as identifiers, flag literals, metalinguistic variables and production identifiers have counterparts in the parser which are as like them as possible. They cannot be identical, however, for two reasons:

(a) Special identifying characters (e.g. ' and <) are not permissible in Pascal symbols.

(b) Whereas all characters of a long symbol are significant in the

definition, only the first eight are guaranteed significant in Pascal.

A two-character prefix is appended to each symbol to solve these problems. It has the form

lx

where _l_ is a letter identifying the type of symbol, such as I for identifier, F for flag literal and M for metalinguistic variable, and _x_ is a digit used to make the first eight characters of every symbol unique. _x_ is usually zero, and only becomes non-zero when a clash occurs. For convenience, whenever the letter _x_, possibly subscripted, appears in a symbol in the sequel, it is assumed to represent this distinguishing digit.

For example, suppose a definition contains the two identifiers LONGSYMBOL1 and LONGSYMBOL2 and the metalinguistic variable <LONG SYMBOL 1>. Their counterparts in the parser are

IOLONGSYMBOL1

I1LONGSYMBOL2

and

MOLONGSYMBOL1

respectively.

In passing, identifiers were fully described above. Metalinguistic variables and production identifiers are now described; other symbols are described in the appropriate context.

4.1.2 METALINGUISTIC VARIABLES

Referenced metalinguistic variables can be accessed in any pass, so they are defined as global variables. As already indicated, the metalinguistic variable

<1>

becomes

Mx1

in the parser.

4.1.3 PRODUCTION IDENTIFIERS

Production identifiers are used frequently in the parser to identify productions or routines associated with productions. The production identifier

n1

in the definition becomes

Pn1

in the parser. Since both _n_ and _1_ are short, no distinguishing digit _x_ is required.

In most cases, the production identifiers are used as prefixes of routine names, such as of qualifier or syntax interpretation routines. Only if a production has a syntax interpretation rule in at least one pass is the production identifier required as a freestanding entity. Such production identifiers are defined to be global scalar values of type PRODUCTIONID.

4.2 DECLARATIONS

Global declarations in the definition become global Pascal declarations in the parser. Similarly, pass *i* declarations become declarations local to PASS*i* and local declarations become declarations local to the appropriate procedure.

The next section describes how the various data types permitted in the definition are represented in the parser, and how they are defined. The two subsequent sections deal with variable and procedure declarations.

4.3 TYPES AND TYPE DECLARATIONS

4.3.1 IMPLICIT TYPES

The implicit types INTEGER and BOOLEAN are represented as Pascal integer and boolean types, but referred to as IOINTEGER and IOBOOLEAN for consistency. Integer and boolean literals require no change.

The standard Pascal representation of strings as packed arrays of characters, however, is inadequate, as such strings are of fixed lengths. Instead, a string is represented as a chain of eight-character chunks, attached to a list head which contains the length of the string and information for garbage collection. A value of type STRING or IOSTRING is actually a pointer to such a list head. (Pointers are used here, as for structures, both to minimise copying, and because they can be returned as the values of Pascal functions). The parser contains global service routines for creating, manipulating, comparing and destroying strings represented thus.

Given this representation of strings, a string literal cannot be denoted by a literal in Pascal, nor can a valid Pascal identifier in general be derived from it. String literals are therefore stored in the global string array STRINGLITERAL, which is initialised from GLOBALTABLEFILE. The literal

"s"

appears in the parser as

STRINGLITERAL[n] (* "s" *)

where _n_ is the appropriate index. If _s_ contains either of the character pairs "(*" or "*)", which would interfere with the comment delimiters, the asterisk is changed to the character "@" in the comment, though not, of course, in the string itself.

Two exceptions are the standard string literals NULLSTRING and QUOTE, representing the strings denoted by "" and QUOTE in the definition, respectively.

4.3.2 THE IDENTITY TYPE DECLARATION

The identity type declaration

$y_1 = y_2$

becomes

$Ix_1y_1 = Ix_2y_2$

The type $\underline{y_1}$ is referred to as _secondary_; all types declared by means of other type declarations are _primary_.

4.3.3 FLAGS

Flags are represented naturally as Pascal scalars, the flag declaration

$y = ('w_1', 'w_2', \ldots, 'w_n')$

generally appearing in the parser as

$Ixy = (Fx_1 w_1, Fx_2 w_2, \ldots, Fx_n w_n)$

However, if a flag literal $'w_i'$ previously appeared in another flag declaration at the same scope level, it is represented here by

$Fz_i x_i w_i$

where z_i is a unique positive integer, since a Pascal scalar may belong to one scalar type only.

4.3.4 RECORDS

A record is represented as a pointer to a Pascal record. The Pascal record, however, contains two extra fields, LINK1 and LINK2, to enable the record to be linked into a stack or tree.

4.3.5 STACKS

A stack of records is represented as a pointer to a stack of the corresponding Pascal records, linked together by means of their LINK1 fields.

A stack of values other than records is represented as a pointer to a stack of special entry records, each containing a value. The declaration

y_1 = STACK OF y_2

where $\underline{y_2}$ is not a record type, becomes

Ex_1y_1 = ↑Jx_1y_1;

Jx_1y_1 = RECORD

LINK1: Ex_1y_1;

VAL : Ix_2y_2

END;

Ix_1y_1 = Ex_1y_1

For example:

ST = STACK OF STRING

becomes

```
EOST = ↑JOST;
JOST = RECORD
       LINK1 : EOST;
       VAL   : IOSTRING
       END;
IOST = EOST
```

$\underline{Jx_1y_1}$ is the entry record, and $\underline{Ex_1y_1}$, a pointer to it, is referred to as an entry. The stack type itself is the same as the entry type, being a pointer to the top record, but the two are separated for clarity.

4.3.6 TREES

A tree is represented as a pointer to a tree head, to which the root is attached. The root and other entries are records as previously described, LINK1 serving as the link to the less subtree, and LINK2 to the greater subtree. In addition to the pointer to the root, the tree head contains details of the most recent search of the tree, used to check the validity of and to implement PUSH statements.

4.3.7 ROUTINES ASSOCIATED WITH USER DECLARED TYPES

Certain operations on structures, such as searching, pushing and copying, are quite complex, and cannot be specified by means of one or two Pascal statements. The parser therefore contains routines to perform them, which are invoked whenever the corresponding operations must be performed.

Unfortunately, because of the rigid type-matching rules governing parameter substitution in Pascal, it is impossible to have a single set of routines to manipulate all structures; a completely separate set is needed for each primary structure type (which may also be used by secondary types derived from the primary types by means of identity type declarations). The set of routines associated with the structure type _y_ consists only of routines for performing operations which will actually be performed on structures of type _y_, chosen from the following list:

AxyADD

DxyDISPOSE

NxyNULL

OxyPOP

SxySEARCH

and UxyPUSH

They are declared at the same scope level as the definition of type _y_.

Having many, repeated, almost identical routines to perform identical manipulations on similar structures is admittedly unattractive, and calls into question the representation chosen for structures. The only way to achieve the ideal situation of having one standard set of routines to cater for all structures, is to combine all record types in the definition into a single Pascal type, and similarly for stacks

and trees. This can be achieved, in the case of records, by defining each definition record to be a variant of a single Pascal record with variants, having the link fields in its fixed part. All stacks and trees of such records would then also be of the same Pascal type.

Such a vast record with variants has the following disadvantages, however:

(a) It is clumsy.

(b) Merging of all structure types detracts from readibility and from the correspondence between the definition and parser.

(c) All fields in the record, even within different variants, must have different names. To ensure this, more artificial prefixes or naming conventions would have to be adopted, further detracting from readibility.

(d) Creating a new record of a specific variant is a clumsy operation, requiring a large case statement. With some implementations, great space wastage may also result.

(e) Search routines would require search fields to be in the fixed part of the record, which would complicate naming them and accessing them as ordinary fields.

For these reasons, the use of a large record with variants was rejected, and the repeated sets of routines tolerated.

4.4 VARIABLES AND VARIABLE DECLARATIONS

4.4.1 SIMPLE VARIABLES

A simple variable in the definition is represented as a Pascal variable of the appropriate type. The variable declaration

$v_1, v_2, \ldots, v_n : y$

therefore appears in the parser as

$$Ix_1v_1,\ Ix_2v_2,\ \ldots,\ Ix_nv_n\ :\ Ixy$$

The automatic initialisation of variables is performed by means of simple assignment statements called initialisation statements. In the case of records and trees, these invoke the appropriate NULL routines (section 4.3.7). The intialisation statements for all global variables appear in the procedure INITGLOBALUSERDECLAREDVARIABLES; for pass i variables in INITPiUSERDECLAREDVARIABLES; and for local variables, at the start of the procedure associated with the construct to which they are local.

4.4.2 FIELDS OF RECORDS

Since a record r is represented as a pointer to a Pascal record, the field

r.f

becomes

r'↑.Ixf

where r' is the Pascal representation of r as usual.

4.4.3 TOP ENTRIES OF STACKS

Since the Pascal equivalent of a stack of records is a pointer to the top entry in the stack,

TOP(k)

becomes simply

k'↑

A stack of other values, however, is represented as a stack of entry

records containing those values in their VAL fields. Hence, for such stacks

TOP(k)

becomes

k'↑.VAL

4.5 PROCEDURES AND PROCEDURE DECLARATIONS

A procedure declaration in the definition appears as a similar one in the parser, either global or within a pass, depending on its scope in the definition.

4.6 EXPRESSIONS

Expressions in the definition become almost identical Pascal expressions in the parser. The few operations that cannot be performed directly in Pascal, such as string concatenation and comparison, are performed by means of function calls. Since the operator precedences in the definition are different from those in Pascal, all sub-expressions are fully parenthesised.

The lookahead test is the only component of an expression worth discussing here. Throughout PASS1, the variable NEXTSYMTOKEN contains the token (see section 4.11.1) of the lookahead symbol. A lookahead test is implemented as a sequence of comparisons between NEXTSYMTOKEN and the tokens of the symbols listed in the test.

4.7 STATEMENTS

On the whole, each simple definition statement becomes a single Pascal statement, and each compound definition statement containing sequences of other statements becomes a compound Pascal statement containing corresponding sequences of Pascal statements. However, the following considerations apply:

(a) Whenever a string or structure is no longer required, disposal statements dispose of it. In the case of a structure, these make use of the appropriate DISPOSE routine (section 4.3.7).

(b) In keeping with the philosophy of destructive assignment of structures, after a structure has been transferred from a variable v by means of a transfer, PUSH or ADD statement, v is initialised by means of initialisation statements like those described in section 4.4.1.

Disposal and initialisation statements are generally single statements. However, in the case of top entries of stacks, they are more complex, involving several statements and temporary variables, described below.

In the light of the above, the Pascal representation of each definition statement is described briefly.

A transfer statement involving integer, boolean or flag values becomes a simple assignment statement; one involving strings is implemented as a call to the procedure ASSIGNSTRING, which takes care of disposal. A transfer statement involving structures is implemented as disposal statements followed by an assignment statement, followed by initialisation statements if the left hand side is a variable.

Procedure calls, WHILE, REPEAT, FORALL and IF statements are directly translated to Pascal procedure calls, while, repeat, while and if statements, respectively. ERROR and GEN statements are implemented as calls to standard procedures SIERROR or SIERRORNUMBER and WRITE, WRITESTRING or WRITELN, respectively. PUSH, ADD, POPUP and CLEAR statements are implemented as invocations of the appropriate PUSH, ADD, POP and DISPOSE procedures (section 4.3.7), in conjunction with disposal and/or initialisation statements where applicable. Temporary variables, described below, may also be involved.

Finally, the SEARCH statement

SEARCH v FOR e FOUND sq_1 FAIL sq_2 ENDS

is implemented as:

```
IF SxySEARCH (v', e', ENTRYi) THEN
          WITH ENTRYi  DO
               BEGIN
               sq1
               END
     ELSE
          BEGIN
          sq2'
          END
```

where y is the primary type of v, and ENTRYi is a temporary variable of the same type as the entry being searched for, i being a number to make the name unique. The purpose of the WITH statement is to open the scope of the fields of the found entry.

As mentioned above, temporary variables are sometimes necessary for the implementation of statements. Each one has a meaningful name containing an integer to make it unique. They are declared in either the global or pass variable declarations, depending on whether the

statements that use them occur within the global area or within a pass.

4.8 SYNTAX FORM RULES

The syntax form rules, as a whole, become the core parser described in section 4.11. Each qualifier, however, becomes a separate routine, described now.

A qualifier associated with production nl in the definition becomes the routine PnlTRYREDUCE in the parser. The routine performs the action in the qualifier (if any), tests the condition, and if necessary causes a reduction to be performed. It also checks that no other qualifier associated with the same state has already been satisfied.

4.9 SYNTAX INTERPRETATION RULES

The syntax interpretation action

p_1, p_2, ..., p_n : ACTION v; sq ENDACT

occuring in pass i becomes the routine

```
PROCEDURE Pp1PASSiSIACTION;
    VAR v';
    BEGIN
    sq'
    END
```

Such routines in passes other than the first may have a few additional statements before sq', described in section 4.14.

4.10 COMMUNICATION BETWEEN PASSES

The program is parsed by the core parser only once, during pass 1, so all reductions are performed during pass 1. However, syntax interpretation rules in subsequent passes specify actions to be taken on specific reductions. A mechanism is therefore required to communicate to these passes what reductions were performed during pass 1. Furthermore, as the syntax interpretation actions associated with a production may refer to metalinguistic variables on the right hand side of that production, this mechanism must also furnish the values of these metalinguistic variables at the time the reduction is performed.

The sequence of productions by which reductions are performed by an LR parser constitutes a right canonical parse. A subsequence of this, called a partial parse, is output during pass 1 for each subsequent pass. The partial parse for pass *i* is sent to the file PASS*i*PARTIALPARSE, and contains details of only those productions for which pass *i* syntax interpretation actions are specified. These details include the production identifiers and the values of all referenced metalinguistic variables occurring on the right hand sides of the productions.

4.11 THE CORE PARSER

As mentioned in section 2.5, the core parser is an LR(0) parser incorporating qualifers to resolve conflicts. It parses a stream of tokens, representing program symbols, recognised by the lexical analyser and presented to the parser on request. The parser consists of a CFSM, represented by means of a parsing action table and a

goto table, and a state stack. Details of the various aspects of the core parser are now described.

The lexical analyser is dealt with in section 4.12. It is worth noting now, however, that when it recognises a symbol, at the request of the core parser, it sets up the four variables NEXTSYMTOKEN, NEXTSYMSTRING, NEXTSYMLINENO and NEXTSYMPOSITION describing that symbol.

4.11.1 TOKENS

Tokens are values identifying terminal symbols recognised by the lexical analyser. They are defined as Pascal scalars of the type TOKEN, local to PASS1. The pseudoterminal

<1>

has associated token

Tx1

Valid Pascal symbols cannot, in general, be constructed from metalinguistic constants, however, so an approach similar to that used in the case of string literals is used: the token associated with the metalinguistic constant

c

is

TOKN*i* (* "c" *)

where *i* is a unique integer.

The tokens are divided into five groups:

(a) Strong beacons, corresponding to the strong beacons declared in the definition, and including an additional token, ENDTOKEN, representing the end of the program.

(b) Weak beacons, corresponding to the weak beacons declared in the definition.

(c) Ordinary metalinguistic constants.

(d) Ordinary metalinguistic variables (i.e. pseudoterminals which are not beacons).

(e) Empty and error tokens, EMPTYTOKEN and ERRORTOKEN, used to represent the symbol <EMPTY>, and in error recovery, respectively.

4.11.2 STATES AND STATE SETS

A state of the CFSM is represented by a positive integer. The initial state is state 1.

Sets of states are also used for various purposes within the core parser. Unfortunately, Pascal restricts the number of elements allowed in a Pascal set to an implementation dependent and usually quite small number. Consequently, the type STATESET is defined as an array of Pascal sets, just as long as required to accommodate all states in the CFSM.

4.11.3 THE STATE STACK

The state stack is implemented as the variable CURRSTATE and the array STATESTACK, the top of which is indicated by the variable TOPOFSTATESTACK. CURRSTATE holds the current LR(0) state. STATESTACK holds information about each state entered by the CFSM in reaching the current state, including the state itself and the token and text of the terminal symbol under which a shift from the state occurred. The token is used for error recovery and the text to set up the values of

referenced metalinguistic variables.

4.11.4 ROUTINES PERFORMING LR ACTIONS

The LR shift, reduce, accept and error actions, described in section 2.5.1, involve fairly straightforward manipulation of the state stack. They are performed by the routines SHIFT, REDUCE or REDUCEEMPTY, ACCEPT and SFERROR, respectively.

If a syntax interpretation action is associated with production n1 in any pass, however, a reduction by that production involves more than a mere LR reduce action, and is performed by the procedure Pn1REDUCE. This procedure performs whichever of the following are necessary:

(a) Sets up the values of referenced metalinguistic variables. If the right hand side of the production contains the referenced metalinguistic variable <1>, m symbols from the end, the mth entry from the top of the state stack will contain its text. This string is transferred to the variable Mx1 for use in syntax interpretation actions. This explains why metalinguistic variables are accessible only when a reduction is being performed.

(b) Invokes Pn1PASS1SIACTION to perform the pass 1 syntax interpretation action, if any.

(c) Outputs partial parse entries for subsequent passes.

(d) Invokes REDUCE or REDUCEEMPTY to perform the actual LR reduce action.

4.11.5 THE PARSING ACTION TABLE

The parsing action table is implemented as the procedure PARSINGACTION, which consists of a case statement indexed on states. Each case option consists of statements to select and perform the appropriate action. The selection of the appropriate action is performed by invoking TRYREDUCE routines, in the case of conflicting reduce actions, or testing NEXTSYMTOKEN, in the case of shift actions. The TRYREDUCE routines corresponding to all qualifiers associated with a state are always invoked, to ensure that only one condition is satisfied. The performance of an action itself is specified as a call to one of the routines described in the previous section. The case option for each state is thus short, readable and easy to modify.

A special situation arises if LR conflicts remain unresolved in the parser. If state *q* is a conflict state, unresolved by means of qualifiers, the parser contains a special routine Q*q*CONFLICT, which simply generates a definition semantic error. The sole action in PARSINGACTION for state *q* is to call this routine.

If no conflict state is entered by the parser, no error will result. Nonetheless, such a parser is of little use, and the intention of the conflict routines is that they be modified by the user. This gives him an even more powerful tool than qualifiers for resolving LR conflicts, as he may perform any manipulations on any parser data structures he pleases in such a modified conflict routine. Use of this method is certainly not recommended for conflict resolution; qualifiers should prove more than adequate. It is available, however, for use in exceptional circumstances.

4.11.6 THE GOTO TABLE

The goto table is implemented as the single array GOTOTABLE. In fact, it is partitioned into sub-tables, one for each non-terminal symbol. The start index of the sub-table for the non-terminal

<1>

is specified by the pass 1 constant

Gx1GOTOTABSTARTINDEX

Each goto table entry contains a state set, STATSET, and a state, NEXTSTATE. After a reduction by a production with <1> on its left hand side, the procedure GOTOSTATE is invoked to determine the next state. It consults the sub-table starting at Gx1GOTOTABSTARTINDEX, and finds the first entry whose STATSET contains the state at the top of the state stack; such an entry always exists. It then sets CURRSTATE to the NEXTSTATE specified in that entry.

4.11.7 ERROR REPORTING AND RECOVERY

A syntax form error is detected by an LR parser when the current state has no possible action for the current input symbol. The syntax form error message sent to LISTING specifies the position in the program at which the error is detected, and the current state. The user should consult the parsing action table entry for the state in PARSINGACTION, to determine what symbols were expected. This error reporting is clearly unsatisfactory; improvement of it is listed as a desirable extension to the parser in section 4.15.

When a syntax form error occurs, the state stack contains states representing the path followed through the CFSM from the initial state to the current state, in which no action is possible on the current

input symbol. Error recovery involves finding a symbol in the program and a new state stack configuration so that the new current state will have an action on the new current input symbol, to enable the parsing to resume. Error recovery will be good only if, for simple, common errors, parsing continues sensibly thereafter. The chance of this occurring should be maximised if as few symbols as possible of the program are skipped, and if the new state stack configuration is as close as possible to the old.

The basic error recovery philosophy described in section 3.8.8 seeks to achieve this. It is implemented as follows:

(a) The state stack is restored to the form it had immediately after the last successful shift action. Since LR(0) reductions are performed without consulting lookahead, a whole string of them may well be performed immediately before a syntax form error is detected, and all or some of these may be invalid.

(b) All program symbols are skipped until the current input symbol is a beacon, $\underline{b}$.

(c) The state stack is scanned from the top down for a state $\underline{q_0}$ with the property that a path $\underline{q_0 q_1 q_2} \cdots \underline{q_n}$ through the CFSM exists such that:

   (1) $\underline{q_n}$ has a shift action on $\underline{b}$; and

   (2) none of the transitions $\underline{q_i} \rightarrow \underline{q_{i+1}}$ is under a beacon.

   This is done using the table STATESWITHDIRECTPATHSTOBEACONS, which consists of a state set for each beacon, $\underline{b}$, containing all states $\underline{q_0}$ satisfying the above conditions.

(d) If such a state is found, the state stack is popped up to $\underline{q_0}$, then $\underline{q_1}$, $\underline{q_2}$, ..., $\underline{q_n}$ are pushed onto it, and error recovery is complete. The states $\underline{q_1}$, $\underline{q_2}$, ..., $\underline{q_n}$ are determined using the table TABLEOFPATHSTOBEACONS, which consists of a separate

sub-table for each beacon. Each entry comprises a state set and a next state. The sub-table for $\underline{b}$ has the property that each state, $\underline{q_i}$, on the path $\underline{q_0 q_1} \ldots \underline{q_{n-1}}$ is in the state set of precisely one of its entries, and the corresponding next state is $\underline{q_{i+1}}$.

(e) If no such state is found, and $\underline{b}$ is a weak beacon, program symbols are skipped until the next beacon is reached, and the process is repeated from step (c).

(f) If no such state is found, and $\underline{b}$ is a strong beacon, then the state stack is scanned from the top down until a state $\underline{q_0}$ is found with the property that a path $\underline{q_0 q_1} \ldots \underline{q_n}$ through the CFSM exists such that $\underline{q_n}$ has a shift action under $\underline{b}$. This is done using the table STATSWITHINDIRECTPATHSTOBEACONS, much as described in step (c). (In this case, there is no restriction on the transitions $\underline{q_i} \rightarrow \underline{q_{i+1}}$; consequently, $\underline{q_0}$ can always be found, for the state at the bottom of the state stack is always the initial state, from which any state in the CFSM can be reached). The state stack is then adjusted as in step (d), and error recovery is complete.

When entries for the path $\underline{q_1 q_2} \ldots \underline{q_n}$ are pushed onto the state stack, the token in each entry is set to ERRORTOKEN. Any syntax interpretation action associated with a reducton involving any such entry is not performed, as it is unreasonable to check the validity of a construct not in the program, but assumed by the parser for the purpose of error recovery.

4.12 THE LEXICAL ANALYSER

The function of the lexical analyser is to recognise program symbols at the request of the core parser, and to pass details of them to the parser. It is not generated with the parser, but must be handwritten in Pascal (or generated by a separate lexical analyser generator). This section serves as a guide to the user in writing a lexical analyser.

The parser generator produces a list of the BNF symbols to be recognised - the terminals and pseudoterminals - and the tokens used to represent them in the parser, on the file LATABLEFILE. This list, the information given in this section, and a knowledge of the meaning of the pseudoterminals are sufficient for writing the lexical analyser; it should not be necessary to examine the generated parser in which it will be embedded.

4.12.1 REQUIREMENTS

Two lexical analyser routines are required:

(a) INITLEXICALANALYSER, to initialise all variables associated with the lexical analyser at the start of parsing; and

(b) GETNEXTSYMBOL, to recognise a program symbol and set up the four variables describing it (section 4.11).

Details of these routines are given in sections 4.12.3 and 4.12.4.

These routines may call any other routines the user chooses to write. The whole body of routines must be embedded in the PASS1 procedure declarations in place of the comment

```
(*** LA PROCEDURES ***)
```

The lexical analyser routines will invariably use non-local variables. The user is free to declare whatever variables he requires, though he is advised to give them names beginning with "LA" to avoid inadvertant clashes with other identifiers used in the parser. These variable declarations must replace the comment

(*** LA VARIABLES ***)

in the PASS1 variable declarations.

4.12.2 SERVICE ROUTINES

The following service routines are provided in the parser, specifically for use by the lexical analyser:

(a) READLINEOFPROGRAM, which reads a program line and lists it, together with any error messages for the previous line.

(b) SETNEXTSYMBOL, which sets up the four variables NEXTSYMTOKEN, NEXTSYMSTRING, NEXTSYMLINENO and NEXTSYMPOSITION from the arguments provided.

(c) CREATESTRING, which constructs a STRING (section 4.3.1) from part of a program line.

(d) ILLEGALSYMBOL, which generates an error message indicating that an illegal symbol has been detected.

These routines are simple and readable, so further details can be obtained by examining them. They appear in all the generated parsers in appendix 8.

The user is urged to use these routines rather than to access any parser files or data structures directly, as this might upset related processes (such as listing and error reporting) performed by the parser.

4.12.3 INITLEXICALANALYSER

This procedure takes no parameters. It is invoked once at the beginning of PASS1 to initialise all lexical analyser variables. It will probably involve reading the first program line. It must not invoke GETNEXTSYMBOL to recognise the first program symbol, as this is done explicitly by the parser.

4.12.4 GETNEXTSYMBOL

This procedure also takes no parameters. It is invoked by the core parser whenever the next program symbol is required, such as during a shift action. It must recognise the symbol, and set up details of it by means of SETNEXTSYMBOL. If an illegal symbol is detected, ILLEGALSYMBOL should be called, and then the next symbol sought and recognised: a legal symbol must always be found for use by the core parser.

The core parser will always go on requesting symbols until the end of the program being parsed is reached. When this occurs, the special endsymbol having token ENDTOKEN, string NULLSTRING and position the end of the last program line, must be set up. It is a valid symbol, expected by the core parser, and is the last symbol that is requested. The end of the program is indicated by the condition

```
EOF(PROG)
```

when a new program line is about to be read. It must be tested explicitly by the lexical analyser, or a Pascal error will result.

The string supplied to SETNEXTSYMBOL for setting up NEXTSYMSTRING is, in fact, required only in the following cases:

(a) Referenced pseudoterminals;

(b) Symbols which may be directly or indirectly reduced to referenced non-terminals.

In any other case, the lexical analyser need not construct the string, and can specify it as NULLSTRING. It must not, however, be left undefined or set to nil.

4.13 THE FIRST PASS

The heart of pass 1 is the core parser, which controls all the action and invokes all the other routines, as already described. The procedure PASS1 itself simply performs necessary initialisation, and then invokes PARSINGACTION repeatedly until an accept action is performed. By that time, all parsing will have been performed, including pass 1 syntax interpretation actions, and all partial parses will have been produced.

4.14 SUBSEQUENT PASSES

Just as the heart of pass 1, controlling all the action, is the core parser, so the heart of pass *i*, $i \geq 2$, is PASS*i*PARTIALPARSE produced by the core parser. The routine PASS*i*SIACTION is the analogy of PARSINGACTION. It reads a partial parse entry, except the values of the referenced metalinguistic variables, determines what action must be performed by means of a case statement, and invokes the appropriate syntax interpretation action routine to perform it. This routine itself obtains the values of any referenced metalinguistic variables from the partial parse, and then performs the action.

Like PASS1, PASSi consists of necessary initialisation, followed by repeated execution of PASSiSIACTION until all partial parse entries have been dealt with. By then, all pass i syntax interpretation actions have been performed.

4.15 EXTENSIONS AND IMPROVEMENTS

The following are the chief improvements that should be made to the parser:

(a) More meaningful syntax form error messages should be produced. At the very least, the parser generator should produce a table explaining the current error messages, so the user would not have to consult the parser program.

(b) Variables should be disposed of at the end of their scopes. At present they are not, with the result that many unused and inaccessible structures occupy valuable space.

5. THE PARSER GENERATOR

The function of the parser generator is to accept a complete syntax definition of the form described in chapter 3, and from it to generate a parser of the form described in chapter 4, together with its GLOBALTABLEFILE and PASS1TABLEFILE, LATABLEFILE for the lexical analyser writer and a listing for the user. The listing includes messages indicating errors in the definition. The correspondence between the definition and parser is so great that much of the generation involves straightforward translation of the definition language into Pascal.

As described in section 2.10.1 and illustrated in fig. 2.9, the parser generator is also a Pascal program, and consists of the following major components:

(a) Lexical Analyser;

(b) Syntax Analyser;

(c) CFSM Generator; and

(d) Macro Generator.

The overall strategy is as follows. The lexical analyser recognises symbols in the definition, and passes them to the syntax analyser. The syntax analyser checks them for validity, combines them into constructs and builds up a variety of internal structures, such as symbol tables, expressions and lists of productions. The CFSM generator uses the structures derived from the syntax form rules to generate an internal representation of the LR(0) CFSM. Using this, it checks for conflicts, and generates the goto table and error recovery tables. The macro generator uses all the internal structures to generate the parser and LATABLEFILE by expanding skeletons.

Section 5.1 describes the internal structures mentioned above, and sections 5.2 to 5.5 the four components of the parser generator. The generation of GLOBALTABLEFILE is dealt with in section 5.6, and the parser generator's own table files, TABLEFILE and SKELTNFILE, are described in sections 5.7 and 5.8. Finally, secion 5.9 lists some desirable improvements and extensions to the parser generator.

5.1 INTERNAL STRUCTURES

5.1.1 SYMBOL TABLES

Five symbol tables are used to store details of all symbols encountered in the definition, viz:

(a) IDTABLE for identifiers;

(b) FLAGLITTABLE for flag literals;

(c) STRGLITTABLE for string literals;

(d) METAVARTABLE for metalinguistic variables; and

(e) METACONTABLE for metalinguistic constants.

The last three are ordinary SYMBOLTABLEs; the first two, however, are STACKEDSYMBOLTABLEs, each consisting of a separate SYMBOLTABLE for each scope.

A SYMBOLTABLE is basically a monkey-puzzle tree (similar to a tree in the definition language), modified to facilitate the generation of long symbols of the form described in section 4.1.1.

5.1.2 EXPRESSIONS

Expressions recognised by the syntax analyser are set up as trees in the usual way. Leaves contain values of literals or pointers to symbol table entries of operands; other nodes contain operators.

5.1.3 PRODUCTION TABLES

A variety of arrays are used to store information about each production, such as the metalinguistic symbols it comprises, its production identifier, whether or not it is qualified, in which passes it has syntax interpretation actions associated with it, and, if it appears in the list of productions associated with a syntax interpretation action, the first production in that list. Throughout the parser generator, productions are referred to by production numbers, which serve to index some of these arrays.

5.1.4 THE CFSM

The CFSM generated by the CFSM generator is a linked structure contained within three arrays:

(a) STATE containing details of each state;

(b) NQTAB specifying all state transitions under terminals (shift actions) and non-terminals (goto actions); and

(c) REDTAB specifying all reduce actions.

All shift, reduce and goto actions applying to a state are specified in STATE by references to NQTAB and REDTAB.

The internal CFSM is therefore just a direct representation of a transition graph, such as that shown in fig. 2.3, as a linked structure, with transitions under terminal, non-terminal and # symbols stored as the shift, goto and reduce actions they represent.

5.1.5 THE SYNTAX INTERPRETATION ACTION TABLE

When the syntax analyser analyses a syntax interpretation action, it builds up a syntax interpretation action table entry consisting of the production numbers of the productions associated with that action. This information is required for construction of the case statement in PASSiSIACTION, $i \geq 2$, (section 4.14). Accordingly, the table is called PASISIACTIONTABLE, though it is also set up for pass 1 actions.

5.1.6 MISCELLANEOUS VARIABLES

The structures already described are the main internal structures used for communication between the four components of the parser generator. In addition, however, there are a variety of isolated variables used to contain miscellaneous items of information such as the text of the current symbol, whether there are unresolved LR(0) conflicts in the CFSM, details of strong and weak beacons declared, the number of string literals in the definition, and so on.

5.2 THE LEXICAL ANALYSER

The lexical analyser is straightforward, and its internal workings are not worthy of detailed description. Its interface with the syntax analyser is interesting, however, and is described briefly.

The symbols recognised by the lexical analyser are classified into three classes:

(a) Metalinguistic symbols: the symbols described in section 3.1.2.

(b) Production Identifiers.

(c) Other symbols. These, in turn, consist of two groups:

  (1) Beacons: keywords and special symbols delimiting or separating major constructs, such as DEFINITION, ENDREC and semicolon.

  (2) Ordinary symbols.

Tokens are used to represent these symbols.

Usually, the syntax analyser requests the lexical analyser to attempt to get a specific symbol of a specific class, by means of a boolean function. The lexical analyser in fact gets the next symbol, whatever it is. If it is the symbol desired, it is skipped by the lexical analyser, and the function value is true; otherwise it is not skipped (so that it will be considered the "next symbol" again), and the function value is false.

A further refinement occurs in the case of beacons. Whenever a beacon is requested, a context set of beacons is specified as well. The lexical analyser then skips and flags as superfluous any symbols encountered before the desired beacon or a beacon in the context set, and then takes the same action on the beacon reached as described above.

The approach used has the following advantages:

(a) The syntax analyser is totally independent of the workings and data structures of the lexical analyser.

(b) The syntax analyser can test and bypass a symbol by means of a single, natural function call. This combined operation greatly simplifies the syntax analyser.

(c) The manner in which beacons are handled greatly facilitates error recovery, as described in section 5.3.2.

5.3 THE SYNTAX ANALYSER

The syntax analyser works by recursive descent. It contains separate sets of routines for analysing the following categories of constructs:

(a) Declarations;

(b) Expressions;

(c) Statements;

(d) Syntax form rules;

(e) Syntax interpretation rules;

(f) The first pass;

(g) Subsequent passes; and

(h) The definition as a whole.

These routines analyse the appropriate constructs, build up the internal structures representing them, and then invoke the macro generator to generate the corresponding parser constructs. The macro generator is thus directly invoked as soon as it is needed to handle a construct, so the internal form of constructs need not be kept long. The routine for handling syntax form rules also invokes the CFSM generator.

As in the case of the lexical analyser, the internal workings of the syntax analyser are not worthy of description. The one aspect of recursive descent analysers which usually causes trouble, viz. error recovery, is described briefly, however.

5.3.1 ERROR RECOVERY

The origin of error recovery problems in recursive descent analysers is that an error detected at a low level, while analysing a minor sub-construct, may make further processing of major constructs containing it undesirable or even impossible. This is for either or both of the following reasons:

(a) Because of the error, all symbols within the minor construct may not have been bypassed. If these symbols are then analysed as part of the major construct, spurious error messages and strange effects may result.

(b) Internal structures normally set up by the low-level routine may be used in the higher level one processing an enclosing construct. If they are not set up because of the error, such use will be meaningless or even illegal.

A common solution is for each high-level routine to test for errors after every invocation of a lower level one, and to exit immediately if one was detected by the lower level routine. Certain very high level routines then skip input symbols until a suitable beacon is reached, completing error recovery. This method is inordinately clumsy, and also has the unfortunate effect that, if an error occurs in a declaration, the symbol table entries associated with that declaration will not be set up, leading to yet more errors later on.

The approach used in the parser generator has three aspects:

(a) The handling of beacons;

(b) Dummy internal structures; and

(c) Extreme Modularity.

These are now described.

5.3.2 BEACONS

Most constructs, even simple, low-level ones, contain identifying or delimiting symbols, and, whenever possible, these are treated as beacons. Whether an error has occurred or not, the lexical analyser is always requested to find them by means of beacon-handling functions described in section 5.2. This ensures that any superfluous symbols are ignored, solving problem (a) above.

Symbols are treated as beacons only in contexts where they may reasonably occur as such. This ensures that stray beacons totally out of context do not upset error recovery. The context sets described in section 5.2 allow the syntax analyser to specify explicitly which symbols are to be regarded as beacons. A syntax analyser routine to analyse a major construct generally takes a context set as a parameter, specifying the context in which the construct occurs.

An unfortunate consequence of this approach is that an entire major construct totally out of context (e.g. a record definition in a statement) will not be analysed as such. Worse still, its components may be analysed in the context in which they occur, rather than as its components (e.g. the field definitions as statements), leading to curious results.

One solution to this problem is for the lexical analyser routine that skips symbols when searching for beacons to trap the few symbols which bracket major constructs, and cause them to be analysed rather than skipped. This has not been implemented, but is listed as a desirable improvement in section 5.9.

A serious problem affecting the implementation of the above method was the Pascal limitation on the number of set elements allowed: 48 in the case of the compiler used. Representation of context sets as other than Pascal sets would have been far too clumsy, so only 48 beacons are used. Consequently, beacon routines could not be used as frequently as desired. Only syntax form rules, expressions, and statements not involving statement sequences are affected, an error within one of them causing rather too many symbols to be skipped.

5.3.3 DUMMY INTERNAL STRUCTURES

A routine invoked to analyse a construct will be expected to set up the internal form of that construct. If an error is encountered, it will recover immediately by setting up dummy, but valid, internal structures. Whatever information was obtained about the construct before the error occurred, is used; anything indeterminate is marked as ILLDEFINED. This ensures that any processing of the internal structures will be valid and meaningful, whatever errors occurred, alleviating problem (b) in section 5.3.1.

This method of error recovery is particularly effective in the case of declarations. Consider, for example, the erroneous declarations

```
VAR I, J ENDVAR
```

Though no type identifier is specified, symbol table entries are

nonetheless created for I and J, indicating that they are variables of ILLDEFINED type. Wherever they are used as variables, they will be accepted, and no type mismatch will be reported. This significantly reduces proliferation of error messages without leaving errors unflagged.

5.3.4 EXTREME MODULARITY

Every construct in the definition, however minor or insignificant, is analysed by a separate syntax analyser routine. Apart from enhancing clarity and modifiability, this has an effect on error recovery. If an error is detected anywhere in a procedure, a jump to the end of the procedure serves to bypass all remaining analysis of that construct, without bypassing any other construct. The handling of beacons ensures that any symbol belonging to the erroneous construct will be skipped eventually.

Most syntax analyser routines therefore contain a few statements at the end dealing with errors and error recovery. The manner in which they are separated from the rest of the routine enhances clarity, by totally separating error recovery from error-free processing.

5.4 THE CFSM GENERATOR

The primary function of the CFSM generator is to generate the internal, linked representation of the CFSM from the BNF rules. Once it has done so, it also generates the goto table and the error recovery tables, using this internal structure.

The technique used to generate the CFSM is based on that described by Aho and Johnson in [1]. The theory behind it is expounded and proved by De Remer in [5]. Space does not permit a full, detailed description and proof of the technique here. Instead, a short, intuitive introduction is given, followed by an outline of how the technique was implemented.

5.4.1 INTUITIVE INTRODUCTION

A path through the CFSM represents terminal symbols or constructs recognised. A particular state represents a "state of the parser"; an indication of what progress the parser has made in recognising constructs, and, ultimately, in recognising the whole program. This is now put in more concrete terms.

Suppose a production

p: <a> ::= $s_1 s_2 \dots s_n$

where each s_i is a terminal or non-terminal symbol, is one of the productions specified in the BNF rules. It defines a construct called <a>. If such a construct should occur in a valid context within a program being parsed, the parser would have to recognise each of the symbols or sub-constructs s_1, s_2, ..., s_n, and then perform a reduction by p. To parse in this way, the CFSM would have to contain the path shown in fig. 5.1.

If the parser is in state q_0 when the start of an <a> construct is encountered, the parsing actions described in section 2.5.1 will ensure that it is parsed. During such parsing, states q_0, q_1, ..., q_n, will build up on the state stack. When the reduction is performed, q_1, q_2, ..., q_n, will be popped, leaving q_0 at the top of

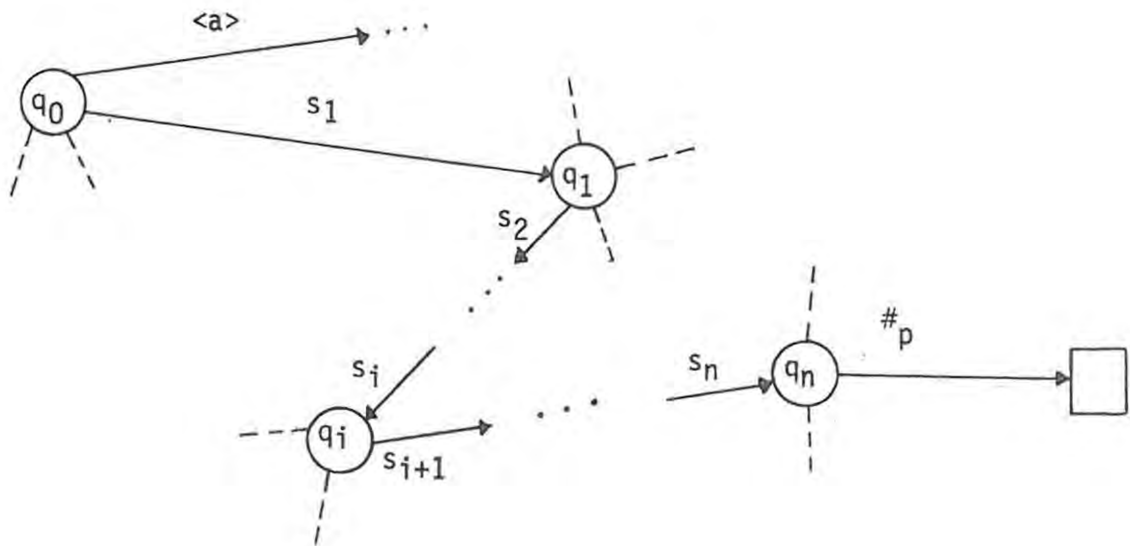

Fig. 5.1: A Path within a CFSM

the stack as it was when the recognition of the construct began. The transition under <u><a></u> will then be taken, and parsing will continue.

Fig. 5.1 is quite complete if each $\underline{s_i}$ is a terminal symbol. If however, the symbol $\underline{s_{i+1}}$ is the non-terminal <u><b></u>, then a <u><b></u> construct must be recognised before a transition under $\underline{s_{i+1}}$ is possible. In other words, when state $\underline{q_i}$ is entered, the transition under $\underline{s_{i+1}}$ cannot be taken immediately; rather, some other path must be followed, leading to the recognition of a <u><b></u> construct. Once such a path has been followed to the end and the construct has been recognised, the parser reverts to state $\underline{q_i}$ and takes the transition under <u><b></u>.

Suppose <b> is defined by means of the BNF rule

k. <b> ::= r_1 ! r_2 ! ... ! r_m @

where each r_i is a sequence of terminal and non-terminal symbols. Then m separate paths corresponding to the sequences r_1, r_2, ..., r_m must emanate from q_i, to allow for all possible varieties of <b> constructs. Furthermore, if the first symbol of the sequence r_j is a non-terminal <c_j>, a <c_j> construct would have to be recognised even before the first step towards recognising the jth variety of <b> construct could be taken. Thus a path corresponding to each possible variety of c_j construct would also have to emanate from q_i and so on to whatever depth is necessary.

This can be specified more rigorously by introducing the concept of closure sets defined as follows:

$$\text{closure}(\langle b\rangle) = \{\langle b\rangle\} \cup [\bigcup_{j=1}^{m} \text{closure}(c_j)]$$

where c_j is the first symbol of r_j and

$$\text{closure}(c) = \emptyset$$

if c is a terminal symbol. Closure(<b>) is thus the set of all non-terminal symbols which can appear at the start of sentential forms derivable from <b>. The state q_i must have paths emanating from it which can recognise all variants of all constructs specified by non-terminals in closure(<b>).

Such a proliferation of paths leaving q_i may cause the CFSM to be non-deterministic. To avoid this, all transitions from a state under the same symbol are combined into a single transition under that symbol to a single state.

The path shown in fig. 5.1 can be constructed automatically from the production p by means of a technique using items (also called configurations). An item is a production with a special marker symbol

(a slash is used here) inserted somewhere on its right hand side, and enclosed in square brackets for clarity. For example:

$[\ \langle a\rangle ::= /s_1 s_2 \ldots s_n\ ]$

$[\ \langle a\rangle ::= s_1 / s_2 \ldots s_n\ ]$

.

.

.

$[\ \langle a\rangle ::= s_1 s_2 \ldots s_n /\ ]$

Intuitively, the slash indicates "how far one has progressed in recognising the symbols on the right hand side of the production". The first one shown, with the slash preceding the first symbol, is called an <u>initial item</u>, and indicates that all symbols on the right hand side have yet to be recognised. This is precisely the situation when the parser is in state $\underline{q_0}$, establishing an association between the initial item and $\underline{q_0}$. By the same token, the item

$[\ \langle a\rangle ::= s_1 s_2 \ldots s_i / s_{i+1} \ldots s_n\ ]$

is associated with $\underline{q_i}$. This item is termed a <u>goto item</u>, since it is derived by following the path $\underline{q_0}\underline{q_1} \ldots \underline{q_i}$, which "goes to" $\underline{q_i}$. However, as previously established, paths for analysing $\langle \underline{b} \rangle$ constructs also emanate from $\underline{q_i}$, so all the initial items derived from productons with $\langle \underline{b} \rangle$ on the left hand side, viz:

$[\langle b\rangle ::= /r_1]$

$[\langle b\rangle ::= /r_2]$

.

.

.

$[\langle b\rangle ::= /r_m]$

are also associated with $\underline{q_i}$. In fact, all initial items derived from productions with members of <u>closure</u>($\langle \underline{b} \rangle$) on their left hand sides are associated with $\underline{q_i}$. All these initial items are termed <u>closure</u> <u>items</u>.

In general, a state such as $\underline{q_i}$ may lie on many paths within the CFSM, and hence have many goto items associated with it. The set of all such goto items, together with all the closure items derived from them, defines $\underline{q_i}$. The CFSM generation technique, now described, involves computing these sets of items and hence the associated states of the CFSM.

5.4.2 THE CFSM GENERATION TECHNIQUE

Firstly, closure sets for all non-terminal symbols are computed by means of a recursive algorithm. Since there may be more non-terminals than permissible elements in a Pascal set, closure sets are represented as arrays of Pascal sets, operated on by special routines.

Secondly, all possible items are set up in an array. The initial items associated with all productions appear first in the array, followed by goto items. Each item is linked to its successor, establishing chains of items akin to the paths in the CFSM to which they give rise.

The generation process proper then begins by creating the initial state, state 1, and associating with it the item

[<ACCEPT> ::= /<g>⊣]

where <g> is the goal symbol in the definition. This item eventually gives rise to the shift action on the end-marker and the accept action in the parser. For convenience, it is considered a goto item. The entire CFSM is then generated by performing the following for every state created.

A state $\underline{q}$ which has been created has goto items associated with it, but nothing else. The present algorithm uses them to determine all transitions from $\underline{q}$, as follows:

(a) For every item of the form

$$[\text{<a>} ::= s_1 s_2 \ldots s_n /]$$

n ⩾ 0, a reduce action by the production

$$\text{<a>} ::= s_1 s_2 \ldots s_n$$

(i.e. a transition under the appropriate # symbol) is associated with $\underline{q}$. For every item of the form

$$[\text{<a>} ::= s_1 s_2 \ldots s_i / s_{i+1} \ldots s_n]$$

$o \leqslant i < n$, its successor

$$[\text{<a>} ::= s_1 s_2 \ldots s_i s_{i+1} / s_{i+2} \ldots s_n]$$

is attached to a chain of goto items associated with symbol $\underline{s_{i+1}}$. Furthermore, if $\underline{s_{i+1}}$ is a non-terminal, closure($\underline{s_{i+1}}$) is added to a united closure set for $\underline{q}$.

(b) Step (a) is repeated for all $\underline{q}$'s closure items (i.e. all initial items derived from productions having symbols in the united closure set on their left hand sides).

(c) At this stage, all reduce actions have been generated for $\underline{q}$. Shift and goto actions are determined using the chains of goto items attached to terminal and non-terminal symbols, respectively. If such a chain of items is identical to the chain of goto items associated with an existing state, a transition to that state under the appropriate symbol is generated; otherwise a new state is created, and the transition is to the new state.

(d) Having generated all the actions for $\underline{q}$, the CFSM generator checks for LR(0) conflicts, and whether or not they are resolved by means of qualifiers. It reports unresolved or incompletely resolved conflicts in the listing.

The above steps are repeated for every state that has been created.

Since a unique state is associated with each set of goto items, and there is only a finite number of such sets, the iteration must terminate.

5.4.3 AUXILIARY STRUCTURES

An important property of LR(0) states gives rise to a useful auxiliary data structure maintained by the CFSM generator: although a state, $\underline{q}$, may be reached from several other states, this is always through a transition under the same symbol. Thus with every state is associated a unique symbol labelling all transitions to that state. The CFSM generator maintains an inversion of this: for every symbol, the list of states to which transitions under that symbol can lead.

This structure is used to reduce the number of states examined in step (c) in the previous section, and also to generate the goto and error recovery tables as described in the next two sections. To facilitate the generation of these tables further, the CFSM generator also sets up backward links between states, in the opposite direction to transitions.

5.4.4 GENERATION OF THE GOTO TABLE

Goto actions as represented in the internal CFSM are associated with each state. As described in section 4.11.6, however, the goto table itself is the inversion of this, consisting of entries grouped by symbol. The process of inverting the goto table is facilitated by the auxiliary structures described in the previous section, and is not described further.

5.4.5 GENERATION OF THE ERROR RECOVERY TABLES

The central task involved in generating the error recovery tables is generating TABLEOFPATHSTOBEACONS, described in section 4.11.7. The other tables are merely summaries of information already in this table, and are computed together with it.

Paths are specified in TABLEOFPATHSTOBEACONS as a number of separate entries, each specifying an individual transition. These entries are generated backwards using the auxiliary structures described in section 5.4.3. The algorithm used always generates the shortest possible paths.

5.5 THE MACRO GENERATOR

The macro generator is responsible for generating the parser and LATABLEFILE from the internal structures described in section 5.1, set up by the syntax analyser and CFSM generator. It does so by expanding macros, called skeletons, inserting information derived from these structures as requested by means of calls in the skeletons.

The macro generator is invoked by the syntax analyser at various times during analysis. In general, it is invoked immediately after a construct has been recognised, to generate the Pascal equivalent of the construct. In the case of a large construct containing smaller ones, such as the definition itself, it is invoked both when the start and end of the construct are recognised. Each invocation specifies a skeleton to be expanded, and certain arguments to be used during the expansion, giving information about the construct concerned, and called arguments of the skeleton. Appendix 4.1 lists all the points

at which the macro generator is called, together with the associated skeleton and arguments.

Though the macro generator is invoked by the syntax analyser, the syntax analyser remains entirely independent of the form of the generated parser. In fact, the skeletons specify the form of the parser entirely, so even the macro generator itself is independent of it. This has the significant advantage that generated parsers can be changed substantially without any change whatever being necessary to the parser generator program.

Space does not permit a full description of the macro generator and the wide variety of facilities it offers. This section is a description of the basic principles underlying it, and an overview of the operations it can perform. A full list of these operations appears in Appendix 4.

5.5.1 SKELETONS

When the macro generator is invoked by the syntax analyser, it expands the skeleton specified. It is therefore appropriate to begin the description of the macro generator with a description of skeletons.

A skeleton consists of text characters to be output directly, interspersed with calls requesting the macro generator to perform operations. Each call has the form

$\$pa_1a_2...a_n;$

where p specifies an operation and a_1, a_2, ..., a_n are arguments to be used in performing that operation. p is either a letter, or a letter followed by a number. An example is G1, which requests the macro generator to generate the natural number specified by the first

argument, a_1. One form of argument is an integer literal, denoted by the letter N followed by the value. The call

$G1N53;

therefore requests the macro generator to generate the integer 53. It has the same effect as the number 53 simply appearing in the skeleton, in which case it would be copied.

As mentioned above, the syntax analyser may specify arguments of the skeleton to be used when expanding a skeleton. They may be, and invariably are, used as arguments in operators specified within the skeleton. The _i_th one is denoted by A_i_. For example

$G1A2;

causes the natural number specified as the second argument of the skeleton to be generated.

Another important operator is Z, which causes expansion of the skeleton to cease. The call

$Z;

must therefore appear at the end of every skeleton (it may also be used within _include_ options; see section 5.5.6).

Some other operators and arguments are described in the sequel. All are listed in appendix 4.

5.5.2 CLASSES OF SKELETONS

There are two classes of skeletons called _array skeletons_ and _file skeletons_, according to where they are stored.

Array skeletons are efficient to access, and order of access is immaterial. An array skeleton may call other array or file skeletons, causing them to be expanded. Such calls may be nested and even recursive, to any depth. Array skeletons are numbered, and skeleton _n_ is called by the call

$\$Sna_1 a_2 \ldots a_n;$

For example, if array skeleton 14 is

$G1A2;$Z;

then the call

$S14N10N20N30;

causes the natural number 20 to be generated.

Array skeletons are stored in the array ASKELARR, not necessarily in numerical order. They are read into this array from TABLEFILE during initialisation of the parser generator, and remain there throughout.

File skeletons are also numbered and are stored on the file SKELTNFILE in numerical sequence. They are less efficient to access, particularly out of sequence, as every out-of-sequence access causes the entire file to be read from the beginning. A file skeleton may not call a file skeleton; it may call an array skeleton, provided that skeleton does not call a file skeleton directly or indirectly.

A call to file skeleton _n_ has the form

$\$Fna_1 a_2 \ldots a_n;$

File skeletons are clearly less efficient and much less versatile than array skeletons. They are intended only for large skeletons used infrequently. The present system in fact uses only two: one for passes other than pass 1, the second for the entire parser, including all the standard declarations and routines.

5.5.3 OUTPUT

The generated parser goes to the file PARSER. However, even the global constant definitions near the start of the parser can only be generated once the end of the definition has been reached, when, for example, all string literals will have been counted. Yet the macro generator is called as each individual construct is recognised, and it must generate the parser equivalent then and there, for space does not permit the storage of all constructs until the end.

The approach followed is that generated output corresponding to such constructs is sent to an intermediate file, from which it can be copied later. Several levels of intermediate files may be required, depending on the nature of the parser and the design of the skeletons. The current system uses two: one to contain parts of passes other than pass 1, used to fill file skeleton 1, the other to contain parts of the parser as a whole, used to fill file skeleton 2. The parser generator therefore implements two, but could easily be extended to handle more.

The macro generator treats the output files as a stack. Initially, the stack has just the file PARSER on it. The so-called pseudo-operation

    $P5;

causes a new intermediate file to be pushed onto the stack. All generated output then goes to it.

Generated output to an intermediate file must be sent to numbered blocks. The macro generator is requested to start a new block numbered n by means of the pseudo-operation

    $P7Nn;

Numbers allocated to blocks must be ascending, though not necessarily

consecutive.

Once output to an intermediate file is complete, it can be popped by means of the pseudo-operation

$P6;

At any time until another pop or push operation is performed, which destroys the contents of the file, its block n may be copied to the output file now at the top of the stack by means of the call

$Bn;

This, then, is the mechanism by which intermediate results can be generated for later inclusion in major constructs. A further refinement is provided, however, to cater for cases where the detailed generation of a construct depends on information appearing only later in the definition. A case in point is the TRYREDUCE routine associated with a qualifier, which contains one statement which depends on whether the associated production has a syntax interpretation action specified for it in some pass or not.

The refinement is that, if block n is generated with the appropriate calls, including

$Z;

at the end, it can be expanded as a skeleton by means of the call

$\$Jna_1 a_2 \cdots a_n;$

instead of merely being copied. Such generation is subject to the restriction that a block cannot call a block directly or indirectly. As the character "$" in a skeleton is always treated as signalling a call, generating one in a block presents a problem. To overcome it, the special operation

$G5;

generates a dollar sign. The call

$Z;

is therefore generated by

$G5;Z;

Generation from blocks should be used only when essential, for efficiency reasons. The current system uses it only twice.

5.5.4 ARGUMENTS

Thus far, only two kinds of arguments have been described, viz:

(a) higher-level arguments, denoted by An; and

(b) integer literals, denoted by Nn.

Others are:

(a) variable values, denoted by Vn;

(b) constants, denoted by Kn; and

(c) temporary locations, denoted by Tn.

Variable values provide access to the various internal structures. For example, V4 stands for STRGLITTABLE, V7 for the number of passes specified in the definition, and V35 for the number of states in the CFSM. If a CFSM has 42 states, therefore, the skeleton extract

LASTSTATE = $G1V35;;

will generate

LASTSTATE = 42;

A variety of constants is provided, such as all symbol and expression types. For example, K13 stands for a string literal, and K37 for a multiply expression. Constants are useful only in comparisons (see section 5.5.6).

Temporary locations provide for the storage of values for later use within skeletons. They are numbered, and at present the system makes 10 available. The call

    $P4Nia;

sets temporary location number _i_ to the argument _a_. T_i_ can then be used as an argument at any time and place, until its value is reset, and it will yield the value assigned to it. For example

    $P4N3N40;$G1T3;

is equivalent to

    $G1N40;

or just

    40

5.5.5 MODIFIERS

Variable values provide access to the parser generator's internal structures. Such access is also provided through the arguments passed to the macro generator by the syntax analyser, which are usually symbol table entries or expressions. Neither of these mechanisms, however, provides access to components of the structures. Such accesses are specified by means of _modifiers_.

A modifier has the form M_i_ and is, in essence, a postfix unary operator applying to an argument. A string of modifiers may follow any argument, and are applied from left to right. For example, M11 acts on a symbol table, and specifies the last chained entry in that table (certain entries, such as those for variables and string literals, are chained). M24 acts on a symbol table entry for a string literal, and specifies the unique number associated with the literal. The skeleton extract

STRINGLITERAL[$G1V4M11M24;]

therefore generates a reference to the last string literal yet encountered.

In general, a modifier is provided for accessing every pertinent field of every internal structure described in section 5.1. There are also some general modifiers, such as M1 which causes 1 to be added to the argument it modifies, and M9 which yields the number of digits in the argument it modifies. For example,

$G1N9M1M9M1;

generates "3". A full list of modifiers appears in appendix 4.10.

5.5.6 INCLUDE OPTIONS

In many contexts within the parser, generation is conditional on certain factors. For example, all declarations, procedures and statements involving partial parses are generated only for multipass parsers. Selective generation is achieved by means of include options, which are effectively if-then-else-endif constructs. The if-then is specified by an I*n* operator, else (otherwise) by O*n* and endif by E*n*. *n* indicates the type of condition on which selection is to be based and is the same for all three operators in the same construct. For example,

$I5$a_1$ a_2;X$O5;Y$E5;

causes "X" to be generated if $a_1 < a_2$, and "Y" otherwise.

The usual relations can be tested, as well as the truth of boolean arguments, whether pointers are null or not, and some special conditions. The else part, O*n*, may be omitted if not required.

Include options are essential to the implementation of recursive skeletons, providing the exit mechanism. An example is array skeleton 202 which is used to generate the list of partial parse files used:

```
PASS$G1A1;PARTIALPARSE%
$I5A1V7;%
    , $S202A1M1;%
   $E5;%
$Z;%
```

(The percent signs indicate that, though the skeleton is written on separate lines for clarity, only one line is to be generated; see section 5.7.1). The call

```
$S202N2;
```

causes the list

```
PASS2PARTIALPARSE,  PASS3PARTIALPARSE ...
```

to be generated, up to the last pass in the definition. Such a call presupposes that it has already been determined that there are at least two passes.

The operator Z is often used within an include option to terminate expansion.

5.5.7 TEMPORARY VARIABLES

As described in section 4.7, temporary variables are required in the parser for the implementation of certain statements. Since these are entirely a property of the parser rather than of the definition, the syntax analyser has no knowledge of them. They are handled completely by the macro generator, as instructed by calls within skeletons, using a temporary variable table.

The pseudo-operation P8 is used to create a new entry in the table, its arguments specifying details. Variable value V6 provides access to the table, and modifiers are provided for accessing fields within it.

5.5.8 OTHER OPERATIONS

Many operations not yet described are also provided by the macro generator. Some specify generation of various entities, such as symbols (including the digits used to make them unique) and production identifiers. Some control formatting, such as indentation, tabulation and line changes. The generator automatically splits lines that are too long, always at a space. The operator H specifies that generation of the parser must be interrupted to allow generation of LATABLEFILE. C acts like a _case_ construct, selecting a skeleton to expand based on the arguments specified.

All operations are listed and briefly described in appendix 4.

5.6 THE GENERATION OF GLOBALTABLEFILE

Once an entire definition has been analysed, STRGLITTABLE contains all the string literals used in it. They are output to GLOBALTABLEFILE.

5.7 TABLEFILE

TABLEFILE contains information used by the parser generator which is not coded in the program. This information falls into two categories:

(a) Symbols to be recognised by the lexical analyser; and

(b) Array skeletons.

The information is coded in source form, and then placed on the file in the required form by the program TABLEFILESETUP.

5.7.1 THE SOURCE FORM OF ARRAY SKELETONS

The following is a brief outline of the source form of array skeletons:

(a) Skeleton n is preceded by the line

```
%n
```

with the percent sign on the extreme left.

(b) Every skeleton line represents a line to be generated unless it contains a percent sign. In that case, all characters after the percent are ignored, and may be used as comments, and the following line is taken to be a continuation of the current line. Thus

```
A%  GENERATE A
B
$Z;
```

is equivalent to

```
AB
$Z;
```

and causes "AB" to be generated, then a new line to be started, whereas

```
A
B
$Z;
```

causes "A" and "B" to be generated on separate lines.

(c) Leading spaces are always ignored, allowing skeletons to be indented for clarity without the spaces being generated. Thus

```
A%
   B
```

is equivalent to

```
AB
```

If one or more leading spaces are to be generated, the first must be coded as an exclamation mark:

```
A%
! B
```

is equivalent to

```
A  B
```

The source form was designed to make the coding of array skeletons easy and readable, bearing in mind that they tend to be short and to contain more calls than simple text, and that the space available for their storage is limited. Since indentation in the source form is insignificant unless exclamation marks are used, indentation required in the generated output is requested by means of calls.

5.7.2 INITIALISATION OF THE PARSER GENERATOR

The very first action taken by the parser generator is to set up its internal structures, particularly ASKELARR, from the information in TABLEFILE. Since the amount of information involved is large, this is quite a time-consuming process. The generator therefore halts immediately after the initialisation is complete, so that it can be saved with its tables initialised. It can then be reloaded and resumed, without further initialisation being necessary.

Unfortunately, the halt had to be implemented by means of the machine-dependent instruction

```
ICL(161B, 0, 5151B);    (* SUSWT 2HII *)
```

It is the only machine-dependent instruction in the entire system.

5.8 SKELTNFILE

SKELTNFILE contains the file skeletons in numerical sequence. Like array skeletons, they are written in source form, and then are placed on the file in the required form by the program SKELTNFILESETUP.

5.8.1 SOURCE FORM OF FILE SKELETONS

The source form of file skeletons was also designed with clarity and readibility in mind. File skeletons differ from array skeletons in **that** they are much larger, tend to contain far more simple text than calls, and storage is not at a premium. Accordingly, indentation in the source form is significant, so explicit calls requesting it, which would be unduly clumsy, are unnecessary.

The source form is therefore identical to that for array skeletons, but for the fact that leading spaces are significant, except before calls which cause no generation, viz. E, I, L, O, P, R and Z.

5.9 EXTENSIONS AND IMPROVEMENTS

Many extentions and improvements to the parser generator are possible; the following are a few of the important ones:

(a) Implementation of weak beacons (section 3.8.8).

(b) Implementation of the extra definition language facilities mentioned in section 3.10 and parser improvements listed in section 4.15.

(c) Extension of the CFSM generator to generate at least SLR(1) or LALR(1) core parsers, rather than just LR(0). Many conflicts currently requiring qualifiers for their resolution would then be resolved automatically. Qualifiers would still be available to handle special cases.

(d) Extension of LATABLEFILE to contain an indication of which symbols must have their strings constructed and passed to the core parser (section 4.12.4).

(e) Improvement of parser generator error recovery, particularly within syntax form rules and statements, and as regards the handling of major constructs occurring out of context (section 5.3.2).

(f) Introduction of a paging system for array skeletons, so that only a few need be kept in main memory at a time, thus reducing the amount of space they occupy.

(g) Disposal of various structures, particularly symbol tables, built up by the parser generator. At present they are not destroyed when they are no longer needed.

(h) Improvement of the internal structures used for the processing of syntax form rules. When the present structures were designed and implemented, only an old version of Pascal was available, which did not implement the heap. Consequently, many structures are implemented as arrays which could better be implemented as records on the heap.

Most of these extensions were borne in mind when the parser generator was written, and the data structures are designed to facilitate them.

This fact, as well as the modularity of the parser generator, should make them fairly easy to implement.

6. TESTING AND CONCLUSIONS

Various routines and aspects of the parser generator system were tested as they were developed. These tests are of little interest. Once the entire system had been developed, however, some complete tests were conducted, and are described here.

The first was the most comprehensive, and listings of all files involved were obtained both for testing and illustration purposes. This test is described in detail in section 6.1. In the case of subsequent tests, only selected files were listed. They are described briefly in sections 6.2 to 6.6. Finally, conclusions drawn from the tests and the project as a whole are given in section 6.7.

6.1 TWO-PASS PARSER

The main test was the generation of a two-pass parser for a simple block-structure language, unimaginatively referred to as SBSL. The definition appears in appendix 8.1.1. Briefly, a block contains integer or real declarations, and statements. A statement is either an assignment statement or another block. Assignment statements contain simple expressions, allowing addition and multiplication. The syntax interpretation rules in pass 1 check that all variables used are declared, and that types are not mixed in expressions. To do this, they use various structures, including a stacked symbol table. For testing purposes, each syntax interpretation rule generates as output the associated production identifier. Pass two syntax interpretation rules generate the postfix equivalent of each assignment statement.

Though this language is admittedly unrealistically simple, and the output generated is not particularly meaningful, it nonetheless uses, and therefore tests, most of the features of the parser generator system. The structures used by the pass 1 syntax interpretation rules are, furthermore, similar to those that could be used in the analysis of more complex block-structure languages.

The contents of all output and temporary files produced by the parser generator are also in appendix 8.1.1. Once those had been produced, a simple lexical analyser was edited into the parser, and the parser compiled, loaded and saved. As indicated by the operating system (Maxibatch) messages at the end of the listing in appendix 8.1.1, the entire process thus far described took 121 mill seconds (on an ICL 1904S computer). The compiler listing at the end of appendix 8.1.1 indicates that the parser occupied 8573 words: quite small for a Pascal program.

The generated parser was then tested by allowing it to parse three programs: the first error-free, the second containing syntax interpretation errors and the third a variety of errors. The source programs, parser listings and generated output are in appendices 8.1.2 to 8.1.4, and can be verified by inspection to be correct.

If errors are detected in pass 1, output is nonetheless generated in pass 1, but pass 2 is not entered at all. The output generated in pass 1 after the first syntax form error would not normally be meaningful, but in the case of this test it indicates what structures were recognised and hence how effective error recovery was. Though the third program (appendix 8.1.4) contains some serious errors, the generated output is reasonable, and indicates that few constructs, even if out of context, were skipped.

6.2 ONE-PASS PARSER

The second test was a test of the generation of a one-pass parser. The definition is also of SBSL, very similar to that used in the previous test, but with no second pass. The generated parser was tested with the same three programs, and found to behave correctly. Appendix 8.2 contains listings of a selection of the files associated with this test.

6.3 SYNTAX AND SEMANTICS OF CONSTRUCTS

The third and fourth tests attempted to test as many as yet untested features of the parser generator system as possible. They use definitions which, between them, contain every type of declaration, statement and expression allowed, in a few different contexts. Unfortunately, they do not define any sensible language. In fact, the syntax interpretation rule in the third test is quite unexecutable, containing infinite loops and other undesirable constructs; it is purely a test of syntax. The rule in the fourth test, however, is legal, and is designed to test the semantics of most of the operations on structures. The output it generates is of the nature of monitors, continually showing the contents of the structures being manipulated.

The parser for the third test was produced, compiled and checked, though it could not be run. It appeared to be correct. That for the fourth test was run using two simple test programs. It produced a great deal of generated output, all found to be correct. Listings of selected files associated with these tests are in appendices 8.3 and 8.4.

6.4 CONFLICTS

The fifth test was of the generation of a parser with unresolved LR(0) conflicts. The test data was a definition of SBSL with all qualifiers removed. The correct conflict messages were produced in the parser generator listings, and the parser produced the correct definition semantic error. Listings are in appendix 8.5.

6.5 PARSER GENERATOR ERROR HANDLING

The sixth test was of error detection, reporting and recovery by the parser generator. Many errors were edited into a definition of SBSL on a fairly random basis, and the result was the input to the parser generator. The error messages are reasonable on the whole, though one associated with the syntax form rules is somewhat odd. Listings are in appendix 8.6.

6.6 DEFINITION SEMANTIC ERRORS

The final test was of definition semantic errors. When the definition of SBSL, on which most of the tests were based, was being written, an oversight caused the popping of an empty stack under certain conditions. This inadvertently erroneous definition was kept, and used as an illustration of definition semantic errors. Listings are in appendix 8.7.

6.7 CONCLUSIONS

The tests described above, though involving short definitions and short programs, between them test almost every feature of the parser generator system. In all cases, results were correct. Both parser generation and parsing were efficient, and parsers were of a reasonable size. It is regretted, however, that time did not permit the construction of a much larger test example, which would have given a better indication of size and efficiency, and of the usefulness of the definition language.

As already mentioned, many extensions to and improvements of the parser generator system described are possible. It is also intended that it be linked to a code generator generator system, thereby creating a complete compiler generator. Whether or not these modifications are made, it is hoped that the system will be of some use. In any event, it has demonstrated that the definition of complete syntax in the manner described is viable and practicable.

REFERENCES

1 Aho, A.V. and Johnson, S.C., LR Parsing, Computing Surveys 6(2), June 1974, pp. 99 - 124.

2 Aho, A.V. and Ullman, J.D., The Theory of Parsing, Translation and Compiling, Vol. 1: Parsing, Prentice Hall, Englewood Cliffs, NJ., 1972.

3 Backus, J.W. et al, Naur, P. (ed.), Revised Report on the Algorithmic Language ALGOL60, The Computer Journal 5, 1963, pp. 349 - 367.

4 Bulmer, A.R., The Automatic Generation of Code Generators with particular reference to Cobol, M Sc. thesis, Rhodes University, 1980.

5 De Remer, F.L., Practical Translators for LR(k) Languages, Project MAC Report MAC-TR-65, MIT, Cambridge, Mass., 1969.

6 De Remer, F.L., Simple LR(k) Grammars, CACM 14(7), July 1971, pp. 453 - 460.

7 Feldman, J.A., and Gries, D., Translator Writing Systems, CACM 11(2), Feb. 1968, pp. 77 - 113.

8 Ichbiah, J.D., and Morse, S.P., A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars, CACM 13(8), Aug. 1970, pp. 501 - 508.

9 Jensen, K., and Wirth, N., Pascal User Manual and Report, Springer Verlag, New York, 1978.

10 Knuth, D.E., Semantics of Context Free Languages, Mathematical Systems Theory 2(2), 1968, pp. 127 - 145.

11 Lecarme, A. and Bochmann, G.V., A (Truly) Usable and Portable Compiler Writing System, Proc. IFIP 74, Vol. 2, pp. 218 - 221.

12 Ledgard, H.F., A Formal System for Defining the Syntax and Semantics of Computer Languages, Project MAC Report MAC-TR-60, MIT, Cambridge, Mass., 1969.

13 McKeeman, W.M., Horning, J.J. and Wortman, D.B., A Compiler Generator. Prentice Hall, Engelewood Cliffs, NJ., 1970.

14 Van Wyngaarden, A. et al, Revised Report on the Algorithmic Language ALGOL68, Acta Informatica 5(1 - 3), 1975, pp. 1 - 236.

15 Williams, M.H., A Formal Notation for Specifying Static Semantic Rules, to appear in Computer Languages.

16 Williams, M.H., Static semantic features of Algol60 and Basic, The Computer Journal 21(3), 1978, pp. 234 - 242.

APPENDIX 1

DEFINITION LANGUAGE KEYWORDS

ACTION	ENDLS	FUNCTION	QUOTE
ADD	ENDPASS	GEN	RECORD
AFTER	ENDPROC	IF	REPEAT
ALTER	ENDPROD	IFF	SEARCH
AND	ENDPT	LENGTH	SFRULES
CLEAR	ENDREC	LSIS	SIRULES
COPY	ENDS	LSISNT	STACK
DECLARATIONS	ENDSB	MOD	STRONGBEACONS
DEFINITION	ENDSF	NEWLINE	SUBSTRING
DIV	ENDSI	NOT	TERMINATOR
DO	ENDTYPE	OF	THEN
ELSE	ENDVAR	OR	TOP
EMPTY	ENDWB	PASS	TREE
ENDACT	ERROR	POP	TRUE
ENDDECL	FAIL	POPUP	TYPE
ENDDEFN	FALSE	PROCEDURE	UNTIL
ENDDO	FOR	PRODUCTIONS	VAR
ENDFUN	FORALL	PSEUDOTERMINALS	WEAKBEACONS
ENDIF	FOUND	PUSH	WHILE
ENDIFF			

APPENDIX 2

BNF DEFINITION OF THE DEFINITION LANGUAGE

The definition language is defined below in BNF, augmented by the construct {X} which stands for X repeated zero or more times.

A2.1 The Definition

<definition> ::= DEFINITION <declarations> <pass 1> {;<pass i>} ENDDEFN

<pass 1> ::= PASS 1 <terminator decl> <pseudoterminal decl> <declarations> <sf rules> <si rules> ENDPASS

<pass i> ::= PASS <number> <declarations> <si rules> ENDPASS

A2.2 Declarations

<terminator decl> ::= TERMINATOR <character>;

<pseudoterminal decl> ::= PSEUDOTERMINALS <meta variable> {<meta variable>} ENDPT;

<declarations> ::= <empty> | DECLARATIONS <decls> ENDDECL;

<decls> ::= <type decls>; <var decls>; <proc decls> | <type decls>; <var decls> | <type decls>; <proc decls> |
<var decls>; <proc decls> | <type decls> | <var decls> | <proc decls>

<type decls> ::= TYPE <type decl> {; <type decl>} ENDTYPE

```
<type decl>          ::= <id> = <type>

<type>               ::= <id> | <flag type> | <record type> | <stack type> | <tree type>

<flag type>          ::= (<flag literal>{ ,<flag literal>})

    <flag literal>   ::= '<id>'

<record type>        ::= RECORD <field decl>{ ; <field decl> }ENDREC

    <field decl>     ::= <id>{ ,<id>}: <id>

<stack type>         ::= STACK OF <id>

<tree type>          ::= TREE OF <id>

<var decls>          ::= VAR <var decl> {; <var decl>} ENDVAR

<var decl>           ::= <id>{ , <id>}: <id>

<local var decls>    ::= <empty> | <var decls>;

<proc decls>         ::= <proc decl>{ ; <proc decl>}

<proc decl>          ::= PROCEDURE <id>; <local var decls><stat seq> ENDPROC
```

<u>A2.3 Syntax Form Rules</u>

```
<sf rules>           ::= SFRULES <productions><strong beacons><weak beacons>ENDSF;

<productions>        ::= PRODUCTIONS <bnf rule> {; <bnf rule>}ENDPROD

<bnf rule>           ::= <number> . <meta variable> ::=<right hand side>{<meta or><right hand side>}<meta terminator>
```

<right hand side > ::=< meta expression> | <meta expression> <metal terminator>< qualifier >

< meta expression > ::= <meta empty> | <meta symbol>{<meta symbol>}

< qualifier> ::= IFF < local var decls> <condition>< after clause> ENDIFF

< after clause> ::=< empty> | AFTER <stat seq>

<strong beacons> ::= ; STRONGBEACONS <meta symbol>{<meta symbol>}<meta terminator> ENDSB

<weak beacons > ::= ; WEAKBEACONS <meta symbol>{<meta symbol>}<meta terminator> ENDWB

A2.4 Syntax Interpretation Rules

<si rules> ::= SIRULES <action>{ ; <action >}ENDSI

<action > ::= <production id> {, < production id >}: ACTION < local var decls>< stat seq >ENDACT

<production id> ::= <number> < letter>

A2.5 Statements

<stat seq> ::= <stat>{ ; <stat >}

< stat > ::=< gen stat> | <error stat> | < transfer stat > | < procedure stat > | < push stat > | <add stat > | <popup stat> |
< clear stat> | <if stat> | < search stat > | <while stat> | <repeat stat > | <forall stat >

< gen stat> ::= GEN <gen exp> {, <gen exp >}

< gen exp> ::= <exp> | NEWLINE

<error stat> ::= ERROR <exp>

<transfer stat> ::= <exp>→<var>

<procedure stat> ::= <id>

<push stat> ::= <exp> PUSH <var>

<add stat> ::= <exp> ADD <var>

<popup stat> ::= POPUP <var>

<clear stat> ::= CLEAR <var>

<if stat> ::= IF <condition> THEN <stat seq><else clause> ENDIF

<else clause> ::= <empty> | ELSE <stat seq>

<search stat> ::= SEARCH <var> FOR <exp><found clause><fail clause> ENDS

<found clause> ::= <empty> | FOUND <stat seq>

<fail clause> ::= <empty> | FAIL <stat seq>

<while stat> ::= WHILE <condition> DO <stat seq> ENDDO

<repeat stat> ::= REPEAT <stat seq> UNTIL <condition>

<forall stat> ::= FORALL <var> DO <stat seq> ENDDO

A2.6 Expressions

<condition> ::= <exp>

```
<exp>                       ::= <bool term>{ OR< bool term>}
<bool term>                 ::= <bool factor>{ AND <bool factor>}
<bool factor>               ::= <bool primary> | NOT <bool primary>
<bool primary>              ::= <other exp> | <other exp>< relational operator>< other exp>
    <relational operator>   ::= = |<> | < |<= |>= | >
<other exp>                 ::= <negation operator> <term>{<adding operator>< term>}
    <negation operator>     ::= <empty> | -
    <adding operator>       ::= + | -
<term>                      ::= <factor>{<multiplying operator> < factor>} | <factor>{ & <factor>}
    <multiplying operator>  ::= * | DIV | MOD
<factor>                    ::= LENGTH(<exp>) | SUBSTRING(<exp>, <exp>, <exp>) | COPY(<exp>) | POP(< exp >) | EMPTY(<exp>) | <number> |
                                TRUE | FALSE | <flag literal> | QUOTE | <string literal> | <lookahead test> | <var> | <meta variable> | (<exp> )
<string literal>            ::= "{<character other than quote>}
<lookahead test>            ::= <lookahead operator> <meta symbol>{<meta or> <meta symbol >}<meta terminator> ENDLS
    <lookahead operator>    ::= LSIS | LSISNT
<var>                       ::= <id> | <var> . <id> | TOP(<var>)
```

A2.7 SYMBOLS

<number> ::= <digit> {<digit>}

<id> ::= <letter> | <id> <letter> | <id><digit>

<meta symbol> ::= <meta constant> | <meta variable>

<meta constant> ::= <character> {<character>}

<meta variable> ::= <meta open><meta var string> <meta close>

<meta var string> ::= <letter> | <meta var string><letter> | <meta var string><space>

<meta terminator> ::= <character>

The following cannot be defined in BNF:

<meta or> |

<meta open> <

<meta close> >

APPENDIX 3

ERROR MESSAGES

A3.1 DEFINITION SYNTAX ERRORS

1 ENDDECL missing

2 ENDTYPE missing

3 Identifier expected

4 Identifier already declared

5 = expected

6 Invalid or missing type

7 Undeclared identifier

8 Type identifier expected

9 OF expected

10 Tree entry not a record

11 First field of tree entry not a string

12 Flag literal already declared

13 : expected

14 Flag literal expected

16 ; missing

17 Superflous ;

18 PROCEDURE expected

19 ENDPROC missing

20) missing

21 ENDREC missing

22 Expression in ERROR statement not of type INTEGER or STRING

23 Variable in POPUP statement not of stack type

24 Obsolete

25 Left and right hand sides of transfer statement of incompatible types

26 Attempt to alter search field of tree entry

27 Variable in PUSH statement not of stack or tree type

28 Left and right hand sides of PUSH statement of incompatible types

32 Variable in ADD statement not of stack type

33 Left and right hand sides of ADD statement of incompatible types

34 Superfluous symbol/symbol out of context/symbol skipped during error recovery

35 Metalinguistic variable incomplete

36 Illegal symbol

37 Metalinguistic variable incomplete/illegal character within metalinguistic variable

38 String literal incomplete

39 Illegal character within flag literal

40 Flag literal incomplete/illegal character within flag literal

41 Too many errors in line

42 Too many different symbols with the same initial 6 characters

43 Recursive record type declaration

44 Line of generated parser too long (the generated parser should be examined to find the cause)

45 ENDVAR missing

50 ENDSF missing

51 PRODUCTIONS missing

52 ENDPROD missing

53 Production number expected

54 . expected

55 Metalinguistic variable expected

56 ::= expected

57 Metalinguistic terminator expected

58 Too many alternatives (right hand sides) in BNF rule

59 Too many productions (right hand sides in all BNF rules)

60 Production number zero or too large

61 Metalinguistic variable on left hand side of BNF rule has already appeared on the left hand side of a previous BNF rule

62 Too many symbols in productions (right hand sides in all BNF rules)

63 Metalinguistic constant or variable expected

64 Too many non-terminal symbols

65 Too many terminal symbols

66 Non-terminal symbol expected

67 Too many pseudoterminals

68 Pseudoterminal already declared

69 ENDPT missing

70 ENDSB missing

71 Undeclared metalinguistic constant

72 Metalinguistic constant already declared a strong beacon

73 Undeclared metalinguistic variable

74 Metalinguistic variable already declared a stong beacon

75 Definition incomplete (end of file reached)

76 Illegal character at start of metalinguistic variable

77 Superfluous metalinguistic symbol

78 SFRULES missing

79 ENDSF missing

80 Illegal metalinguistic terminator character

81 DEFINITION missing

82 PASS 1 missing

83 ENDDEFN missing

84 Pass number expected

85 Too many passes

86 Pass out of sequence

87 ENDPASS missing

88 Superfluous symbols at end of definition

89 Operands of OR not BOOLEAN

90 Operands of AND not BOOLEAN

91 Operands of NOT not BOOLEAN

92 ENDLS expected

93 Lookahead test invalid in current pass

94 ! expected

95 Operand of relational operator of invalid type

96 First operand of relational operator is flag literal

97 Operands of relational operator of incompatible types

98 Operand of negation operator (-) not INTEGER

99 Operand of + or - not INTEGER

100 Operand of * DIV or MOD not INTEGER

101 Operand of & not STRING

102 (expected

103 Unexpected symbol in expression

104 Operand of TOP not of stack type

105 Operand of LENGTH not STRING

106 First operand of SUBSTRING not STRING

107 , expected

108 Second operand of SUBSTRING not INTEGER

109 Third operand of SUBSTRING not INTEGER

110 Operand of COPY not of structure type

111 Operand of POP not of stack type

112 Operand of EMPTY not of stack type

113 Variable identifier expected

114 Variable expected

115 Field identifier expected

116 Variable on left hand side of . not of record type

117 Undeclared field

118 Undeclared flag literal

119 Metalinguistic variable may not be referenced

121 Variable in CLEAR statement not of stack or tree type

122 Invalid statement

123 Condition in IF statement not BOOLEAN

124 THEN missing

125 ENDIF missing

127 FOR missing

128 Expression after FOR in SEARCH statement not STRING

129 Variable in FORALL statement not of stack type

130 DO missing

131 ENDDO missing

133 Condition in WHILE statement not BOOLEAN

134 UNTIL missing

135 Condition in REPEAT statement not BOOLEAN

136 Expression in GEN statement not INTEGER or STRING

137 Unsearchable variable in SEARCH statement

138 SIRULES missing

139 ENDSI missing

140 ACTION missing

141 ENDACT missing

142 Production identifier expected

143 Invalid production identifier

144 Condition in qualifier not BOOLEAN

145 ENDIFF missing

147 ENDPROC missing

148 Number too large

A3.2 LR PARSER GENERATION ERRORS

Each of these errors indicates that one of the Parser Generator's constants controlling the size of its internal structures is too small for the definition being processed. The error numbers and associated constants are listed below.

1 ITEMMAX

7 STATEMAX

8 REDTABMAX

12 NQTABMAX

A3.3 DEFINITION SEMANTIC ERRORS

1 Attempt to push entry onto unsearched tree

2 Attempt to push entry onto tree after successful search

3 Attempt to push entry onto tree with search field different from the string on which the tree was searched

4 Second operand of SUBSTRING negative

5 Third operand of SUBSTRING zero or negative

6 Substring requested extends beyond end of string

7 State stack overflow (increase STATSTKMAX)

8 Qualifiers not mutually exclusive

9 Attempt to pop empty stack

10 Conflict state reached

APPENDIX 4

MACRO GENERATOR CALLS, OPERATIONS, ARGUMENTS AND MODIFIERS

A4.1 Calls Issued by the Syntax Analyser

All calls to the Macro Generator issued by the Syntax Analyser are listed below, together with the numbers of the skeletons to be expanded and the arguments supplied. An argument is described by giving its type and an indication of its source, which may be understood by referring to the description of the appropriate definition language construct in Chapter 3.

Context of Call	Skeleton	Arguments
Start of definition	21	None
End of definition	22	None
Start of pass 1	23	None
End of pass 1	24	None
Start of pass i	96	None
End of pass i	97	None

Context of Call	Skeleton	Arguments
Start of type declarations	1	None
Between type declarations	2	None
End of type declarations	3	None
Identity type declaration	4	SYMTABENTRY y_1, SYMTABENTRY y_2
Flag type declaration	5	SYMTABENTRY y_1, EXPRESSION ('w_1', 'w_2', ..., 'w_n')
Start of record type declaration	6	SYMTABENTRY y
End of record type declaration	7	SYMTABENTRY y
Stack type declaration	8	SYMTABENTRY y_1, SYMTABENTRY y_2
Tree type declaration	9	SYMTABENTRY y_1, SYMTABENTRY y_2
Start of variable declarations within block of declarations	12	None
End of variable declarations within block of declarations	13	None
Start of local variable declarations within action	90	None
End of local variable declarations within action	91	None
Start of local variable declarations within qualifier	92	None
End of local variable declarations within qualifier	93	None
End of local variable declarations within procedure	95	None

Context of Call	Skeleton	Arguments
Between variable and field declarations	10	None
Variable or field declaration	11	EXPRESSION (v_1, v_2, ..., v_n), SYMTABENTRY y
Start of procedure declarations	83	None
Between procedure declarations	84	None
End of procedure declarations	85	None
Start of procedure declaration	86	SYMTABENTRY p
End of procedure declaration	87	None
Start of productions	14	None
End of productions	15	None
Start of qualifier	80	None
Qualifier condition	81	EXPRESSION c
End of qualifier	82	None
Immediately after generation of the internal LR parser	16	None
Start of strong beacons	17	None
Between strong beacons	18	None

Context of Call	Skeleton	Arguments
End of strong beacons	19	None
Strong beacon	20	SYMNO m_i
Start of syntax interpretation rules	75	None
Between actions	76	None
End of syntax interpretation rules	77	None
Start of action	78	PRODNO p_1
End of action	79	None
Start of statement sequence in procedure declaration	72	None
Between statements in procedure declaration	73	None
End of statement sequence in procedure declaration	74	None
Start of statement sequence in qualifier	69	None
Between statements in qualifier	70	None
End of statement sequence in qualifier	71	None
Start of statement sequence in action	66	None
Between statements in action	67	None

Context of Call	Skeleton	Arguments
End of statement sequence in action	68	None
Start of statement sequence in THEN clause	45	None
Between statements in THEN clause	46	None
End of statement sequence in THEN clause	47	None
Start of statement sequence in ELSE clause	48	None
Between statements in ELSE clause	49	None
End of statement sequence in ELSE clause	50	None
Start of statement sequence in FOUND clause	51	None
Between statements in FOUND clause	52	None
End of statement sequence in FOUND clause	53	None
Start of statement sequence in FAIL clause	54	None
Between statements in FAIL clause	55	None
End of statement sequence in FAIL clause	56	None
Start of statement sequence in WHILE statement	60	None
Between statements in WHILE statement	61	None
End of statement sequence in WHILE statement	62	None
Start of statement sequence in REPEAT statement	63	None

Context of Call	Skeleton	Arguments
Between statements in REPEAT statement	64	None
End of statement sequence in REPEAT statement	65	None
Start of statement sequence in FORALL statement	57	None
Between statements in FORALL statement	58	None
End of statement sequence in FORALL statement	59	None
Start of GEN statement	25	None
NEWLINE	26	None
Expression in GEN statement	27	EXPRESSION e_i
End of GEN statement	28	None
ERROR statement	29	EXPRESSION e
Transfer statement	42	EXPRESSION e, EXPRESSION v
Procedure statement	88	SYMTABENTRY p
PUSH statement	43	EXPRESSION e, EXPRESSION v
ADD statement	44	EXPRESSION e, EXPRESSION v
POPUP statement	30	EXPRESSION v
CLEAR statement	31	EXPRESSION v

Context of Call	Skeleton	Arguments
Start of IF statement	32	EXPRESSION c
End of IF statement	33	None
Start of SEARCH statement	34	EXPRESSION v, EXPRESSION e
End of SEARCH statement	35	None
Start of WHILE statement	38	EXPRESSION c
End of WHILE statement	39	None
Start of REPEAT statement	40	None
End of REPEAT statement	41	EXPRESSION c
Start of FORALL statement	36	EXPRESSION v
End of FORALL statement	37	None
When lexical analyser tables must be generated	89	None

A4.2 OPERATIONS

$Ana_1 \ldots a_m$	Expand array skeleton specified by argument _n_
Bn	Copy block _n_ from intermediate file
$Cna_1 \ldots a_m$	Expand array skeleton from collection based on criterion _n_
En	End of include option _n_
$Fna_1 \ldots a_m$	Expand file skeleton _n_
$Gna_1 \ldots a_m$	Call generate procedure _n_
H	Generate lexical analyser tables
$Ina_1 \ldots a_m$	Include option _n_
$Jna_1 \ldots a_m$	Expand block _n_ from intermediate file as skeleton
L	Decrease indentation (left)
On	Otherwise part of include option _n_
$Pna_1 \ldots a_m$	Pseudo-operation _n_
R	Increase indentation (right)
$Sna_1 \ldots a_m$	Expand array skeleton _n_
U	Generate end-of-line
W	Generate a space followed by the number of the current pass
X	Generate the number of the current pass
Ya_1	Generate symbol corresponding to SYMTABENTRY a_1
Z	Stop expansion of current skeleton

A4.3 COLLECTIONS

	Alternatives	Type of a_2
C1	SYMTYPE = INTEGERTYPE, BOOLEANTYPE, FLAGTYPE, STRINGTYPE, RECORDTYPE, STACKOFRECORDSTYPE, STACKOFOTHERTYPE, TREETYPE	SYMTABENTRY
C2	Type simple, string or structure	SYMTABENTRY
C3	SYMTYPE = RECORDTYPE, STACKOFRECORDSTYPE, STACKOFOTHERTYPE, TREETYPE	SYMTABENTRY
C5	All RECORDOPERATIONS values (a skeleton is expanded for each of these occuring in RECOPSPERFORMED)	SYMTABENTRY
C6	All STOPERATIONS values, each repeated 3 times for STACKOFRECORDSTYPE, STACKOFOTHERTYPE and TREETYPE (a skeleton is expanded for each operation in STOPSPERFORMED, the particular one chosen depending on SYMTYPE)	SYMTABENTRY
C10	EXPKIND = any KINDOFEXPRESSION value	EXPRESSION
C11	EXPKIND = EBAR, EMETAVARIABLE, EMETACONSTANT	EXPRESSION

A4.4 GENERATE PROCEDURES

	Value Generated	Argument Types
G1	Natural Number	NATURAL
G2	Symbol string (excluding unique digit)	SYMTABENTRY
G3	Symbol string with "(*" or "*)" converted	SYMTABENTRY
G4	Production Identifier	PRODNO
G5	$	None
G6	Natural number a_1 with minimum length a_2	NATURAL, NATURAL
G7	%	None
G8	The number of spaces specified by a_1	NATURAL

A4.5 INCLUDE OPTIONS

	Condition on which to Include	Argument Types
I1	a_1 TRUE	BOOLEAN
I2	a_1 FALSE	BOOLEAN
I3	a_1 = a_2	Any, Any
I4	a_1 <> a_2	Any, Any
I5	a_1 < a_2	INTEGER
I6	a_1 <= a_2	INTEGER
I7	a_1 >= a_2	INTEGER
I8	a_1 > a_2	INTEGER
I9	a_1 NIL	Any pointer
I10	a_1 NOT NIL	Any pointer
I11	a_1 OR a_2 TRUE	BOOLEAN, BOOLEAN
I15	a_1 INTEGER, BOOLEAN or flag type	SYMTYPE

I16	a_1 structure type	SYMTYPE
I17	a_1 stack type	SYMTYPE
I20	Any SI actions for a_1 in any pass	PRODNO
I21	Any SI actions for a_1 in pass a_2	PRODNO, PASSNO
I22	Any SI actions for a_1 in passes 2 onwards	PRODNO
I26	Conflicts at a_1 resolved by means of qualifiers	STATENO

A4.6 PSEUDO-OPERATIONS

	Operation	Argument Types
P1	Skip to column $\underline{a_1}$	POSINT
P4	Set temporary location number $\underline{a_1}$ to value $\underline{a_2}$	POSINT, Any
P5	Push stack of intermediate files	None
P6	Pop stack of intermediate files	None
P7	Start block $\underline{a_1}$ on intermediate file	POSINT
P8	Create temporary variable table entry: $\underline{a_1}$ is number of skeleton to be expanded to generate reference to variable; $\underline{a_2}$ is type of variable	POSSINT, SYMTABENTRY
P9	Clear temporary variable table	None

A4.7 ARGUMENTS

An	Argument _n_
Kn	Constant _n_
Nn	The natural number _n_
Tn	Temporary location _n_
Vn	Variable value _n_

A4.8 CONSTANTS

SYMTYPE Constants

K1	SECONDARYSYMBOL
K2	INTEGERTYPE
K3	BOOLEANTYPE
K4	FLAGTYPE
K5	STRINGTYPE
K6	RECORDTYPE
K7	STACKOFRECORDSTYPE
K8	STACKOFOTHERTYPE
K9	TREETYPE
K10	LOCATION
K11	PROCEDRE
K12	FLAGLITERAL
K13	STRINGLITERAL
K14	METAVARIABLE
K15	METACONSTANT

METAVARREFSTATUS Constants

K21	CANTBEREFERENCED
K22	CANBEREFERENCED
K23	REFERENCED
K24	LHSREFERENCED

METASYMBOLSTATUS Constants

K26	STRONGBEACON
K27	WEAKBEACON
K28	ORDINARYTERMINAL
K29	NONTERMINAL

BOOLEAN Constants

K31	TRUE
K32	FALSE

KINDOFEXPRESSION Constants

K35	EPLUS
K36	EMINUS
K37	ETIMES
K38	EDIV
K39	EMOD
K40	EOR
K41	EAND
K42	EEQ
K43	ELT

K44	ELE
K45	ENE
K46	EGE
K47	EGT
K48	ECONCAT
K49	EFIELD
K50	ECOMMA
K51	EBAR
K52	EUNARYMINUS
K53	ENOT
K54	ETOP
K55	ELENGTH
K56	ESUBSTRING
K57	ECOPY
K58	EPOP
K59	EEMPTY
K60	ELSIS
K61	ELSISNT
K62	ELOCATIONID
K63	EINTEGERLITERAL
K64	ETRUE
K65	EFALSE
K66	EFLAGLITERAL
K67	EQUOTE
K68	ENULLSTRING
K69	ENEWLINE
K70	ESTRINGLITERAL
K71	EMETAVARIABLE
K72	EMETACONSTANT

A4.9 VARIABLE VALUES

Global Values

V1	SETMAX
V2	GLOBALIDTABLE
V3	FLAGLITTABLE↑.SYMTAB
V4	STRGLITTABLE
V5	METAVARTABLE
V6	TEMPVARTABLE.LASTTVARENTRY
V7	NOPASSES
V8	NOSTRINGLITERALS
V9	GLOBALDECLINFOREC.UDECLDTYPES
V10	GLOBALDECLINFOREC.UDECLDVARIABLES
V11	GLOBALDECLINFOREC.UDECLDROUTINES

PASS 1 Values

V15	PASS1IDTABLE
V16	PASS1DECLINFOREC.UDECLDTYPES
V17	PASS1DECLINFOREC.UDECLDVARIABLES
V18	PASS1DECLINFOREC.UDECLDROUTINES

SF Values

V19	METACONTABLE
V20	GOALSN
V21	NONTS
V22	LASTPT
V23	LASTT

V24	ANYSTRONGBEACONS
V25	ANYWEAKBEACONS
V26	FIRSTSTRONGBEACONNO
V27	LASTSTRONGBEACONNO
V28	FIRSTWEAKBEACONNO
V29	LASTWEAKBEACONNO
V30	TOKENSETSLONG
V31	TOKNSETLASTINTSET
V32	NOTOKENSETS
V34	NOPRODS
V35	NOQS
V36	STATESTACKMAX
V37	STATSETLASTINTSET
V38	GOTOTABMAX
V39	PATHTABMAX
V40	IFFCLAUSES
V41	LRCONFLICTS

PASS I Values

V50	NOPASSES
V51	PASSIIDTABLE
V52	PASISIACTIONTABLE
V53	PASSIDECLINFOREC.UDECLDTYPES
V54	PASSIDECLINFOREC.UDECLDVARIABLES
V55	PASSIDECLINFOREC.UDECLDROUTINES

Local Values

V58	LOCALIDTABLE
V59	SEMICOLNDETECTED

A4.10 MODIFIERS

INTEGER Modifiers: Effect

M1	+1
M2	-1
M4	- (negate argument)
M5	+T1
M6	-T1
M9	Set argument to number of digits in argument

BOOLEAN Modifiers: Effect

M3	NOT (negate argument)

Pointer Modifiers: Effect

M7	Set argument TRUE if NIL, and FALSE otherwise
M8	Set argument TRUE if not NIL, and FALSE otherwise

SYMTABROOT Modifiers: Field Selected

M11	LASTCHAINEDENTRY
M12	FIRSTLOCATIONENTRY
M13	LASTLOCATIONENTRY

M14	FIRSTINDEXNODE

SYMTABENTRY Modifiers: Field Selected

M15	SYMTYPE
M16	SECPRIMARYSYMENTRY
M17	FTLITENTRYOFFIRSTLITERAL
M18	RECFIELDTAB
M19	RECPREVSTRUCTURETYPEENTRY
M20	STENTRYPRIMARYTYPEENTRY
M21	STPREVSTRUCTUREETYPEENTRY
M22	LOCPRIMARYTYPEENTRY
M23	NEXTLOCATIONENTRY
M24	STRGLITNO
M25	MVARREFSTATUS
M26	MVARSYMSTATUS
M27	MVARGOTOTABSTARTINDEX
M28	MCONNO
M29	MCONSYMSTATUS
M30	NEXTENTRYINCHAIN
M31	FTLITRECOFFIRSTLITERAL

FLAGLITREC Modifiers: Field Selected

M33	PRIMARYTYPEENTRY
M34	UNIQUENO

TVARTABENTRY Modifiers: Field Selected

M35	ASKELNOTOGENERATEREFERENCE

M36 UNIQUENO

M37 PRIMARYTYPEENTRY

M38 PREVTVARENTRY

EXPRESSION Modifiers: Field Selected

M40 EXPTYPESYMTABENTRY

M41 LEFTOPERAND

M42 RIGHTOPERAND

M43 OPERAND

M46 LOCSYMTABENTRY

M47 INTLITVALUE

M48 STRGLITABENTRY

M49 METAVARTABENTRY

M50 FLAGLITTABENTRY

M51 APPROPFLAGLITREC

M52 METACONTABENTRY

M53 EXPKIND

PRODNO Modifiers: Field of PROD or PRODINFOTABLE Entry Selected

M55 SNAFIRSTX

M56 SNALASTX

M57 LHSSYMMNO

M58 IFFCLAUSE

M59 PRIMARYPRODFORACTION

M62 PRODIDNO

PRODSNAINDEX Modifiers: Result

M60 PRODSNA [argument]

SYMNO Modifiers: Result

M61 SYM [argument] (SYMTABENTRY for argument)

STATENO Modifiers: Field of STATE Entry Selected

M65 SHIFTFIRSTX

M66 SHIFTLASTX

M67 REDFIRSTX

M68 REDLASTX

M69 REDINFN

M70 CONFLICT

NQTABINDEX Modifiers: Field of NQTAB Entry Selected

M75 SYMN

M76 QN

REDTABINDEX Modifiers: Field of REDTAB Entry Selected

M77 INFN

INDEXNODE Modifiers: Field Selected

M80 LESSINDEXNODE

M81 GREATERINDEXNODE

M82 FIRSTENTRYINCHAIN

PASSISIACTIONENTRY Modifiers: Field Selected

M85 FIRSTPRODNO

M86 LASTPRODNORECINCHAIN

M87 NEXTPASSISIACTIONENTRY

PRODNOREC Modifiers: Field Selected

M90 PRODN

M91 PREVPRODNOREC