Aston University

Some pages of this thesis may have been removed for copyright restrictions.

If you have discovered material in Aston Research Explorer which is unlawful e.g. breaches copyright, (either yours or that of a third party) or any other law, including but not limited to those relating to patent, trademark, confidentiality, data protection, obscenity, defamation, libel, then please read our Takedown policy and contact the service immediately (openaccess@aston.ac.uk)

The Development of a

New Compiler-compiler Front-end

Nadia Bendjeddou

Doctor of Philosophy

The University of Aston in Birmingham

April 1989

© This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior, written consent.

The University of Aston in Birmingham

The Development of a New Compiler-compiler Front-end

Nadia Bendjeddou

Doctor of Philosophy
April 1989

Summary

The increasing cost of developing complex software systems has created a need for tools which aid software construction. One area in which significant progress has been made is with the so-called Compiler Writing Tools (CWTs); these aim at automated generation of various components of a compiler and hence at expediting the construction of complete programming language translators. A number of CWTs are already in quite general use, but investigation reveals significant drawbacks with current CWTs, such as lex and yacc. The effective use of a CWT typically requires a detailed technical understanding of its operation and involves tedious and error-prone input preparation. Moreover, CWTs such as lex and yacc address only a limited aspect of the compilation process; for example, actions necessary to perform lexical symbol valuation and abstract syntax tree construction must be explicitly coded by the user.

This thesis presents a new CWT called CORGI (COmpiler-compiler from Reference Grammar Input) which deals with the entire "front-end" component of a compiler; this includes the provision of necessary data structures and routines to manipulate them, both generated from a single input specification. Compared with earlier CWTs, CORGI has a higher-level and hence more convenient user interface, operating on a specification derived directly from a "reference manual" grammar for the source language.

Rather than developing a compiler-compiler from first principles, CORGI has been implemented by building a further shell around two existing compiler construction tools, namely lex and yacc. CORGI has been demonstrated to perform efficiently in realistic tests, both in terms of speed and the effectiveness of its user interface and error-recovery mechanisms.

Keywords: programming languages, compiler writing tools, lexical analysis, syntax analysis.

To my parents,

two people who, because of their inability to read and write, sacrificed so much to allow me to pursue my academic goals.

Acknowledgements

I would like to express my gratitude to the following people:

To my supervisor, Dr. E. F. Elsworth, for his patience, guidance and advice.

To Dr. Ian Johnson for his friendship and much needed moral support.

To all my contemporaries as research students in the department of Computer Science and Applied Mathematics of Aston University, for their companionship and stimulating conversation during the period of this research project.

To academic and non-academic members of staff in our department for their comments about this research work, both from seminars and informal discussions.

To members of the Language Studies Unit for their tuition in the English language.

To all members of staff in Computer Services for their help and friendship, in particular David Stops, Roy Parsons and Tony Bray.

To my brothers and sisters for their encouragement and understanding throughout my studies.

To Trevor Strudley for being a good friend to me and to my whole family and for being there whenever needed.

To Hyacinth Nwana, a research student in our department, for his moral support and for making the writing-up period more interesting.

And finally to the Algerian Government for the sponsorship of this research project.

List of Contents

Title page....1
Summary....2
Dedication....3
Acknowledgements....4
List of Contents....5
List of Figures....8

Chapter 1: Introduction....9

Chapter 2: Programming Languages - Definition and Implementation....13
2.1 Definition of programming languages....13
2.1.1 Syntax....17
2.1.2 Semantics....19
2.1.2.1 Operational method....20
2.1.2.2 Axiomatic method....21
2.1.2.3 Denotational method....22
2.1.2.4 Comparison of methods....23
2.2 Implementation of programming languages....23
2.2.1 Analysis of the source program....25
2.2.1.1 Lexical analysis....25
2.2.1.2 Syntax analysis....28
2.2.1.3 Semantic analysis....30
2.2.1.4 Symbol table....32
2.2.1.5 Error detection and correction....33
2.2.2 Synthesis of the object program....35
2.2.2.1 Code optimization....35
2.2.2.2 Code generation....37
2.2.2.3 Linking loader....37
2.3 Summary....38

Chapter 3: Translator Writing Systems....39
3.1 Brief history and overview....39
3.1.1 Lexical analyser generators....43

3.1.2 Syntax analyser generators 49
3.1.3 Semantic analyser generators 56
3.1.4 Code generator generators 58
3.2 Summary 63

Chapter 4: lex and yacc: A Detailed Investigation 65
4.1 Introduction 65
4.2 The lex lexical analyser generator 66
4.2.1 Input specification 66
4.2.2 Operation of the generated lexical analyser 69
4.3 The yacc syntax analyser generator 70
4.3.1 The input specification 71
4.3.2 Operation of the generated syntax analyser 76
4.4 The use of lex and yacc 77
4.4.1 MSL lex specification 78
4.4.2 MSL yacc specification 82
4.4.3 Augmented lex specification for MSL 85
4.4.4 Augmented yacc specification for MSL 87
4.5 Summary 91

Chapter 5: The Problem and the Proposed Solution 92
5.1 Rationale and intended goals 92
5.2 Development environment 93
5.3 Functional specification of the CORGI system 94
5.3.1 CORGI input specification 94
5.3.1.1 Annotation section 95
5.3.1.2 Rule section 103
5.3.1.3 User's routine section 110
5.4 Overall structure and use of the CORGI system 111
5.4.1 Deferred semantic reduction 112
5.4.2 Direct semantic reduction 114
5.5 Summary 116

Chapter 6: The Realisation of the CORGI System 118
6.1 Introduction 118
6.2 The front-end 119
6.3 The back-end 127
6.3.1 Generating the lex specification 127

6.3.1.1 The lex definition section 127
6.3.1.2 The lex rule section 130
6.3.1.3 The lex user routine section 136
6.3.2 Generating the yacc specification 137
6.3.2.1 The yacc definition section 137
6.3.2.2 The yacc rule section 138
6.3.2.3 The yacc user routine section 140
6.3.3 The abstract syntax tree declaration 141
6.3.3.1 The IDL system 142
6.3.3.2 The CORGI approach in generating the AST 148
6.3.4 Augmenting lex and yacc specifications 152
6.3.5 Error-recovery 155
6.3.6 The manipulation routines 158
6.4 Error handling in programs generated by lex and yacc 163
6.5 Summary 166

Chapter 7: Demonstration and Evaluation of CORGI 168
7.1 Introduction 168
7.2 Testing 169
7.3 Application of CORGI 169
7.3.1 Writing a CORGI specification for MSL 170
7.3.2 Overview of other PLs 174
7.3.3 Other limitations of the CORGI system 181
7.3.4 Testing error recovery 181
7.3.5 The performance of the CORGI system 185
7.3.6 Compiler construction using the CORGI system 187
7.4 Summary 193

Chapter 8: Conclusion 194

References 201

Appendix A 213
Appendix B 240
Appendix C 271
Appendix D 277
Appendix E 282
Appendix F 297

List of Figures

Figure 3.1 An idealized compiler-compiler model 41
Figure 3.2 Compiler-compiler model often used in practice 42
Figure 3.3 The PQCC compiler generator 60
Figure 3.4 The PQCC generated compiler 61
Figure 4.1 The lex lexical analyser generator 70
Figure 4.2 Cooperation of lex and yacc 71
Figure 4.3 The yacc syntax analyser generator 77
Figure 4.4 MSL syntax in EBNF 83
Figure 5.1 The syntax of the Annotation section in EBNF 103
Figure 5.2 The syntax of the rule section in EBNF 110
Figure 5.4 Deferred semantic reduction approach of the CORGI system 114
Figure 5.5 Direct semantic reduction approach of the CORGI system 115
Figure 5.3 The EBNF syntax of a CORGI specification 117
Figure 6.1 CORGI overview 118
Figure 6.2 IDL model of a process 142
Figure 7.1 CORGI performance 186
Figure 7.2 Time comparisons for three versions of the parser 187

Chapter 1

Introduction

The development of complex software systems is a lengthy and error-prone process but since it is often algorithmic in nature, researchers have examined the possibility of automating many of its aspects. From this research have emerged a number of "application generators", which accept a formalized description of a problem and produce a procedural implementation of a solution. This has given rise to the design of so-called fourth generation languages, which are largely application-oriented.

More specifically, tools have been developed to automate each of the phases of a "traditional" compiler: lexical analysis, syntax analysis, semantic analysis, code optimization and generation.

For the lexical analysis phase, systems based on the powerful and general-purpose regular expression notation, and also on less powerful, special-purpose notations have been developed. A number of syntax analyser generators, which use either LR or LL-based algorithms in the syntax analysers that they produce, have been investigated. Perhaps the most widely used tools to generate lexical analysers and syntax analysers, are lex (Lesk, 1975) and yacc (Johnson, 1975) respectively.

The automatic generation of routines to perform semantic analysis from a formal semantic description is a more complex and less-understood process than for lexical and syntax analysers. This is mainly due to the fundamentally more complex nature of programming language semantics.

The majority of systems which automatically generate semantic analysers use attribute grammar techniques to describe static semantics; also more recently systems have been developed based on the mathematical rigour of denotational semantics. In the less well-understood area of code generators, a variety of different approaches have been investigated which provide algorithms working in both a machine-dependent and machine-independent manner; this area has been studied by many researchers. The production of the code generation phase of a compiler has been the most difficult to automate. The goal of research directed towards this area of compiler construction is to be able to take a formalized description of the target machine architecture together with an intermediate representation of the program being compiled; from these, machine code for the target architecture should be emitted.

Existing tools have a number of apparent drawbacks. They often require the user to be conversant with their detailed internal operation, and may only deal with unduly limited aspects of compilation and their syntax is difficult and error-prone.

This thesis addresses the issue of overcoming the drawbacks of current tools by automatic construction of the compiler front-end directly from the "reference manual" grammar. The structure of the thesis is outlined below.

In chapter 2, we review the fundamental principles of programming language definition and implementation. We note the adoption of formal techniques applied to the area of syntax definition in a routine and universal manner. The general basis for compiler construction and organisation is described.

In chapter 3, we investigate the automation of software construction, for which compiler construction, due to the degree of formality readily available, is a

particularly suitable candidate. Various approaches taken by researchers to the automation of the process of compiler design and construction are also reviewed.

Chapter 4 gives a more detailed account of the use of lex and yacc in the construction of compiler front-ends. This chapter reveals how lex and yacc specifications can be constructed for a simple programming language, called MSL. In this way, we gained experience of the use of these two tools, and established a number of improvements which should be included in our system.

In chapter 5, we identify the need for a compiler-writing system which uses a "reference manual" grammar as its input. In this chapter we describe, in broad terms, the nature of such a system, its functional specification and overall structure.

In chapter 6, we shall describe in more detail the design and construction of our system as briefly introduced in chapter 5.

In chapter 7, we show how our new system (which we call "CORGI") can be used to generate a compiler front-end for typical modern programming languages such as Pascal and Modula-2 and a full compiler for a simple programming language, namely MSL. Results of a number of tests which we conduct using this system with grammars for typical modern programming languages are presented. A general conclusion is given in chapter 8.

Appendix A contains the hand-written version of lex and yacc specifications for MSL. It also contains the hand-written tree-walking routines which perform the semantic analysis and code generation required for MSL. Appendix B contains the system specification for MSL, together with the generated programs. Appendix C contains four MSL source programs together with the results produced by the hand-written version, the lex and yacc version and the CORGI-generated version of the

compiler. Appendix D contains a demonstration of the error-recovery mechanism incorporated in our system. Appendix E contains the system specifications for full Modula-2 together with the test programs used to run the generated front-end. The same test is done for Pascal. Finally appendix F contains the CORGI specification of the syntax of CORGI specification.

Chapter 2

Programming Languages - Definition and Implementation

2.1 Definition of programming languages

In this chapter, in order to provide a concrete basis for the subsequent work, we briefly review in outline the methods used in the Programming Language (PL) field. This is divided into two major sections, definition and implementation.

A complete definition of a programming language must include descriptions of its syntax (structure) and its semantics (meaning). Syntax defines the structure of legal sentences in the language, whereas semantics specifies the meaning of these sentences. Establishing the separation between syntax and semantics of PLs has been the subject of considerable debate. The main two contentions are that:

- Syntax covers only the context-free aspects.
- Semantics covers all other compile-time and all run-time aspects of PLs, known as static semantics (eg. type compatibility of operators and operands) and dynamic semantics (eg. procedure call mechanisms) respectively.

or that:

- Syntax deals with all context-free and context-sensitive aspects.
- Semantics only deals with run-time features of PLs.

The author supports the first view, since it is based on the traditional model of a compiler, which is well-understood, and there exists no overwhelming evidence to justify modifying this structure.

One useful and well known way of defining the syntax and the static semantics of a programming language is by means of a grammar, which is often referred to as a rewriting system. A grammar is a formal device for specifying in a finite way a potentially infinite set of sequences of characters grouped in a certain structure. A grammar consists of terminals, nonterminals, a start symbol, and a set of production rules (rules for short), informally defined as follows:

- Terminals are the basic symbols (tokens) of the language.
- Nonterminals are the syntactic variables that denote sets of strings. They also specify the hierarchical structure of the language in question.
- One special nonterminal, the start symbol, is identified as the basis for all derivations using the grammar.
- The set of production rules specifies the structure of the language.

Formally a grammar is defined to be a quadruple:

$$G = (Vt, Vn, P, S)$$

Where:

- Vt is an alphabet whose symbols are known as terminals.
- Vn is an alphabet whose symbols are known as nonterminals, with

$$Vt \cap Vn = \emptyset \text{ and } Vt \cup Vn = V$$

- P is a finite set of pairs called productions (or rules), such that each production (α,β) has the form

$$\alpha \rightarrow \beta \qquad \text{with } \alpha \in V^{+}, \beta \in V^{*}$$

- S is known as the sentence symbol (or axiom) and is the starting point in generating any sentence in the language.

A mathematical theory of grammars was developed by the linguist Noam Chomsky (1959), who classified grammars into four formal types. The classification depends on the form of the productions, and may be summarized as follows:

1- Type-0 grammar (Unrestricted): A type-0 grammar is one in which there are no restrictions on the form of all the productions. These productions have the following form:

$$\alpha \rightarrow \beta \qquad \alpha, \beta \in V^*$$

This type of grammar is much too powerful for PLs although it is still unable to cope with natural languages.

2- Type-1 grammar (Context-sensitive): A grammar is said to be of type-1 or context-sensitive if all of its productions have the form

$$\alpha \rightarrow \beta \text{ with } |\alpha| \leq |\beta| \qquad \alpha, \beta \in V^*$$

where $|\alpha|$ denotes the length of α, ie. the number of symbols in α, and similarly for $|\beta|$. This formal definition means that a grammar is context-sensitive if:

i) The number of symbols, terminals or nonterminals on the left side of every rule is less than or equal to the number of symbols on the right side of the rule.

ii) It contains at least one rule where the number of symbols on the left side of the rule is higher than one.

3- Type-2 grammar (Context-free): A further restriction leads us to the concept of a Context-Free Grammar (CFG). A grammar is said to be context-free if the left side of every production consists of a single nonterminal symbol, therefore the productions have the form

$$\alpha \rightarrow \beta \text{ with } \quad \alpha \in Vn \text{ and } \beta \in V^*$$

Notice that $\beta \in V^*$ may be found (Hopgood, 1969) as $\beta \in V^+$, but it does not matter since any type-2 grammar with empty rules can be rewritten as an equivalent grammar without empty rules.

At about the same time as Chomsky's work, the BNF (Backus-Naur Form) grammar model was developed by Backus and Naur for the syntax definition of Algol 60 (Backus, 1960). In fact Context-Free Grammars are equivalent in power to BNF grammars; the differences are basically notational. Hence the terms BNF and CFG are often used interchangeably. The following are properties of context-free grammars:

- The question as to whether a context-free grammar is ambiguous is unsolvable.
- The question as to whether a context-free grammar generates an inherently ambiguous language is unsolvable.
- The question as to whether two context-free grammars generate the same language is unsolvable.
- A single stack recognizer for context-free languages can be built.

The above statements are proved by Minsky (1972).

4- Type-3 grammar (Regular, right-linear or left-linear): Even further restrictions are imposed on productions which form a regular grammar. If each production of the grammar has one of the forms

$$A \rightarrow a \quad \text{with} \quad a \in Vt$$
$$A \rightarrow Ba \quad \quad A, B \in Vn$$

then the grammar is said to be left-linear or regular or a type-3 grammar. Similarly, a right-linear grammar is one where all productions are of the form

$$A \rightarrow a \quad \text{with} \quad a \in Vt$$
$$A \rightarrow aB \quad \quad A, B \in Vn$$

In most programming languages, most of the basic symbols such as *identifiers, integers, operators, reserved words, etc* can be defined using this type of grammar. It has been shown that type-3 grammars, that is regular grammars, are equivalent in power to regular expressions (REs) (Aho and Ullman, 1972). Moreover, efficient algorithms have been developed to construct finite automaton recognizers for tokens directly from REs (McNaughton and Yamada, 1960). The use of REs as a tool for building lexical analysers was originally exploited by Johnson *et al.* (1968), and Lesk (1975). The essential properties of regular expressions are:

- The question of whether two regular expressions generate the same regular set is solvable.
- The question of whether a regular expression is ambiguous is solvable.
- No regular language is ambiguous.
- There exists an algorithm to determine whether a string belongs to a given regular set defined by a RE.
- Regular languages can be recognized by a finite-state machine.
- Regular expression cannot describe nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on.

The above statements are proved by Minsky (1972).

2.1.1 Syntax

Since 1960 BNF has been widely used in defining formally the syntax of programming languages. A BNF grammar consists of a set of grammar rules (productions), which together define programming languages. In the simplest case a grammar rule may simply list the elements of a finite language (eg. the alphabet)

```
< letter > ::= A | B | C | .......| Z
```

This grammar rule describes a finite language composed of upper-case letters only.

Despite the power, elegance, and simplicity of BNF grammars, they are not an ideal notation for describing the syntax of programming languages. This stems from their inability to naturally express many of the constructs commonly found in programming language such as optional items, grouping of alternative items and repetitive items. Therefore, computer scientists have striven to extend the BNF notation to provide a more natural and consice method for defining language syntax. Several notations appear in journals and technical reports. Wirth (1977) in his short communication presents a simple notation that has proven valuable and was satisfactorily used to define the Pascal standard (BSI 1982).

In conclusion, a good syntactic language description has two primary benefits:

- It helps the programmer write a syntactically correct program.
- It can be used to determine whether a program is syntactically correct. In other words, it acts as a processor for the language. The compiler writer uses the grammar to write the syntax analyser which is able to recognize all valid programs. This process is now well understood and in fact, there are program generators that take the grammar of the language and generate the appropriate analysers from it.

Problems with CFGs

One of the problems which arise in describing the syntax of languages, natural or programming, using CFGs is ambiguity. Since parts of the meaning of a program are sometimes specified by the syntactic structure of PLs, ambiguity must be avoided in some appropriate manner. Typical examples of ambiguous grammars are:

- The "dangling-else" construct found in Algol-like languages.
- The expression construct, found in most PLs, described as:

```
expr = expr + expr.
```

A grammar is said to be ambiguous if it generates multiple syntax trees, which may lead to multiple interpretations, for any valid program. One important question is: Is there a general procedure for determining whether a given BNF grammar is ambiguous? Unfortunately the answer from theoretical studies is disappointing, since this has been shown to be an undecidable problem of grammars (Hopcroft and Ullman, 1969). In fact, some languages are inherently ambiguous ie. they cannot be generated by an unambiguous grammar. However, there might be an algorithm that could determine with certainty and in a finite time whether a particular grammar is ambiguous. In addition, certain ambiguities can be resolved by one of the following techniques:

- Rewriting the grammar to an equivalent unambiguous one generating the same language; eg. to solve the "dangling else". Alternatively, an explicit disambiguating rule can be appended to the grammar. Such a rule could have the effect of causing each else to be matched with the closest previous unmatched then (Aho *et al.*, 1986).
- Augmenting the grammar with additional information to resolve certain ambiguities; for example including a table showing the operator precedence and associativity (Johnson, 1975) and (Early, 1975).

2.1.2 Semantics

We have seen in previous sections that the definition of the syntax of a programming language is a well-understood process, for which formal methods exist, allowing easy automation of syntax analyser construction. In order to

completely specify all aspects of a language, however, we also need a mechanism for describing the meaning of syntactically-correct statements written in that language. Such a meaning is termed the semantics of a programming language. Semantic definition does not have a universally agreed method unlike syntactic definition, and in this section we will examine some of the approaches taken to this problem.

Often in the original specification of languages such as Algol60 and Fortran, the syntactic definition was augmented by paragraphs of informal prose, giving the semantics of each language construct. The main disadvantage of this approach is that the description is possibly ambiguous (due to the lack of mathematical rigour and formalism), and this may lead to different implementations by different compiler writers. It may also result in the language definition being incomplete or inconsistent.

Due to the inadequacy of an informal prose semantic description, the need has been identified for a definition mechanism which is precise and understandable, but which is also mathematically rigorous and formal. This mechanism should aid both the programmer to construct a correct program and the compiler writer to design a language implementation consistent with the definition. To this end, three main approaches have been taken; they are operational, axiomatic and denotational methods, which are described below.

2.1.2.1 Operational method

The operational method defines the semantics of a programming language by modelling the execution of program statements on a virtual computer. The virtual computer is represented in terms of a complex automaton, whose internal states correspond to the state of a program when it is executing. This state consists of the

program code itself, the value of variables and any housekeeping data required. Each statement of the program is then described as operations which modify the state of the automaton. Lucas and Walk (1969) describe the operational method called the Vienna Definition Language (VDL) (Bjorner & Jones, 1978) that was used to describe the semantics of PL/I.

There are numerous drawbacks to this approach. Since it uses an idealized model of the operation of a compiler or interpreter it is useful for the language implementor, but is difficult for the user who is more interested in an abstract description of program statements and not in implementation details. Moreover, when tracing the execution of a program, this method tends to present the result of statements rather than a clear idea of the program's execution. It is also difficult to prove an implementation of a program to be correct, since the virtual computer may only loosely resemble the real computer for which the compiler is being written; due to lack of mathematical rigour this method of describing the semantics is difficult to mechanize.

2.1.2.2 Axiomatic method

This method uses axioms or inference rules to define the effect of execution of program constructs, which are used in a manner similar to mathematical proofs to show the action of a whole program. It is based on mathematical logic and uses predicates which must remain true for program variables before and after performing a particular programming language operation. A typical problem facing the programmer is to write a program that transforms data satisfying certain properties 'D' into results satisfying properties 'R' which may be formalized thus:

```
{D} program {R}
```

If the predicate D (known as a precondition) is true before the program is executed, then the predicate R (known as a postcondition) must be true after the program is executed and has terminated. Thus the operation of program constructs is defined by transformations of predicates; predicates which describe simple statements (such as assignment) can be combined via control-flow constructs to build up more complex operations.

The axiomatic method is relatively easy for the user to understand, but provides no guidance for the language implementor. Since it has a sound mathematical foundation, it can be (and indeed has been) used to prove program correctness; such an idea was originally found in McCarthy & Painter (1967), and more recently in Thatcher *et al.* (1980). It has in fact shown the undesirability of such features as aliasing and the goto statement because of the enormous extra complication they introduce in correctness proofs. Hoare and Wirth (1973) have worked on this method, and it was used extensively for the first time to describe the semantics of a subset of the programming language Pascal.

2.1.2.3 Denotational method

In the denotational method, each language construct is defined in terms of a mathematical function, which maps values in the syntactic domain to those of the semantic domain. Again the program is described in terms of state transformations on three entities: memory, the input stream and the output stream. Denotational semantics (Scott & Stratchey, 1971) mix both static and dynamic semantics and represent them in the λ-notation, that is the λ-calculus (Hindley & Seldin, 1986) augmented with data types, working on the abstract syntax of the language. A denotational semantics definition of a subset of Ada has been presented (Honeywell, 1980).

The main advantage of this approach is that, as it is not tied to any particular implementation, it allows the language designer to reason more easily about the underlying features of program semantics. Also, since it has a firm mathematical base, it facilitates formal reasoning about programs, and lends itself to mechanization. In fact, considerable work is being carried out regarding direct execution of denotational semantics, which would lead to automatic construction of a whole language compiler.

2.1.2.4 Comparison of methods

None of the methods described above is capable of being applied to all aspects of language design and implementation. Axiomatic and denotational semantics are based on different mathematical foundations; the former is based on logic, in particular, predicate calculus; whereas the latter is based on functions, in particular, recursively defined functions. Denotational semantics are used to study the detailed interaction of language constructs independent of their implementation details, and axiomatic semantics provide a means for program correctness proving. On the other hand, operational semantics allow guide-lines to be laid down to the implementor. Unfortunately, these definitions are usually too detailed to be of much use to the user.

Although formal semantic definitions are becoming an accepted tool in defining some programming languages for example Ada (using denotational semantics) and PL/I (using operational semantics), the final definition is still generally given in prose. The final Ada manual is presented in English prose with BNF defining formally only its context free syntax.

2.2 Implementation of programming languages

> "Compilers and interpreters are a necessary part of any computer system -- without them, we would all be programming in assembly language or even machine language!"
>
> D. A. Gries (1971)

In order to profit from high-level languages (HLLs), a translator is needed. Simply stated, a translator is a program that translates a source language into equivalent code in some object language generally the machine language of some particular computer. Translators may be classified into three major classes, depending on the source language and the object language dealt with. These three classes, namely "compiler" (the term that was coined in the early 1950's by Grace Murray Hopper), "assembler" and "interpreter", are defined as follows:

Compiler

A compiler is a program that translates a program written in a high-level language such as Fortran into an equivalent machine language of some computer or into assembly language.

HLLs —————————> machine or assembly language

Assembler

If the source language is an assembly language, and if the object language is the machine code of a particular computer, either in the form of a relocatable or a numeric code, then the translator is called an assembler. Assembly language closely resembles machine language, and is often just a mnemonic version of a computer's instruction set.

Interpreter

An interpreter on the other hand, is a program that accepts a source language as input and executes it. An interpreter, in contrast to a compiler, does not build an object language representation; it may typically perform two functions:

- It translates a source program written in the source language into an intermediate form; this part is similar to the analysis part of a compiler.
- It then executes the program in the intermediate form.

A compiler must perform two major tasks: an analysis of the input source program and the synthesis of the executable object program. Neither analysis nor synthesis is yet a simple enough task to describe as a single entity. Each of these tasks needs further subdivision, into what are usually called phases, as we shall see below.

2.2.1 Analysis of the source program

The analysis part of the translation (compilation) process breaks up the source program into basic components, then builds an intermediate representation of the source program. One good and widely used method of representing the syntactic structure of the source program is a linked data structure, known as a syntax tree or a parse tree. The analysis part may be divided into the following phases:

2.2.1.1 Lexical analysis

The lexical analyser is the simplest part of the compiler, yet is typically the most time-consuming task in the compilation process as a whole. It is the section that communicates with the outside world, through the operating system. In discussing lexical analysis, it is necessary to introduce the terms "lexeme", "pattern", and "token", defined as follows:

- token: a terminal symbol used in the grammar describing the language
- lexeme: a sequence of characters from the source text grouped together in a particular structure.
- pattern: a formal method for defining a lexeme; in practice a pattern is in fact a regular expression.

The first task of any programming language processor must be to read in the input text, character-by-character, grouping them into elementary constituents termed lexemes. The lexemes are placed in categories (which will be the tokens of the relevant grammar) such as identifiers, operator symbols, numbers, comments, keywords, etc, which are then passed to the next phase of the compiler known as the parser (discussed later in the chapter). Some lexemes are further processed before being returned to the parser. Lexemes such as numbers are often converted to an internal representation (to internal binary-fixed or floating-point form), and identifiers may be stored in a symbol table. For these lexemes, both their token number and their value are passed for use by the semantic routines. A typical lexical analyser performs the following sub-tasks:

- Partition the program into lexemes.
- Eliminate unnecessary information such as blanks and comments.
- Establish the nature of symbols - eg. whether they are keywords or identifiers.
- Enter some preliminary information into a symbol table.
- Output a program listing and error messages.

Although lexical analysis is simple in concept, over 50% of compilation time is often spent in this phase (Waite & Carter, 1985). One reason for this is simply because there is so much character-level processing; the other reason is that sometimes in practice it is difficult to find the boundaries between lexical items.

Feldman (1979) discusses the potential difficulty of token recognition in Fortran77. A popular example that illustrates this fact is :

```
DO  50  I = 1.5        and
DO  50  I = 1,5
```

The first statement is an assignment, whereas the second statement is a DO statement. This fact, however, cannot be discovered by the lexical analyser until reading the decimal point character or the comma. This difficulty arises for two reasons: keywords are not reserved, and blanks are not significant, hence token termination depends on statement recognition. In this case, a rather complex context-dependent analysis algorithm is needed. This algorithm must look far ahead beyond the end of a lexeme before a token can be determined with certainty.

Specification of tokens

A precise and formal definition of tokens may seem unnecessary, given the simple structure found in most programming languages. However, the structure of tokens can be more tedious than we might expect. For example here is an informal definition for a Pascal string.

```
" A string can be any sequence of characters and is
  delimited by a single quote character."
```

The problems with this definition are:

- Is a null string allowed?
- Can a linefeed character appear in a string?
- And what happens if there is a single quote character in the string?

Hence a precise definition of tokens, such as by using regular grammars, is necessary.

2.2.1.2 Syntax analysis

Simply stated, the role of the parser is to examine the sequence of items obtained from the lexical analyser and discover how these items are grouped together to form 'phrase' or 'sentence' fragments. The parser also reports syntax errors. Once the syntactic structure is recognized, the parser either builds an intermediate representation known as a syntax tree, which is used to drive semantic processing after the tree is completely constructed, or calls corresponding semantic routines directly. The latter technique is needed in order to perform single-pass translation, which is important for compile-time efficiency; however, not all languages allow single-pass compilation. In fact, it may be cleaner and clearer to perform distinct phases of compilation in separate passes.

A great deal of research has been carried out into the design of efficient syntactic analysis techniques, in particular, techniques that are based on the use of formal grammars, notably context-free grammars. Some researchers, namely Kasami (1965), Younger (1967) and Earley (1970) developed algorithms for parsing any context-free grammar, which was too ambitious and resulted in inefficient parsers. These methods are termed universal parsing techniques.

There are two common techniques to parsing; top-down and bottom-up techniques. A parser is considered top-down if it "builds" the parse tree starting from the root (top) and terminates at the leaves (bottom). However, with a bottom-up technique, the parser discovers the structure of a parse tree by beginning at its leaves and continues until it reaches the root of the tree. One feature these two techniques share is the fact that they both scan the input from left to right, one lookahead symbol at a time.

The best and commonly used top-down and bottom-up parsing techniques are known as LL(k) and LR(k) respectively. The first letter L states how the input is

read that is Left-to right; and the second letter L or R states the kind of parser produced, that is Leftmost or Rightmost parsing. k is the number of lookahead symbols; in practice it is 1. Both LL and LR parsers work on sub-classes of CFGs, namely LL and LR grammars respectively. LL, also known as predictive, parsers were thoroughly investigated by Knuth (1971) and used in compilers by Lewis *et al.* (1976). On the other hand, LR parsers and grammars were first developed by Knuth (1965).

A great deal of research has been conducted into the theory of LR grammars, the largest natural class of CFGs that can be parsed with a deterministic pushdown automaton. Further work has concentrated on the improvement of Knuth's LR parser, which is considered to be impractical for real systems, and the result has been the development of the so-called SLR, Simple LR, (DeRemer, 1971) and LALR, LookAhead LR, (DeRemer, 1969) methods. SLR and LALR parsers provide significant improvement in terms of the time and space required to construct a parser from a grammar. This topic is discussed at more length by Aho & Johnson (1974).

Both LL and LR-like grammars have the same advantage in that they require no backtracking. Parsers which are linear in both space and time (Aho *et al.*, 1986) can be produced and errors can be detected at the earliest possible opportunity. However, LR-like grammars cover a wider class than LL-grammars which also tend to be less natural.

In addition to the context-free grammar properties given in section 2.1, LL and LR grammars have the following further properties:

- The question as to whether a grammar is LL(k) or/and LR(k) is solvable.
- The question as to whether a language is LL(k) or/and LR(k) is unsolvable.

- Each LL(k) grammar is also LR(k).
- There are LR(1) grammars that are not LL(k) for any k.
- LR(k) grammars are not ambiguous.

2.2.1.3 Semantic analysis

Semantic analysis is one of the most complex phases of a compiler. It plays a communication role between the analysis and synthesis parts of the compilation process. During this phase, the correct syntactic structure (syntax tree) recognized by the parser is further processed to generate some internal form, a preliminary version of the final executable program.

The main purpose of this phase is to determine and check the static semantics of the source code read by the compiler. The static semantics (discussed earlier in the chapter) are those restrictions concerned with the scope rules (eg. visibility and accessibility of program objects) and type rules (eg. type compatibility of program objects) of the language in question. The semantic analyser generally also performs other functions, which vary depending on the logical structure of the compiler, and the language involved. However, symbol-table maintenance is one of the most common functions in the translation process of most programming languages.

Much effort has been directed towards the development of techniques, based on methods with formal mathematical foundation, for defining the static semantics of programming languages. Several different approaches namely operational, axiomatic, and denotational have been introduced. These approaches have been discussed earlier in this chapter.

An alternative approach is based on the concept of grammars. The revised report on Algol-68 (VanWijngaarden *et al.*, 1976) used a W-grammar, also called a two-level grammar, to describe the syntax and the static semantics of the language. This type

of grammar is a context-free grammar augmented with a second grammar allowing the syntactic treatment of context-sensitivity. Indeed this grammar is found to be equivalent to a type-0 (unrestricted) grammar which is a rather powerful concept, but large and complex.

The issue of whether Van Wijngaarden grammars decrease or increase the clarity of the context-sensitive syntax of a language is a debatable question. However, what is certain is that there has not been any kind of general parsing technique that can use these powerful grammars. In fact, it is extremely difficult to visualize a recognizer for general two-level grammars due to the problem of finding out what rules are applicable after a certain rule; it is known to be an unsolvable problem.

Another approach based on extending the context-free grammar is the concept of attribute grammars (AGs). Attribute grammars were initially introduced by Knuth (1968) as a means for defining the static semantics of programming languages. AGs are regarded as a suitable tool for use in writing compilers because of their capability in formally specifying often costly translations; however they may be very complex and expensive to evaluate, for example, and the evaluation process may not even terminate in all cases. The literature contains several examples of how various features of PLs can be described using AGs. A bibliography on (the use of) attribute grammars was published by Räihä (1980).

Informally stated, an attribute grammar is an ordinary context-free grammar enhanced by means of attributes and attribute evaluation rules (semantic rules). Each grammar symbol (terminal or nonterminal) has an associated set of attributes (inherited and synthesized), and each production rule is provided with the appropriate semantic rules setting up dependencies between the attributes of symbols. These dependencies are often represented by a graph from which an evaluation order for the semantic rules can be derived. Using AGs, the semantics

are given in a declarative rather than algorithmic (procedural) specification which is independent of any parsing method. Moreover, AGs have the advantage of being a context-free grammar based tool in that all the parsing techniques used with CFGs can also be used with the AGs; yet they still have not reached the popularity of the former, due to the difficulty of obtaining implementations efficient enough for practical and general use. At about the same time as Knuth's work on attribute grammars, Koster (1971a) developed so-called affix grammars, which are a derivative of Van Wijngaarden grammars; they have the advantage of being designed to be parsed and are hence generative, whereas attribute grammars were designed purely for translation purposes.

One of the major problems encountered in the use of attribute grammars is that of circularity. An attribute grammar is said to be circular if the dependency graph for some parse tree has a cycle. Cyclical dependencies cause evaluation to fail during compilation, therefore it is important to be able to decide whether an AG is circular or not; Knuth (1968) and Bruno & Burkhard (1970) presented an algorithm to solve this problem. However it is very hard to test an attribute grammar for circularity. Jazayeri *et al.* (1975) have shown that for any algorithm there is an infinite number of grammars, for which the circularity test is of exponential time complexity. Nevertheless, there are several sufficient conditions that can be checked in polynomial time (Bochmann, 1976; Jazayeri & Walter, 1975; Kennedy & Warren, 1976).

2.2.1.4 Symbol table

During the compilation process, it is necessary to record the use of identifiers in a program. For this purpose the compiler uses a data structure called a symbol table. In this table it stores the character string used to form the identifier together with further information such as its type (ie. real, integer, character etc.), its class (ie. is

it a simple or structured variable, etc.), its location in memory, and other specific attributes. Often, identifiers are stored in separate tables depending on whether they are labels, procedure names, variable names and so forth.

Information is entered into the symbol table during the lexical and semantic analysis phases. Whenever an identifier is encountered in the program text, the symbol table is searched, and if the identifier is not already there, a new entry is formed. The information can then be used during the semantic analysis to check that the use of an identifier is correct in its context, and also during code generation, to allocate the amount of storage required at run-time.

There are three common methods of storing symbol tables, namely: linear lists (ordered or non-ordered), binary trees and hash tables. This topic is fully elaborated in Aho *et al.* (1986).

2.2.1.5 Error detection and correction

A compiler must be able to detect errors in a program and should produce meaningful error messages to allow corrections to be made by the programmer. It must also be able to continue parsing the program in the face of such errors, since it is not acceptable to simply abort the compilation process as soon as an error is found. It is useful to group errors according to the compilation phase in which they are detected; hence there are lexical, syntactic and semantic errors.

A lexical error is found when the input character stream cannot be matched with a valid token of the language. In order to recover and continue scanning, the lexical analyser could simply discard characters from the remaining input until a valid token is found. This strategy, known as "panic mode", may sometimes confuse the parser. Another error-recovery strategy the lexical analyser may use is error

transformation eg. inserting a missing character, replacing an invalid character by a valid one, deleting an extraneous character or swapping two adjacent characters, etc.

Errors detected by the syntax analyser are violations of the context-free specification of the programming language. LL and LR parsers detect such violations when they find an error entry in the parsing action table. This means that the location at which the error is detected is as close as possible to where it actually occurred in the program text. A simple recovery strategy is to enter "panic mode" and repeatedly discard input symbols until a synchronizing token (like ";" or "end") is found, and then start parsing a new nonterminal. A more intelligent but similar error-recovery scheme known as "error production" is employed by Wirth (1968) for handling errors in a PL360 compiler and is also used by the parser generator yacc. The compiler writer can incorporate a production of the following form:

```
statement --> error ';'
```

which indicates that if a parsing error is found, the parser should scan until it finds a semicolon which is the synchronizing token. It then discards symbols from the parse stack replacing them by the nonterminal `statement` and finally continues parsing.

A method that was first used in top-down parsing is known as "local recovery" or "phrase-level recovery" proposed by Leinius (1970). The parser may insert, delete or replace the prefix of the remaining input to allow the parser to continue. A similar concept known as "global recovery" can be used, whereby the correction is performed upon the invalid string itself, however due to the costs incurred both in terms of space and time, this technique is only of theoretical relevance; this technique can also be used by the lexical analysis at a character level.

During semantic analysis, typical errors found are undeclared variables and type inconsistencies. In the case of an undeclared variable the usual error-recovery mechanism is to create a new entry in the symbol table with attributes assigned depending on the context in which the variable was found. This variable is flagged as being declared due to a semantic error, and subsequent error messages are only produced when a "new" error involving this variable is detected.

2.2.2 Synthesis of the object program

The main purpose of this part of translation deals with the construction of the executable program from the output (intermediate code, eg. Polish notation) generated from the semantic analyser. The major phase of this section is code generation which may be preceded by an optional phase known as code optimization. If subprograms are compiled separately then the code produced is passed to a further phase called the linker and loader, which is strictly part of the operating system software, but of sufficient importance to compiler construction to be of relevance to our work .

2.2.2.1 Code optimization

The translation of a source code program into an intermediate representation is mainly concerned with the structure of the program considered in isolated fragments or blocks, corresponding to language constructs. If code is generated directly from such an intermediate representation it is likely to be inefficient. The main reason for this inefficiency is that no account is taken of possible improvements which can be found by looking at the control and data flow within the program. The phase of the compiler that attempts to find such improvements is known as code optimization. The optimizer's task is to create a program whose effect is identical to the original, but which is more efficient.

Much optimization can be performed at the level of a basic block of the PL. A basic block as defined by Tremblay & Sorenson (1985) "is a program fragment that has only one entry point and whose transfer mechanism between statements is that of proceeding to the next statement" .

One of the simplest and most effective optimizations is that of constant folding. This involves evaluating at compile time, expressions which just involve constants, and this concept can be extended to variables whose value can be identified as remaining constant throughout a whole block.

The method of deferred storage is used to optimize store accesses. Instead of always accessing variables in main memory, it is possible to hold their values in fast registers. By marking this fact in the symbol table, subsequent access can be made to those registers rather than to main memory, only resorting to storing the variable's value at the end of a block. Linked with deferred storage is global register allocation, which establishes points in the code at which register values should be dumped to main memory, thus freeing them for later use in the program. If several statements all reference a common sub-expression, then much redundant code can be eliminated, by evaluating this sub-expression once and recording the other references to it. This is particularly effective when used in conjunction with global register allocation.

After the above techniques have been applied, further improvement can be found by analysing the code over small "windows". This is so-called "peephole" optimization (McKeeman, 1965). Typical peephole optimizations are to remove unreachable code (for instance after jump instructions), simplifying algebraic expressions (eg. changing a+0 to a), strength reduction (eg. replacing a^2 by a*a). In fact, peephole optimization may be used with either object code, after the target machine code is produced, or with the intermediate code produced by the analysis

part. Further useful peephole optimizations include redundant load-store operations, dead-code elimination, jump-chain compression and application of idioms specific to the target architecture (eg. best usage of addressing modes, auto-increment registers).

2.2.2.2 Code generation

The code generation phase of a compiler takes the intermediate representation of the program, after it has been optimized as described above, and translates this into an object language. Depending on the type of translator this could be machine language, assembly language, or another high-level programming language (eg. a preprocessor for structured Fortran). In the case of machine language, this may be absolute, meaning that it can be loaded and executed immediately, or relocatable which allows program modules to be compiled separately and combined by a linking loader .

2.2.2.3 Linking loader

The role of the loader is to take the relocatable code which has been generated and to form it into a single executable code sequence. During code generation, the compiler will not have been able to use absolute addressing for objects referenced which are defined in another separately compiled module. Instead it will have inserted requests for the loader to insert these addresses when the different sections are combined. Previous phases of the compiler will have stored a table of addresses as part of the relocatable machine code to be used for this purpose.

When each module is compiled it is assumed to start at some fixed address in memory, and so a further task of the loader is to relocate addresses so that they are correct when modules are joined into one executable program. It is usual during

this process to separate data and program segments, since the former can be written and read, but the latter can usually only be read.

The loader's final task is to set up the program's run-time environment by initializing the stack, allocating heap space, reserving input/output buffers and preparing for run-time debugging.

2.3 Summary

In this chapter we have reviewed the fundamental principles of programming language definition and implementation. We have noted the adoption of formal techniques applied to the area of syntax definition in a routine and universal manner. Approaches used in semantic definition are less widely accepted, and we have presented the most common directions of research towards the formalization of static semantics.

We have also described methods for language implementation, through compiler construction, for which a model structure was presented.

In the next chapter we will begin to investigate the automation of software construction, for which compiler construction, due to the degree of formality readily available, is a particularly suitable candidate.

Chapter 3

Translator Writing Systems

3.1 Brief history and overview

As the development of complex software systems has become increasingly costly, a need for tools has arisen which aid software construction. A fertile field of research has been that of so-called compiler writing systems, which facilitate the generation of programming language translators. Automatic compiler production has grown from this research and a number of projects have been undertaken with various degrees of success.

A common use of the above technique is that of translator generators, which are given a formal description of an input language, and produce a translator from that language to a specified object form. Translator generators have been employed to construct:

- Compilers and interpreters for programming languages (Johnson, 1979 and Feldman, 1979).
- Command line interpreters.
- Editors (both normal and language-based) (Reps *et al.*, 1983; Reps & Teitelbaum, 1987)
- Query languages for data base systems.
- Spreadsheets.
- Text formaters and mathematical typesetters (eg. troff, nroff, EQN, PIC using lex and yacc).
- File processors (eg. awk).

The production of compilers has seen wide use of translator generators. The first compiler for Fortran, which was hand-coded, required 18 man-years to develop, and its successor took a further 10 man-years. There is considerable motivation for automating this process since hand-coding a compiler for a large high-level programming language, despite advances in compiler techniques, still typically takes at least 4 man-years. Work on such automation began with the compiler-compiler of Brooker and Morris (1962), Brooker (1963) and Rosen (1964). When the syntax of Algol-60 was specified using a context-free formalism, generation of syntax analyzers was made possible. Irons (1963) extended this further to incorporate semantic analysis.

Compiler-compilers are the main subject of this thesis. They range from systems which assist the development of particular compilation phases (Branquart *et al.*, 1977; Leverett *et al.*, 1980; Koskimies *et al.*, 1982; Ganzinger *et al.*, 1982; Reiss, 1987; Yed, 1988), to systems which tried to produce a complete compiler (DeRemer, 1975; Kastens, 1980; Asbrock *et al.*, 1981). Meijer & Nijholt (1982) give a large bibliography on compiler-compilers and more recent ones are found in (Hennessy & Ganapathi, 1986).

An idealized model of a compiler-compiler as shown in Figure 3.1 uses descriptions of a programming language and a target language and produces a compiler that translates the source language into the target language. Early attempts to construct such a system, for example CDL developed by Koster (1971b, 1974a) using a two-level grammar (Koster, 1974b), did not succeed, since they required an algorithmic description to be provided by the compiler-writer. This does not approach the declarative style of specification which is required to ease the task of automatic compiler construction. As DeRemer (1975) notes:

" CDL falls short of the ideal because part of the language description as written in CDL is a set of program fragments that describe what the translator being constructed is to do, rather than what the language being described is to be."

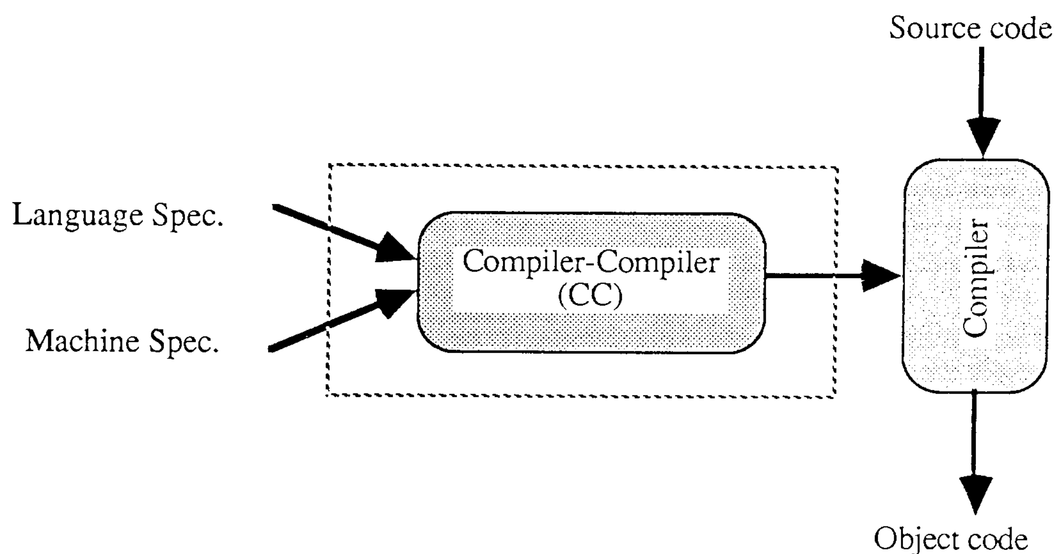


Figure 3.1 An idealized compiler-compiler model

The failure of systems such as CDL to satisfy the necessary requirements has led to the partition of compiler specification and generation along the lines of the phases normally recognized in compiler implementation. Thus lexical, syntactic and semantic analysis each have their own descriptive languages from which tables are built to drive general-purpose algorithms appropriate to each phase of the compilation process. An example of this approach was proposed by DeRemer (1975). In his system a sequence of BNF-like grammars are used to denote the various levels of a language. The operation of such system is shown diagrammatically in Figure 3.2.

Subsequent developments, for example those employing attribute grammars or denotational semantics, have attempted to use the properties of language semantics

to produce completely automated construction of the compiler front-end. Advances have also been made in the automatic mapping of these language semantics into machine semantics; thus allowing specification of the entire compilation process and so enabling the construction of code generator generators.

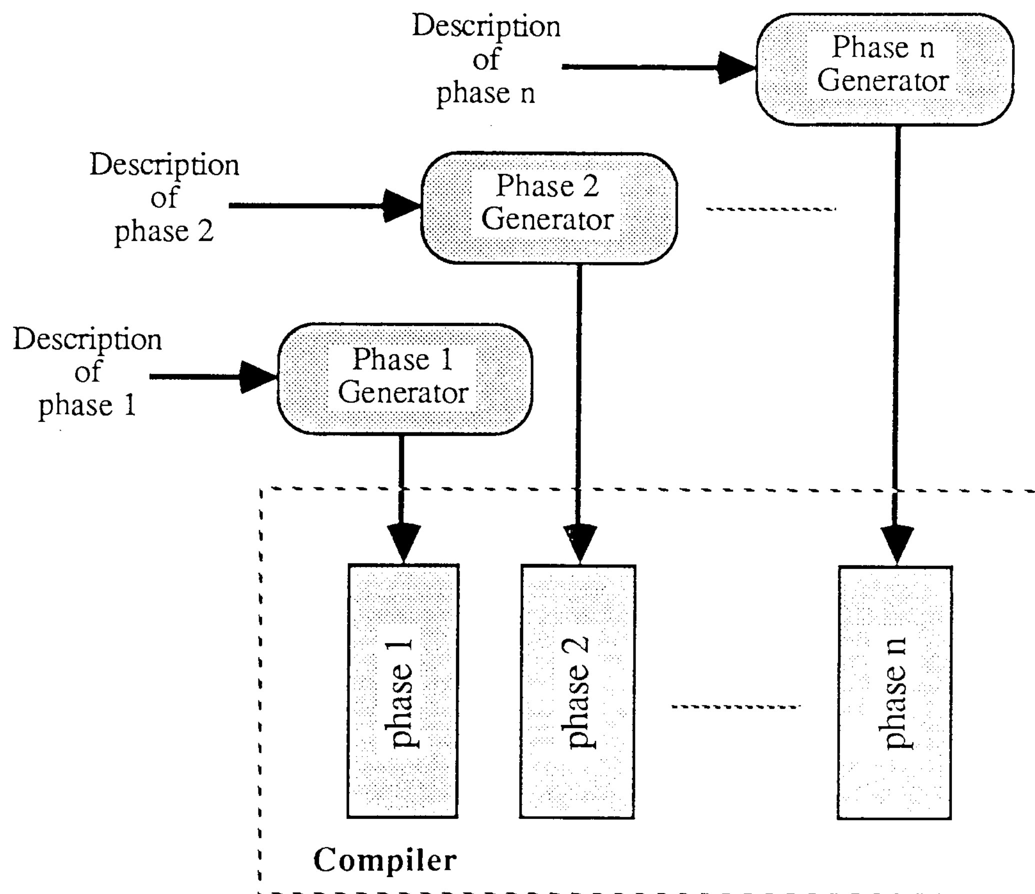


Figure 3.2 Compiler-compiler model often used in practice

In the rest of this chapter, we will examine in more detail previous work on compiler-compiler development, based on the phases of the compilation process on which a particular system concentrates. We have chosen to group this work as follows:

- Lexical Analyser Generators.

- Syntax Analyser Generators.
- Semantic Analyser Generators.
- Code Optimizer and Code Generator Generators.

Lex and yacc, the two most widely used tools, are discussed at greater length in chapter 4.

3.1.1 Lexical analyser generators

The process of constructing an analyser which is implemented using a finite state automaton (FSA) to recognize the fundamental lexemes of a given programming language is both time-consuming and tedious. Since the use of a FSA is well-understood and algorithmic in nature, it is natural to wish to automate this process. Thus we need to provide a mechanism via which a lexical analyser can be generated, given a formal description of the structure of a language's lexical items. However, since up to 50% of compilation time is spent in lexical analysis (Waite & Carter, 1985), the generated lexical analyser must not be much less efficient than a carefully constructed hand-written version.

The lexical analyser must be given a number of patterns, specifying how characters taken from the input stream should be grouped into valid lexemes. To facilitate this specification, many lexical analyser generators provide a means of naming character classes (eg. `letter = [A-Za-z], digit = [0-9]`).

There are generally two approaches taken in the operation of a lexical analyser generator. One is to produce a set of tables, where entries specify state transitions which occur when particular characters are encountered in the program text being scanned; thus each entry is given as a character/transition pair. Such tables are then used to drive a general-purpose scanning algorithm coded once and for all. An

alternative approach is to output executable code from the lexical specification to implement a lexical analyser for a particular language.

An executable-code lexical analyser has the advantage of being faster and does not require significant amounts of storage for state transitions; however it is by definition produced in a given language and therefore has limited portability. A table-driven lexical analyser on the other hand, can be easily ported to a new system by simply recoding the general scanning algorithm; its major disadvantages are that it can run up to five times slower than an executable-code equivalent (Waite, 1984, 1986), and it requires storage for its tables, which may prove sizeable.

The notation used to specify lexical items can be categorized as either general- or special-purpose. General-purpose notations are based on regular expressions (REs) which have been widely studied, and shown to be a powerful tool for lexical specification. Such a notation can be used in pattern-matching applications other than compiler front-ends, and are sufficiently expressive to allow lexical analysis of special-purpose programming languages which have an "unconventional" lexical structure. However, they do tend to lead to increased execution time due to this generality.

Special-purpose notations are less expressive, but it has been argued that they cover most kinds of lexical items used in modern programming languages (Heuring, 1986) and (Horspool & Levy, 1987). However, systems based on such notations make certain assumptions, which cause some features of PLs to be difficult or impossible to specify; for example, it may be assumed that integers consist of purely numeric characters, but the 'C' programming language requires hexadecimal constants to contain an 'x' as their second character; also the use of string delimiters can prove tedious using such a notation.

We shall examine a number of lexical analyser generators, grouped depending on whether they use a general- or special-purpose notation.

General-purpose lexical analyser generators

The first lexical analyser generator to use the theory of finite automata to build efficient lexical analyzers was AED-RWORD, developed by Johnson *et al.* (1968). This system uses a notation based on regular expressions, and the structure of a lexical specification is as follows:

```
BEGIN  { < character class definition > } END
BEGIN  { < symbol description > } END
FINI
```

A typical example would be:

```
BEGIN  space  = / /
       letter = /ABCDEFIJKLMNOPQRSTUVWXYZ/
       digit  = /0123456789/
END
BEGIN identifier (1, Lookup) = letter { letter | digit } $
       int (2) = digit { digit } $
       plus(3) = + $
       comments(4, comment) = /* $
       Ignore = space $
END    FINI
```

From this specification, RWORD produces a number of tables together with a series of routines to be called during scanning; also user-specified routines (eg. Lookup and comment) may be included, these are called whenever a complete symbol has been recognized. The construction of this table-driven code is performed in two stages: first, a version of the finite state automaton is output in the AED-0 language, then a second phase produces a macro-assembly equivalent.

Although RWORD represented a significant advance in lexical analyser generator techniques, it did suffer from some drawbacks: the use of large tables and numerous subroutine calls when a symbol is being recognized is both space- and time-consuming, and since the emitted code is in the AED language this severely limits the portability of the system.

The Alex lexical analyser generator (Mössenböck, 1986) was developed as part of the Coco compiler-compiler system (quoted in Mössenböck, 1986) and produces code for the lexical analyser written in Modula-2; its input notation is based on EBNF. However this powerful notation is used in such a restricted way that it can be argued that in fact the system uses a regular grammar specification. This is exemplified by the fact that only character classes and literals (but not nonterminals) may appear on the right-hand side of the lexical productions. Also, the authors themselves say that the notation is unable to deal with nested comments, thus necessitating a separate section for comment specification. Each lexical production gives a number as its left-hand side, being the value returned by the lexical analyser on recognizing a token. Ambiguities are resolved by taking the longest match, but if this is inappropriate it may be overridden by the user by specifying exceptions to this rule using the `IF FOLLOWED BY` feature. It is claimed that this feature solves the Fortran DO-loop problem, described in section 2.2.1.1.

The following is an example specification describing a subset of Modula-2:

```
CHARACTER SETS
   letter    = "ABCDEFGHIJKLMNOPQRTSUVWXYZ";
   ocdigit   = "01234567";
   digit     = octdigit + "89";
   hexdigit = digit + "ABCDEF";
   ...
KEYWORDS
   1 = "AND"
```

```
   2 = "DIV"
   3 = "IF"
   ...
   40 = "MODULE"
TOKEN CLASSES
   41 = -- identifier
        letter { letter | digit } EXCEPT KEYWORDS.
   42 = -- integer or cardinal
          digit { digit }
        | octdigit { octdigit } "B"
        | digit { hexdigit } "H"
        | digit { digit } IF FOLLOWED BY ( "." "." ).
   ...
SINGLE TOKENS
   0 = endfile.
   1 = "&".
   46 = "*".
   ...
   70 = endline.
COMMENTS FROM "(" "*" TO "*" ")" NESTED.
```

Perhaps the most widely known lexical analyser generator is lex (Lesk, 1975), which was developed for Unix-based systems; it was used to implement interpreters for many of the common Unix utilities (eg. awk, PIC, EQN, etc.). Its input notation allows all features of standard REs. A user-supplied action, written as a C fragment, is performed whenever the corresponding REs are matched in the input stream. Lex produces a set of tables which drive a general scanning algorithm, and are constructed in such a manner as to directly implement a deterministic finite state automaton from the regular expressions. This means that the resulting analyser is quite fast even for large sets of regular expressions. A recent improvement to lex, namely flex (Jacobson, 1987), also incorporates an automatically generated trace facility, allowing detailed operation of the lexical analyser to be monitored during execution.

A conceptually similar lexical analyser generator, called ScanGen, was developed by Gray Sevitsky, enhanced by Robert Gray, and further changed by Fischer & LeBlanc (1988). However, ScanGen purely produces tables, requiring the user to write his own driver; also, unlike Lex, user-specified actions are not allowed to be included, thus precluding any further processing of tokens.

The LAWS (Lexical Analyzer Writing System) (Gammill, 1983) provides the user with a means of generating state transition tables and finite state automata, written in a language designed by Schwanke (1972) for this purpose, namely STATE-DEF. The STATE-DEF compiler, written in Fortran, produces tables to map characters encountered into character classes; the finite state automaton interpreter (also written in Fortran) returns action codes to user-written lexical analyser routines.

Special-purpose lexical analyser generators

The GLA system (Generator for Lexical Analysers) (Heuring, 1986; Waite *et al.*, 1986) was designed to run in a Unix environment, and can, for example, replace the use of lex when generating a syntax analyser using yacc. It produces a directly executable lexical analyser in Pascal and C. In order to increase execution speed of the generated lexical analyser, it was decided to restrict the allowable symbol sets to those commonly used in programming languages, ie. identifiers, numbers, strings and comments. Identifiers and numbers are specified in terms of the set of characters with which they may start, together with the set of all possible following characters; strings are defined in terms of their delimiting characters, as are comments. Thus a complete specification describing the language to be recognized is partitioned into two sets: the basic symbol specification which defines the set of 'tokens' of the language, treating identifiers and denotations (literals) as 'generic' ie., it describes them only through their terminal symbol codes. The second set of specifications consists of definitions for the restricted symbol sets as described

above. The following is an example specification describing a very small subset of ISO Pascal for a GLA system:

Basic symbols specification	Token definition specification
'+' '-' '<' '=' ';' ... 'IF' 'THEN' ... Pascal-Idr. Pascal-Int. Pascal-Str.	IDENTIFIER Pascal-Idr [a-zA-Z][a-zA-Z0-9] INTEGER Pascal-Int [0-9] REAL Pascal-Real [eE] STRING Pascal-Str ['] DOUBLED BEGINCOMMENT 1 { (* ENDCOMMENT 1 } *)

The Mkscan (Horspool & Levy, 1987) system takes a similar approach to GLA in that it does not attempt to provide a tool for general pattern matching, but restricts symbol sets to identifiers, keywords, numbers and special symbols. Its major motivation was to enhance the user-interface to lexical analyser generator tools, by using a screen-based, menu-driven approach to lexical specification; thus the generation and subsequent editing of lexical analysers during their lifetime are facilitated. However both Mkscan and GLA (and its successors) do not seem to provide a mechanism for handling certain special kinds of tokens, mentioned earlier, including space characters. Similarly to GLA, Mkscan produces code rather than tables to form the generated lexical analyser.

3.1.2 Syntax analyser generators

Syntax analysis, using a context-free grammar description of a programming language, is a well-understood task for which there is a sound theoretical base. Since a paper by Knuth (1965) laid the foundations of automating the production of syntax analysers, a large number of syntax analyser generators have been developed; these are found both as part of whole compiler-compilers and as systems in their own right. By far the majority of modern generators produce syntax analysers which use LL or LR techniques, with the latter being the most popular, since they cover a wider class of grammars and provide superior facilities for error detection. The most notable LR-syntax analyser generator is yacc, designed originally for the Unix system.

In the following discussion we choose to group syntax analyser-generators based on whether they produce LL or LR syntax analysers.

LR-syntax analyser generators

There exist a large number of LR-syntax analyser generators, so we have restricted our discussion to those systems which are representative of the techniques used.

The EAGLE syntax analyser generator (Franzen *et al.*, 1977), developed at the University of Berlin, was designed with the objectives of using a specification language to concisely describe the syntax and the static semantics of a programming language, and to generate practical compilers. The specification language used (also called EAGLE) is based on extended affix grammars (EAGs) originally proposed by Watt & Madsen (1983) to exploit the benefits of W-grammars and affix grammars. The system transforms its EAG input into a form resembling a traditional context-free grammar. This grammar can then be used as input to a modified version of a context-free generator, also developed by Watt (1974).

The syntax analyser which is generated by EAGLE consists of routines for syntax analysis, static semantic checking, error recovery, and for dumping the derived syntax tree for use by subsequent code generation phases. The resulting syntax analyser is not table-driven, but instead produces directly executable code which implements a so-called characteristic finite state machine (CFSM). For each state of the CSFM a specific procedure is generated, thereby implicitly containing parsing information in the flow of control and allowing systematic error handling (both detection and correction).

The HLP78 (Helsinki Language Processor) (Räihä *et al.*, 1978) system from the University of Helsinki, was originally designed to study the applicability of using attribute grammars as input to compiler-compilers. The intention was that by storing attributes in the parse tree, the intricacies of the details of symbol table entries would be removed from the grammar level. However the designers found that an attribute-grammar-based input specification suffers from a lack of readability, since its essential graph structure is difficult to discern from a linear description. The original version of this system was restricted to accepting only pure LALR grammars (DeRemer, 1969) as its input. This guarantees that the grammar will be non-ambiguous, but imposes a serious restriction on the range of grammars which can be employed.

Due to the aforementioned short-comings of HLP78, it was later modified to incorporate a disambiguating mechanism to resolve parsing conflicts thus relaxing the constraints on the language specification. The manner in which a language's syntax is specified resembles that of a block-structured programming language with the intention of concentrating on nonterminals of a grammar. Thus all components of a nonterminal can be found "nested" within its specification in a hierarchical rather than rule-based fashion, with the outermost definition being for the start

symbol of the grammar. This is illustrated by the following outline specification for Pascal (taken from Koskimies *et al.*, 1988):

```
nont Program: ()      -- () means "no attributes"
token identifier: String = letter(letter|digit)*;
   ...
   nont ProgramParamList: ();
      ...
   end ProgramParamList;
   nont Block: ();
       ...
       nont LabelPart: ();
       ...
       end LabelPart;
       nont ConstPart: ();
       ...
       end ConstPart;
       ...
   end Block;
   Program = 'PROGRAM' identifier
                 (| '(' ProgramParamList ')'  ) ';' Block '.'
end Program.
```

The later version of this system, namely HLP84 (Koskimies *et al.*, 1988) produces a variety of the components of a compiler front-end, namely LL(1), SLR(1) and LALR(1) syntax analysers and provides two lexical analyser options, one table-driven and one as directly executable-code. All generated code is emitted in Pascal.

Another project (Roberts, 1988) has attempted to produce optimized syntax analyser generators (OPG). In this system a context-free grammar is used as input, and OPG scans this input to determine the appropriate type of syntax analyser to be used for each rule. Thus a "hierarchy" of complexity is established ranging over regular, LL(1), LR(1), context-free grammars, listed here in order of increasing power; the least-cost parsing method is chosen for a particular rule. OPG produces action

tables to be used in parsing, which are then optimized using traditional algorithms for optimization during the code generation phase of a conventional compiler.

The PRESTO syntax analyser generator (Elliott, 1988) was designed for languages having a large set of syntax rules that change frequently. Its specification language is similar to EBNF, augmented with actions to be taken as a particular part of the language is parsed. From the grammar rules supplied by the user, PRESTO constructs a syntax analyser consisting of a table of interpretive instructions which are language-independent and are stored on file in numeric form. This means that the driver program for the instructions can be written in a non-recursive language using a state stack and a return stack.

The generated syntax analyser acts as a finite-state automaton, reading parsing instructions from the stored table and the source which is to be parsed. The latter is scanned by a lexical analyser whose sole purpose is to place characters in a buffer; parsing can then be "backed up" to any point in the buffer (resulting from failed state transition), but any attempt to back-up further will be reported as a syntax error.

Other syntax analyser generators of note are LINGUIST-86 (Farrow, 1982), which concentrates on the use of attribute grammars for semantic analysis; RRP (Dwyer, 1988), which introduces a new notation – Regular Right Part grammars – for specifying SLR(1) syntax analysers; SLS/1 (Lewi *et al.*, 1975), which emphasizes semantic analysis and is able to produce two syntax analysers, one suitable for a production system and the other for educational purposes; and finally yacc, an LALR(1) syntax analyser generator produced for the Unix system which will be described in later sections in more detail, since it is used by the system developed during the work described in this thesis.

LL-syntax analyser generators

By automating LL-type parsing techniques, the resulting syntax analyser has the advantage of facilitating processing of its input in a single pass; LR-parsing will only permit single pass translation for simple programming languages such as a desk calculator but not for more typical PLs, because of the order in which information becomes available. However LL-parsing techniques have the drawback that they can only deal with a restricted class of grammars.

An early LL(1) syntax analyser generator project (Bochmann & Ward, 1978), took a previously designed LR-syntax analyser generator (Lecarme & Bochmann, 1974) and modified it to operate in a top-down manner. This system accepts a set of production rules to specify the language syntax, augmented with embedded variables, procedures and functions to deal with semantic constraints. These rules are expressed in a regular expression notation since this ensures that the grammar is free of left-recursion, which is a necessary precondition of LL(1) parsing. The system translates this RE description into an equivalent BNF representation, but still preserving the property of being non-left-recursive.

The ALL(1) compiler generator Aparse (Milton *et al.*, 1979), was designed to incorporate attributes into an LL(1) grammar, using embedded attribute evaluation rules and declarations written in C. By considering previously evaluated attributes, and thus providing a disambiguating mechanism, the syntax analyser is able to handle a wider class of grammars than normal LL(1) techniques. Aparse consists of four independent modules: a grammar preprocessor, an ALL(1) syntax analyser generator, an error-corrector generator and a table compactor. It produces a table-driven syntax analyser.

The Visible Attributed Translation System (VATS) (Berg *et al.*, 1984) is also an attributed LL(1) syntax analyser generator based on a previous LL(1) system called

ATS. In a similar manner to Aparse the grammar is augmented by attribute evaluation actions, but in addition the user is able to interactively examine the operation of the syntax analyser. The "visibility" model of VATS displays the current state of the syntax analyser at each step, and shows previous parsing reductions, semantic actions and error recovery as they occur; thus the task of debugging the compiler is considerably eased.

The LLGen syntax analyser generator (Fischer & LeBlanc, 1988), is a system which accepts a CFG specification and produces a set of tables to be used by a user-written driver program. The specification consists of three main sections: options requested for the run; a set of terminal symbols for the grammar; and a set of production rules. The grammar is augmented by user-supplied semantic actions which are each identified by a unique number in the syntax analyser's tables. In addition to these tables, LLGen also produces error-repair tables, with actions determined by the context-free grammar together with a list of repair costs (a default cost being inserted if no such list is given). A similar syntax analyser generator, LALRGen (Fisher & Leblanc, 1988), exists for generating LALR syntax analysers.

The motivation behind the development of the LLgen syntax analyser generator (Grune & Jacobs, 1988), was to produce LL(1) syntax analysers similar to those written by hand. LLgen accepts a parameterized extended CFG, with embedded semantic actions written in C. These parameters, which correspond to C data structures, are used for communication between grammar rules and semantic actions in a manner identical to parameters in function calls. This implements a mechanism for inheriting and synthesizing attributes within the grammar; in fact LLgen uses the same method for attribute storage allocation as in hand-written recursive-descent syntax analysers. Further user-supplied functions are used to resolve LL(1) conflicts in the context-free grammar, and their operation is guided by semantic

actions. An error-recovery mechanism called "acceptable set recovery" is also included.

The generated parsing functions from LLgen are emitted as C code (ie. not table driven), and may be produced from a number of separate files giving the language specification. These files can then be combined (in a manner similar to the Unix make facility) to form a single executable syntax analyser.

3.1.3 Semantic analyser generators

The automatic generation of routines to perform semantic analysis from a formal semantic description is a more complex and less-understood process than for lexical and syntax analysers. This is mainly due to the more elaborate semantic restrictions imposed by modern programming languages, and, of course, to the impossibility of context-free specification.

Many compiler writing systems have subsequently been based on attribute grammar specifications, generally enforcing the rule that the AG should not be circular, thus allowing practical systems to be developed. In order to save on compilation time and storage requirements, it has been suggested that attribute evaluation should be performed "on-the-fly" during parsing; this means that there is no need to construct a parse tree. Such a method was used in the SDELTA system (Lorho, 1977), but it restricts the class of attribute grammar which can be parsed to only those whose attributes can be evaluated in a left-to-right order. It can thus be used to construct one-pass compilers.

More generality can be gained by first building a parse tree of the program being compiled, and then decorating the tree with attribute evaluation rules. A method of tree traversal must then be defined which ensures that the order of attribute evaluation adheres to the constraints imposed by the attribute dependency graph

(Kennedy & Warren, 1976). The resulting syntax analyser will thus operate in a multi-pass manner. A system for generating such a syntax analyser was proposed by Fang (1972) using a collection of parallel processes, but this resulted in non-deterministic behaviour and is hence very inefficient. A deterministic system, called DELTA (Lorho, 1977), has been developed for general attribute grammars, but it too trades efficiency for generality since it constructs a dependency graph at compile time.

In order to gain efficiency, but still cover a reasonably large class of grammars, Bochmann (1976) suggested that attribute evaluation should be restricted to grammars which allow evaluation to be performed in a fixed number of left-to-right passes of the parse tree. This work was extended by Jazeyeri & Walter (1975) to yield a technique also used in HLP described by Räihä *et al.* (1978) and in the LINGUIST-86 system (Farrow, 1982), where passes of the tree are made alternately left-to-right and right-to-left. This method is also not generally applicable, but is restricted to grammars allowing this alternating-pass attribute evaluation. The MUG2 system (Ganzinger *et al.*, 1977) which was developed to produce multi-pass optimizing compilers, uses a similar method except that it restricts tree traversal, and hence attribute evaluation, to a number of top-down, left-to-right passes.

The intention of the so-called KW method (Kennedy & Warren, 1976) is to maximize the degree of determination of attribute evaluation order at evaluator construction time rather than at operation time. This approach was taken for efficiency reasons, but is a more difficult task. In this method, a number of recursive routines are generated to perform tree-walking. These routines follow a set of "plans", which are constructed from the grammar. A plan gives a sequence of evaluation actions to be performed when visiting a node, and visit actions, which specify the routines to be called next to continue the tree-walk. In contrast to the

pass-oriented systems described above, the KW method accepts the much larger class of absolutely noncircular grammars. For a grammar in this class, an evaluator's action at a node need not depend on the structure of the node's subtrees. This is because its control mechanism is tailored to the particular attribute grammar being evaluated. Saarinen (1978) describes a modification of Kennedy & Warren's method that saves space by keeping attribute values stackwise rather than nodewise.

More recently, researchers have begun to examine the possibility of using denotational semantics as a specification language, since they provide a formal description at a suitable level of abstraction from implementation detail (Raskovky & Collier, 1980; Jones & Schmidt, 1980; Thatcher *et al.*, 1980; Deschamp, 1980).

The SIS system (Mosses, 1975, 1978) uses semantic equations written in an applicative language. These equations are translated into a language based on λ-expressions called LAMB, which are then interpreted. This mechanism allows programs to be run and tested as soon as a formal semantic specification of the programming language used is available. Jones and Schmidt (1980) have investigated the use of state transition machines (STMs) as target code. Since STMs are closely related to the λ-calculus they facilitate the process of proving the compiler correct.

The PERLUETTE system (Deschamp, 1980) also stresses provability of the generated compiler. In this system a compiler is produced as a number of Lisp-lists which can be manipulated and evaluated, and it is planned to attach an automatic theorem prover to verify the compiler.

3.1.4 Code generator generators

The production of the code generation phase of a compiler has been the most difficult to automate. The goal of research directed towards this area of compiler

construction is to be able to take a formalized description of the target machine architecture together with an intermediate representation of the program being compiled; from these, machine code for the target architecture should be emitted.

One approach to easing the task of designing and writing a code generator is a procedural one (McKeeman *et al.*, 1970) and also Elson & Rake (1970) who concentrated especially on code generator specification languages. This involves taking a machine description, expressed in some formal language, and producing sets of tables to be used by a skeletal code generation routine. These tables should also be used to perform optimizations, where many of the techniques used are machine-independent, The portable C compiler (Johnson, 1978), uses machine description tables in its code generation phase, but it has not proved possible to generate these tables automatically.

The Production Quality Compiler-Compiler (PQCC) (Leverett *et al.*, 1980; Cattell, 1980; Cattell *et al.*, 1979) was developed at Carnegie Mellon University and work was mainly concentrated on the production of compiler back-ends for a variety of machines, to compete with hand-coded compilers. It works in a manner similar to yacc (Johnson, 1975), in that it accepts formal language and machine descriptions and builds tables to be used by a skeletal code-generation and optimization program called PQC as shown in Figure 3.3. The compiler-writer must supply a syntax analyser which emits an intermediate representation of the parsed program. This intermediate representation is an abstract syntax tree expressed in a form named TCOL. A standard syntax analyser generator like yacc can be used for this purpose. In order to pass this tree between compiler phases it is "flattened" for input and output into a notation known as linear graph notation.

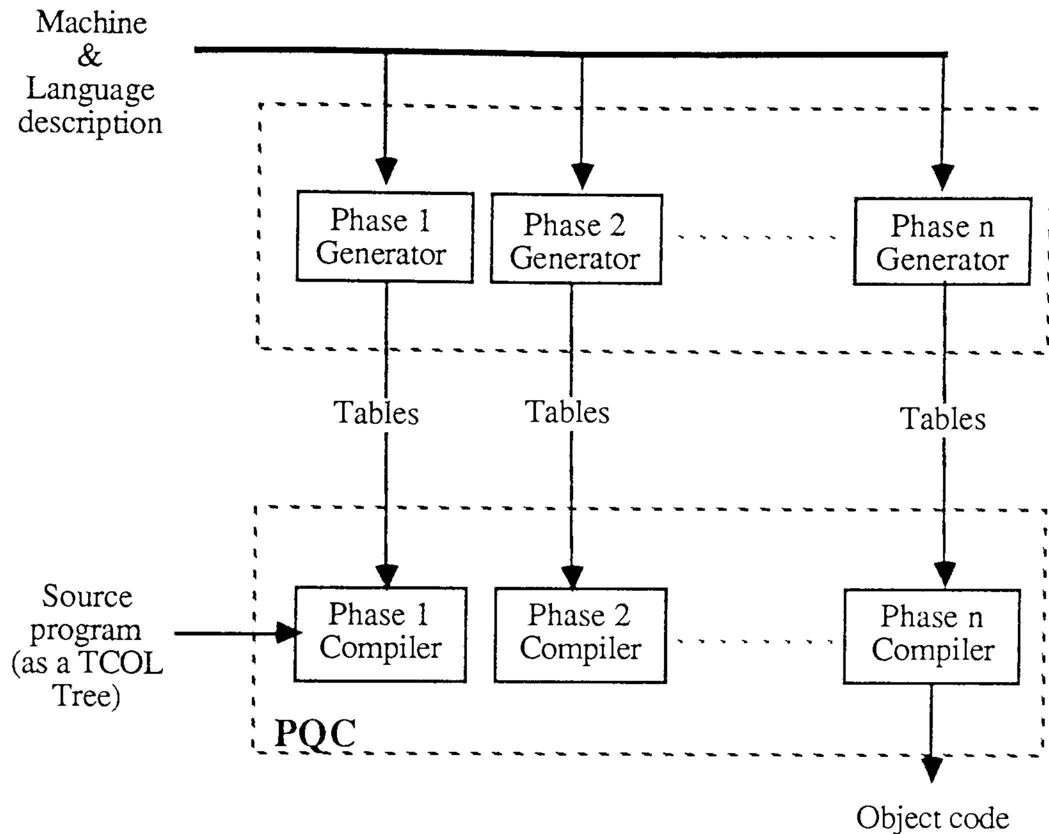


Figure 3.3 The PQCC compiler generator

The machine description used in PQCC is based on the ISP language developed by Siewiorek *et al.* (1982) and is non-procedural and easily human-readable. In this description, the user gives a mapping between machine operations and subtrees of the TCOL representation. The PQCC system reverses this mapping to create "templates" which relate TCOL subtrees to machine code sequences.

PQCC acts in several phases, as shown in Figure 3.4, based on the model of an optimized compiler called BLISS-11 (quoted in Leverett *et al.*, 1980). Each phase performs a particular part of the code-generation and code-optimization process, hence the specification can be developed and debugged as manageable subsystems. The TCOL tree is passed through a series of optimizations, and the system is parameterized to permit use on a variety of machines. The first optimization phase,

called FLOWAN, is machine independent and builds a graph of the basic blocks of the program. It then performs flow analysis, resulting in optimizations such as moving constant code outside loop bodies. The next group of phases, known collectively as DELAY, perform a number of source-to-source transformations. These phases include context determination (adding semantic information regarding the use of tree nodes as operands), operator propagation (which may result in a changed order of evaluation), and finding the most efficient address calculations. The TNBIND phase then performs register allocation, for variables and temporary results in expressions, and also storage allocation. The penultimate phase, named CODE, attempts to match subtrees of the optimized TCOL representation with patterns in the previously produced tables. The matching is goal-directed and takes account of cost functions for code sequences - if a match cannot be found, axioms are used to manipulate the subtree until it fits a pattern. Such a method does not guarantee to always find an optimal solution, but testing showed that it does in many cases. The last phase, called FINAL, emits machine code, performing any machine-dependent optimizations which were not found at the TCOL level.

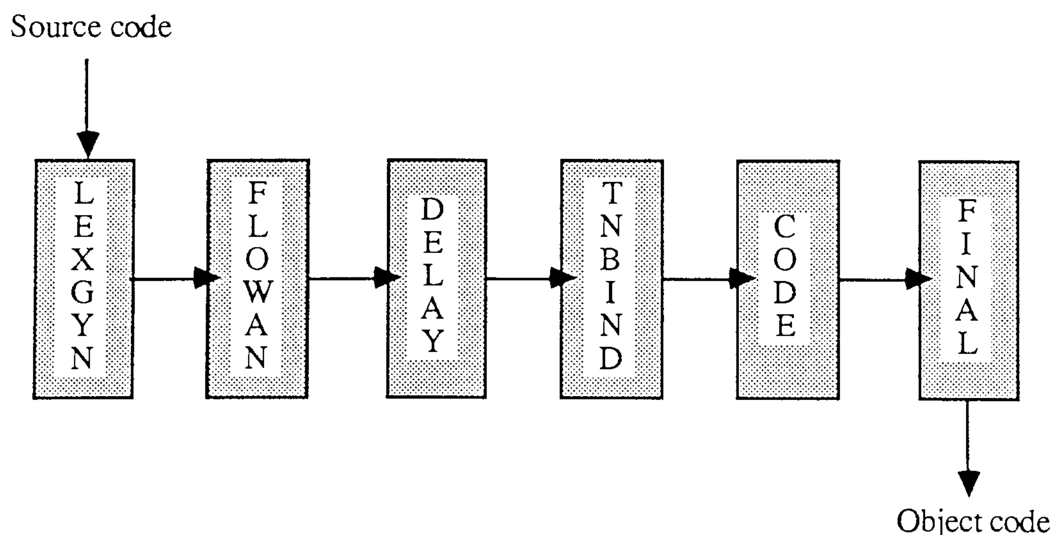


Figure 3.4 The PQCC generated compiler

In the table-driven code-generator developed by Glanville and Graham (1978), also discussed by Graham (1980), a pattern-matching scheme is also used, but it uses an LR-based algorithm rather than a goal-directed search. The user must supply storage allocation, binding and optimization procedures. The code-generator is provided with tables which are formed from a machine specification, and the intermediate representation used for the parsed program is Polish-prefix. The machine description is given in the form of productions of a context-free grammar in a language called TMDL, and defines a mapping from the intermediate representation to machine code sequences. The left-hand side of each production provides the destination of a computation, and the right-hand side consists of the prefix expressions with semantic rules, together with an equivalent assembly language format.

The LR algorithm parses the intermediate representation, matching subtrees with the TMDL description for the target machine and emitting code when a "reduction" is performed. This method leads to many shift/reduce and reduce/reduce conflicts, since the target machine description is often ambiguous. Shift/reduce conflicts are resolved in favour of a shift and reduce/reduce conflicts choose the longest rule as a match.

The efficiency of this method of generating a code-generator was tested by using it to replace the back-end of the portable C compiler, and it was found that it produced as good if not better code. However this approach does have its drawbacks; there is too close a mapping between the machine description and the intermediate representation used, which often means totally rewriting the description even for just a change of language being compiled. Also the system does not address the machine-dependent optimization issue, although it has been extended by Bird (1982) to handle machine idioms. A further extension has been proposed to use

attribute grammars for describing the target machine, machine-dependent optimizations and code-generation.

Frazer (1977), has used a knowledge-based approach in his system called XGEN which was written in LISP. This is an attempt at formalizing the ad hoc rules which are used by assembly language programmers. It was designed to generate good code for Algol-like languages, but does not provide a mechanism for dealing with machine-independent optimizations or global register allocation. It does however allow the specification of machine-dependent optimizations and local register allocation strategies.

The intermediate representation used in XGEN is called XL, which consists of a number of tuples. These tuples have only one operator but allow any number of operands, thus providing a means of giving extra context information. The tuples are recursively broken down until they are simple enough to produce equivalent assembly code.

Donegan *et al.* (1979) have proposed the use of a finite-state machine description in their code-generator generator language (CGGL) to produce a code-generator written in Pascal. As the code generator parses the intermediate representation of the program being compiled it passes through various states, emitting code as it does so, until it reaches the final state. This approach is noted for its simplicity, but may be an oversimplification of computer description, since it has difficulty dealing with more than one machine register. It is also an unsuitable method for specifying machine-dependent optimizations.

3.2 Summary

In this chapter, we have reviewed various approaches taken by researchers to the automation of the process of compiler design and construction. We have seen that

tools have been developed to automate each of the phases of a "traditional" compiler: lexical analysis, syntax analysis, semantic analysis, and code optimization and generation. For the lexical analysis phase, we presented systems based on the powerful and general-purpose regular expression notation, and also on less powerful, special-purpose notations. We surveyed a number of syntax analyser generators, which use either LR or LL-based algorithms in the syntax analysers that they produce, working from a BNF-like description of the language to be compiled. We saw that the majority of systems which automatically generate semantic analysers use attribute grammar techniques to describe static semantics; we also noted that more recently systems have been developed based on the mathematical rigour of denotational semantics. In the less well-understood area of code generators, we examined a variety of different approaches, which provide algorithms working in both a machine-dependent and machine-independent manner.

In the next chapter, we shall give a more detailed account of the program generators lex and yacc since the system described in this thesis is largely based on these two tools. In order to determine enhancements which can be made to their user interface and to the facilities they provide we shall examine, via an illustrative example, their use in the construction of compiler front-ends.

Chapter 4

lex and yacc: A Detailed Investigation

4.1 Introduction

Two of the most commonly used tools for the automatic generation of compilers are lex, a lexical analyser generator, and yacc, a syntax analyser generator. Since this thesis is largely based on these tools, we present in the following sections an account of how they function and interact in the production of a compiler front-end.

Also before beginning work on our proposed system which should provide a tool for automating compiler front-ends, we decided to code lex and yacc specifications for a small but illustrative example. In this way, we gained experience of the use of these two tools, and established a number of features which should be included in our system. We were thus able to gauge the ease of use of lex and yacc and to target a number of improvements.

The example chosen was to write a compiler for a small BCPL-like language called MSL (Mini System Language), which is used at Aston as an exercise in practical compiler writing in the programming language implementation module of a final year BSc course. The MSL language has facilities for manipulating values of type integer, boolean and text, and provides an elementary mechanism for establishing pointer-based data structures, which can include one-dimensional arrays. An MSL program can be procedurized, but all variables are treated as global. The language also has simple input/output routines.

In addition we augmented the lex and yacc specifications in order to produce a syntax tree, to perform semantic analysis and emit mnemonic code called TM (Elsworth, 1989) for a hypothetical von Neumann machine.

4.2 The Lex lexical analyser generator

The lex lexical analyzer generator was developed at AT&T laboratories by Lesk (1975). It is intended as a tool which accepts a high-level specification based on regular expressions, and generates a C program to recognize instances of these regular expressions appearing on the input stream. Hence it is useful for performing editor-like transformations on its input, or for "tokenizing" the input for use by a language syntax analyser. For this reason it represents a complementary tool to yacc, and indeed was designed with this in mind.

4.2.1 Input specification

The format of a lex specification is divided into three sections separated by "%%". The general format is thus:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the first and third sections are optional and are often omitted. The definition section allows the user to give names which are to be associated with commonly used regular expression patterns. For example since alphabetic characters are permitted in identifiers and keywords of a programming language, the user may wish to define `letter` as:

```
        letter              [a-zA-Z]
```

He can then use the name "letter" in the rule section, in place of its corresponding regular expression. The definition section also allows the inclusion of arbitrary C code fragments placed between the delimiters "%{" and "%}", which are simply copied unchanged into the file containing the generated lexical analyser.

The rule section is structured as a sequence of regular-expression/action pairs. Hence a typical rule will appear thus:

```
integer          { printf("found keyword INT \n"); }
```

Every time the characters `integer` are encountered in the input, the message "found keyword INT" is displayed.

The form of the regular expressions is very similar to these used in QED (Kernighan *et al.*, 1972) and in the Unix text editor "ed" (Thompson & Ritchie, 1975). Whenever the characters in the input stream match with one of the regular expressions, the corresponding action is executed. Character strings not matching any of the defined regular expressions are copied to the output (so that in a compiler application, we must ensure that all possible input, including error text, will be matched).

Lex allows the specification to be ambiguous, in which case the following rule is applied to resolve the ambiguity:

- The longest match is preferred.
- Among rules which match the same number of characters, the rule given first is preferred.

eg.

```
integer                 {/*  keyword action   */ }
[a-z][a-z]*             { /*  identifier action */ }
```

If the input symbol found is `integers` then the identifier action is performed, due to first rule given above, however if the input found is `integer` then the keyword action is performed, due to the second rule given above.

In order to implement this matching process, the lexical analyser may need to read a significant number of characters ahead. However when a string is finally matched, the input is appropriately backed-up, so that this lookahead process remains invisible to the user.

The action part of each rule is written by the user as a C code fragment which may include calls to his own functions or to functions contained in the lex library. Typically, when lex is used in conjunction with the yacc syntax analyser generator, these actions will include a statement which returns an integer value identifying the group of characters consumed from the input stream (ie. the language token). For example when matching an identifier we would use a rule such as:

```
[a-zA-Z][a-zA-Z]*            return (IDENTIFIER);
```

In many applications the user does not purely require the identifying integer value returned in the manner described above, but also the actual characters in the input which matched the regular expression. For this purpose, lex provides an external character array called `yytext`, which is over-written every time a regular expression is matched. The following example illustrates a rule which will echo the character string matched:

```
[a-z]+                 printf("%s", yytext);
```

Lex does not automatically evaluate the string which it has matched (eg. finding the integer value of a string of digits). Such an operation must be written by the user as a C code fragment in the associated action, but lex does provide an external variable `yylval` through which this value can be communicated to the syntax analyser. If

lex is used in conjunction with yacc then `yylval` should be declared in the generated syntax analyser as a union of all the types appropriate to the grammar symbols appearing in yacc rules.

In certain cases, the user may wish to override lex's method of choosing the longest match from the input stream. The special action `REJECT` is provided for this purpose, and calls the lex function `yyreject()`. Essentially, when lex matches a RE whose action contains `REJECT`, it passes on to the next alternative match. This is particularly useful when definitions of the items being matched overlap; for example we will need to distinguish between identifiers and keywords of the language. A lex specification to achieve this would include:

```
[a-zA-Z][a-zA-Z]*      { /*  if yytext isn't in the keyword table
                         then reject, using REJECT, this and go
                         and try the identifier rule   */ }

[a-zA-Z][a-zA-Z0-9]*  { /* identifier action   */   }
```

4.2.2 Operation of the generated lexical analyser

The lex specification, as described above , is converted into a C program held in a file `lex.yy.c`, see Figure 4.1, with a main function called `yylex()`. This is an interpreter which is driven by a number of generated tables, and acts as a deterministic finite state automaton. The tables are a representation of a transition diagram for the specified language. This results in the generated lexical analyser being quite fast even for a large collection of REs. In fact, the time taken to partition an input stream is proportional only to that input stream's length, regardless of the complexity of the lex rules, as long as the amount of rescanning required is not excessive, a condition satisfied for typical PLs. The only overhead of a complex and large lex specification is the increased size of the code produced to implement the lexical analyser.

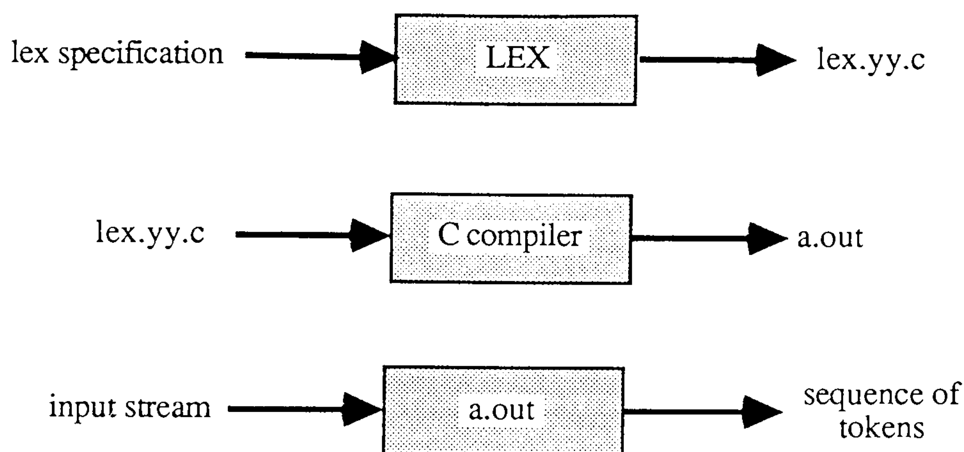


Figure 4.1 The lex lexical analyser generator

According to Jacobson (1987), an author of flex (Paxson, 1988) , a lex generated lexical analyser can be tuned to gain a factor of 10 performance improvement. flex is a rewrite of lex intended to improve some of lex's deficiencies: in particular, it generates lexical analysers much faster, and the analysers use smaller tables and run faster. flex also provides tracing facilities to allow monitoring of the generated lexical analyser as it matches input characters with regular expressions; whenever a pattern is recognized the lexical analyser will write a line of the form:

```
--accepting rule $<rule nº>
```

4.3 The yacc syntax analyser generator

Yacc, which is an acronym for "Yet Another Compiler-Compiler", was developed by Johnson (1975) at the AT&T Laboratories and runs mainly on Unix based systems. It is a tool designed to generate an LALR(1) syntax analyser for a language, given a specification in the form of a context-free grammar, and has been used in the production of many Unix utilities, in IDL (Lamb, 1987) and in the PQCC project (Leverett *et al.*, 1980). It is not strictly a compiler-compiler since it does not generate code to perform semantic analysis and code generation, however it does allow the user to manipulate a semantic stack (a stack of attributes associated

with the grammar symbols used), and to associate semantic actions with grammar rules. The syntax analyser generated by yacc is intended to be used in conjunction with a lexical analyser, which can be either hand-coded or produced by a lexical analyser generator such as lex. Figure 4.2 shows an outline of the syntax analyser's operation.

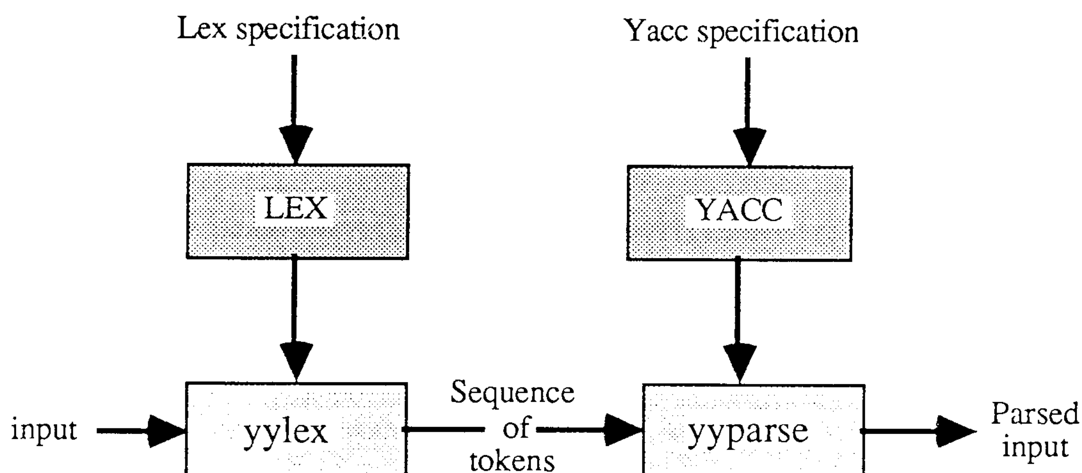


Figure 4.2 Cooperation of lex and yacc

Since the parsing algorithm used in yacc is LALR(1), its power is limited by this method with respect to the class of grammars for which it can generate a syntax analyser. However it does allow ambiguous grammars to be used, and provides a mechanism for resolving conflicts in such grammars.

4.3.1 The input specification

Input to yacc is separated into three parts: the declaration, production and user-routine sections, where the separation is denoted by "%%". Thus it has the following structure:

```
{ declaration }
%%
{ production}
%%
{ user-routine }
```

We shall now examine each of these sections in more detail.

Declaration Section

The declaration section has two optional parts. The first of these, delimited by %{ and %}, contains ordinary C declarations which are used to declare variables used by the user-written routines to deal with semantic actions. This part can also contain C compiler directives such as #include and #define, for example:

```
%{
#include     <string.g>
#include     "tree.h"
#define      STSZ  64
...
char         namestack[STSZ];
int          sp;
%}
```

All lines enclosed by %{ and %} are copied to the parser; therefore, they must be in a correct C syntax.

The second, and most important part contains declarations of the tokens returned by the lexical analyser. The declaration of tokens may have the following form:

```
%token      token1 integer1
%token      token2 integer2
```

The optional integers following a token declaration give a numeric value to that token, and they must be unique. If the integer value is not explicitly stated, then yacc assigns the token a value above 257, incrementing the value by one for each token it deals with.

Production section

The production section contains the context-free grammar of a language, expressed in a BNF notation, augmented with user-defined actions. The left-hand side of each production is a nonterminal of the grammar; the right-hand side is a sequence of zero of more alternatives separated by a bar "|". Each alternative consists of both terminals and other nonterminals. A quoted single character on the right-hand side is taken to be a terminal symbol, and unquoted strings of letters and digits not declared to be tokens are assumed to be nonterminals. There must be a production in the grammar for each such nonterminal symbol. The form of a yacc production is as follows:

```
Production-name : Production-body ;
```

User-defined actions

The user can insert actions to be performed when a production has been recognized by the syntax analyser. Such actions are written as a C code fragment delimited by "{" and "}", and can be placed anywhere in the right-hand side of a production, provided that this does not cause confusion with other productions. A number of special symbols in an action are used to refer to specific parts of the right-hand side. The symbol $$ represents the attribute value of the nonterminal appearing on the left-hand side of the production; the symbol $i (where i is an integer) represents the value of the i^{th} grammar symbol of the right-hand side (which may be a terminal or a nonterminal). Normally the action will compute the value of $$ in terms of some function of the $i's. If no action is specified, the default is to evaluate $$ as the value of the first grammar symbol, that is $1. For example, the operation of a simple desk calculator would be written as:

```
expr : expr '+' term  { $$ = $1 + $3; /* expr = expr + term */ }
     | term;          { $$ = $1;  /* expr  = term */      }
```

Ambiguity

As previously mentioned, yacc allows the use of ambiguous grammars. In order to resolve an ambiguous production, the user may provide disambiguating rules, but if these are not supplied yacc takes pre-defined default action. Verification that the ambiguities have been resolved as user intended can be done by examining a trace file y.output, which is produced by yacc when called with its "-v" option. This file contains a list of the states entered by the syntax analyser during its operation.

Two different types of conflicts can occur in LR parsers; shift/reduce and reduce/reduce. Shift/reduce conflicts occur when the parser has to decide between shifting, or reducing by a production. Reduce/reduce conflicts occur when the parser has to decide which of several productions to reduce.

The default action taken by yacc to resolve ambiguity is similar to that of Aho & Johnson (1974). A shift/reduce conflict is resolved in favour of the shift (which solves such problems as the "dangling-else"); a reduce/reduce conflict is resolved in favour of the production appearing earliest in the specification.

If the user wishes to provide his own disambiguating rule, this is done by specifying the precedence and associativity of operators in a table in the declaration section. Associativity is indicated by %left (left-associative), %right (right-associative), or %nonassoc (non-associative). Precedence is given by listing the operators in ascending order of priority. Thus most shift/reduce conflicts are resolved by giving precedence and associativity not only to each symbol but also indirectly to each production involved in a conflict. In situations where an operator can be either unary or binary (eg. '-'), the user can enforce a particular precedence by appending the following "tag" to a production:

```
%prec <terminal>
```

where the terminal's precedence and associativity have been given in the declaration section.

Error-recovery

The error-recovery strategy used by yacc is a form of "panic mode" called "error production" as described in section 2.2.1.5. In this method, the user augments the grammar with error productions which are of the form:

```
A → error B
```

Where A is a nonterminal and B a sequence of grammar symbols (both terminals and nonterminals). The symbol `error` is a reserved word, and yacc treats a production containing it like any other production. When an error is detected the syntax analyser behaves as if it had just seen the special symbol `error` immediately before the token which caused the error. The syntax analyser then looks for the nearest production rule for which the error symbol is a valid token and resumes processing at this rule.

A yacc-compatible tool called SERCC (Systematic Error Recovery Compiler Compiler) has recently been developed (Yang *et al.*, 1988). It uses an extension to the "forward move" algorithm (Pennello & DeRemer, 1978), combined together with the "panic mode" supported by yacc, for its error recovery. It is claimed that the users of yacc are not obliged to change their yacc input to serve as input to SERCC. However, when compared with yacc, tests have shown that the number of states generated by SERCC increases by about one third; also the size of the parsing table is more than doubled. In the same context Park (Park, 1988) designed a system, which accepts actions written in a special-purpose language called y+, to be a preprocessor for yacc.

User-supplied routine section

Since the actions specified by the user to be performed during parsing may need be more than a just few statements in length, yacc provides a section in which the user can declare his own C functions. These can then be called from their associated productions. In addition, the user must supply a function called `yylex()`, unless this has been provided by an external reference. The generated syntax analyser calls this function to scan the input and return token values corresponding to those listed in the declaration section. These token values are returned to the syntax analyser by `yylex()` via the yacc defined variable yylval. `yylex()` can either be hand-written or generated by a tool such as lex.

4.3.2 Operation of the generated syntax analyser

The specification as described above is transformed by yacc into a C program called `y.tab.c`, whose main procedure is called `yyparse()`, see Figure 4.3. `yyparse()` operates as a finite state automaton and is driven by interpreting a set of tables which specify state transition based on the input tokens. The syntax analyser attempts to reduce these tokens to the nonterminals appearing in the BNF productions and carries out the user-defined actions.

The finite state automaton used has four possible actions: shift, reduce, error and accept. A shift action is performed when the next token is valid in the current state. A new state is then pushed onto the stack and becomes the current state. When the syntax analyser has successfully matched the entire right-hand side of a production, a reduce action is taken. When reducing, the syntax analyser will pop the number of states corresponding to the number of grammar symbols on the right-hand side of a production; the current state is then the one remaining on the top of the stack. If the input cannot be matched against any production then the syntax analyser

performs an error action as discussed earlier. If the user has not supplied an appropriate error production, the syntax analyser simply prints an error message. When the syntax analyser reaches the end-marker of the grammar it enters the accept state, and returns an indication that its parsed input was a valid sentence of the given language.

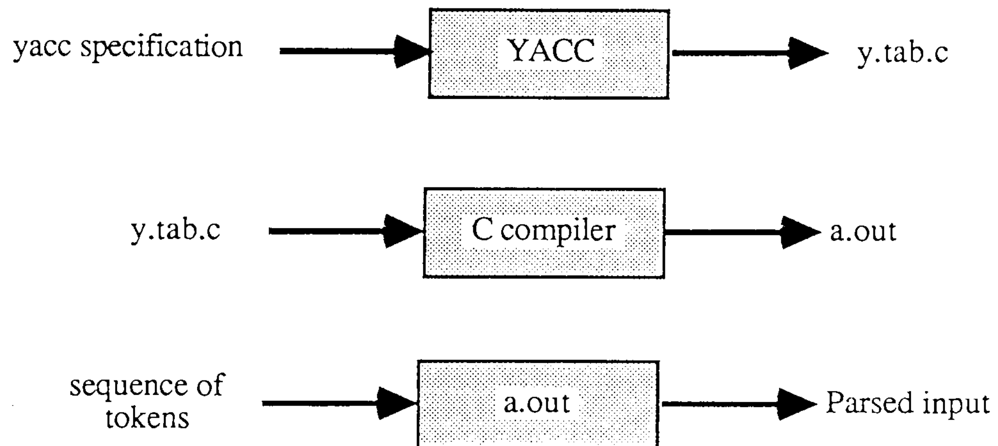


Figure 4.3 The yacc syntax analyser generator

4.4 The use of lex and yacc

In the following sections we describe the implementation of an MSL multi-pass compiler using lex and yacc, and include observations regarding the practical use of these tools with special reference to the situations where the onus still rests with the user to add extra code. We have chosen to build a multi-pass compiler for MSL rather than single-pass since in practice relatively few PLs may be recognized using a single-pass compiler; also in order to be able to use yacc and its method of passing attributes in a single-pass style it is necessary to use undesirable techniques such as rearranging the grammar (Atteson *et al.*, 1989).

We present first the lex specification, followed by the yacc specification of MSL's syntax, and finally the routines and data structures added for semantic analysis and code generation.

4.4.1 MSL lex specification

MSL has the following lexemes:

- integer literal: an integer is a string of digits representing an integer in the range 0..maxint of the target machine.
- boolean literal: a boolean literal is given by the strings TRUE or FALSE, which are keywords.
- text literal: a text literal is a string of any ASCII characters delimited by double quotes.
- identifiers: an identifier is denoted by an arbitrary length string of letters and digits, beginning with a letter, with the first 12 characters significant. Case is not significant.
- keywords: there are a number of keywords which are reserved, and consists of a string of letters, where case is not significant.
- operators/delimiters: MSL has a number of logical, arithmetic operators and various delimiters.

Comments in MSL begin with "--" and continue until the end of the current line of the program. These can be dealt with during lexical analysis, by simply consuming characters without returning a value to the syntax analyser.

We can specify these lexemes using the following lex definitions and rules:

<u>integer literals</u>

```
digit                    [0-9]
```

```
int                     ({digit})({digit})*
{int}                   { return(INT); }
```

text literals

```
anybut_DQ_NL            [^\n\"]
text                    "\""{anybut_DQ_NL}*"\""
{text}                  { return(TEXT); }
```

Identifiers

```
letter                  [A-Za-z]
idr                     ({letter})(({letter})|({digit}))*
{idr}                   { return(IDR); }
```

Keywords

```
("I"|"i")("F"|"f")       { return(IFSYM);  }
("F"|"f")("I"|"i")       { return(FISYM);  }
("D"|"d")("O"|"o")       { return(DOSYM);  }
```

... etc for all keywords of MSL

Operators and delimiters

```
":="                     { return(ASSop); }
"="                      { return(EQUop); }
"+"                      { return(PLUSop);}
```

... etc for all operators and delimiters of MSL

Comments

```
"--"{anybut_NL}*[\n]     { /* do nothing */ }
```

Assembled as a valid lex specification, we obtain:

```
%{
#include                "y.tab.h"
#include                <ctype.h>
%}
anybut_NL               [^\n]
anybut_DQ_NL            [^\n\"]
digit                   [0-9]
uplow_case              [A-Za-z]
```

```
stringpic                  {anybut_DQ_NL}
letter                     {uplow_case}
text                       "\""{stringpic}*"\""
int                        ({digit})({digit})*
idr                        ({letter})(({letter})|({digit}))*
%%
...
":="                        {   return(ASSop);    }
"="                         {   return(EQUop);    }
"+"                         {   return(PLUSop);   }
...
("I"|"i")("F"|"f")          {   return(DOSYM);    }
("F"|"f")("I"|"i")          {   return(FISYM);    }
("O"|"o")("D"|"d")          {   return(ODSYM);    }
...
{idr}                       {   return(IDR);      }
{int}                       {   return(INT);      }
{text}                      {   return(TEXT);     }
"--"{anybut_NL}*[\n]        {   /* it is a comment */ }
...
%%
```

Primitive lex specification for MSL

As can be seen from the above specification, it can be quite cumbersome to specify the keywords of a language explicitly in the lex rules. This form of specification also results in a large number of cases in the main C switch statement generated by lex. In many programming languages the patterns (REs) for identifiers and keywords have much in common; often a neater and more efficient solution is to determine their inter-relation, and use a function to distinguish between them, by searching a table of keywords.

In considering the problem of keywords and identifiers more generally, we can identify three possible cases. If K is the set of keywords, and I the set of identifiers, then these cases are:

i) $I \cap K = \emptyset$ - the keywords and identifiers are two disjoint sets
ii) $I \supseteq K$ - the keywords are a subset of the identifier set
iii) $I \cup K \neq I$ and $I \cap K \neq \emptyset$
- the keywords and identifiers are two overlapped sets

The first of these would occur, for example, in a language which uses only lower case for its identifiers and only upper case for keywords, ie:

```
L = [A-Z]
l = [a-z]
Keyword = LL*
Identifier = ll*
```

The second case would arise given the following example definitions:

```
l = [a-z]
d = [0-9]
Keyword = ll*
Identifier = l(l|d)*
```

The third case is rarer, but would arise where certain characters were allowed only in keywords, and others only in identifiers such as:

```
l = [a-z]
d = [0-9]
Keyword = l(l|'-')*
Identifier = l(l|d)*
```

In the case of MSL the keywords are a subset of identifiers thus the following specification may be used.

```
...           {   definition section }
%%
...
":="          {   return(ASSop);       }
"="           {   return(EQUop);       }
"+"           {   return(PLUSop);      }
...
{idr}         {  return(screen());}
{int}         {   return(INT);         }
{text}        {   return(TEXT);        }
%%
```

The function `screen()` is a routine that checks the input which has been recognized against a keyword table and returns a code indicating whether it was a keyword or identifier.

A user who is conversant with lex will know which of the above cases applies; the method for dealing automatically with these three cases in a lex specification will be discussed in chapter 6.

4.4.2 MSL yacc specification

The context-free syntax specification for MSL was given in EBNF for the aforementioned exercise. As noted in chapter 1, EBNF is commonly used in such specifications since the features which it has in addition to those of BNF, make syntactic descriptions more understandable and natural. We present below the syntax of MSL in EBNF:

```
Program       =  ["RESERVE" int ] {ProcDec} Series ".".
ProcDec       =  "PROC" idr  ["(" Idrlist ")"] Series "END".
Idrlist       =  idr {"," idr}.
Series        =  Stmt {Stmt}.
Stmt          =  Assignst | Whilest | Ifst
               | Callst | Readst | Writest.
Assignst      =  StoreAccess ":=" Expr.
Whilest       =  "WHILE" Expr "DO" Series "OD".
Ifst          =  "IF" Expr "THEN" Series ["ELSE" Series] "FI".
Callst        =  "CALL" idr ["(" Exprlist ")"].
Exprlist      =  Expr {"," Expr}.
Readst        =  "READ" Optionalplus  Readinlist.
Writest       =  "WRITE" Optionalplus  Writeoutlist.
Readinlist    =  Optionalhash StoreAccess
                 { "," Optionalhash StoreAccess }.
Writeoutlist  =  Optionalhash Expr {"," Optionalhash Expr}.

/*  The following rules specify the lexical structures and */
/*  in this exercise these have already been dealt with    */
/*  using lex but are included for completness             */

Optionalhash  =  ["#"].
```

```
Optionalplus   =  ["+"].

Expr           =  Operand {opr Operand}.

Operand        =  int | text | "TRUE" | "FALSE"

               | "@" idr | StoreAccess | "(" Expr ")".

StoreAccess    =  idr ["!" Operand].

idr            =  letter { letter | digit }.

int            =  digit { digit }.

text           =  """" {stringpic} """" .

stringpic      =  anybut_DQ_NL.
```

Figure 4.4 MSL syntax in EBNF

Although it is more convenient to specify the syntax of a programming language using EBNF, unfortunately yacc only allows rules to be given using BNF. In order to transform the EBNF specification into its BNF equivalent, we need to examine the extra features of EBNF and how to re-express rules which use these features as BNF rules. There are three such features:

Optional items: If part of the RHS of a rule is enclosed in square brackets "[" and "]", then this part of the rule is optional; eg. an IF statement in most PLs would be described as:

```
Ifst = IF Expr THEN Stmt [ELSE Stmt].
```

which means that an IF statement may or may not have an ELSE part. The equivalent yacc rule would be:

```
Ifst   :  IF Expr THEN Stmt Elsepart;
Elsepart :  /* empty */
         |  ELSE Stmt;
```

Repeated items: If part of the RHS of a rule is enclosed in braces "{" and "}", then it may be repeated zero or more times; eg. a series of statements in many PLs would be described as:

```
Series = Stmt {Stmt}.
```

which means that `Series` consists of a statement followed by zero or more statements. The equivalent yacc rule would be:

```
Series  :  Stmt Stmts;
Stmts   :  /* empty */
        | Stmt Stmts;
```

Grouped items: If part of the RHS of a rule is enclosed in round brackets "(" and ")", then it is considered as a single entity. Normally this is used to override the precedence of catenation over alternation; eg. `constant` in Pascal is defined as:

```
constant   = [sign](unsigned-number|const-identifier)
           | character-string.
```

The equivalent yacc rule would be:

```
Constant     : Opsign  Grouped-item
             | character-string;
Opsign       :  /*  empty  */
             | sign;
Grouped-item : unsigned-number | constant-identifier;
```

The algorithm for performing this transformation will be discussed in chapter 6.

The full yacc specification obtained from the original EBNF grammar is shown in appendix A. This provides purely a recognizer of a context-free-correct program written in MSL. In order to produce the entire compiler for MSL, lex and yacc specifications must be augmented with routines to build a syntax tree and symbol

table, to perform semantic checking, and finally to generate code for the target machine. We describe this more fully in the next section.

4.4.3 Augmented lex specification for MSL

The lex specification given in section 4.3 simply returns token numbers to the parser; however, for certain types of tokens, additional information must be made available. This additional information will be held at appropriate points in the syntax tree, to be used by semantic analysis and code generation routines. Such information needs to be extracted and evaluated from the characters forming the token held in the external array `yytext`.

The process of augmenting a lex specification varies from one language to another, depending on the type of lexemes, the form in which the lexemes are given, and their evaluation functions. For example, for numeric literals, some code needs to be added to a lex specification to determine the numeric value of the characters in `yytext`. This may involve evaluation in various number bases, since many programming languages allow numeric values to be specified as binary, denary, octal or hexadecimal constants. Modula-2, for example, uses the letters B and H appended to the end of a numeric value to indicate base 8 and 16 respectively; eg.

`yytext = "110B" evaluates to` 6_{10}

In MSL, only one representation is allowed which is a denary representation, thus the augmented lex rule for numbers will be:

```
{int}        { yylval.int = evaluate_denary(); return (INT);}
```

String literals also require some processing, since they may contain special character sequences to represent non-printable characters, or "escaped" character values. For example, in C a string may contain the characters "\n" to indicate a carriage return;

when the string is evaluated and placed in the syntax tree, these characters should be replaced by the ASCII value of carriage return; eg.

```
yytext  = "ab\nc"   evaluates to   ab<CR>c
```

A further example is the need to replace multiple quotes by a single quote, as in the use of a quote within a character string in Pascal. The same translation process also applies to single character literals.

In MSL, a string is delimited by a double quote characters ("), and consists of any printable characters except double quotes or carriage return, similar to Pascal. The augmented lex rule for strings in MSL is given below:

```
{text}     { yylval.text = evaluate_Pstr(); return (TEXT); }
```

Tokens such as identifiers and operators must also have their values stored in the syntax tree for later use by semantic analysis and code generation. The name of an identifier will need to be available in order to be able to look it up in the symbol table. The type of operator used determines the instructions which should be emitted during code generation.

In MSL, the augmented lex rules for operators and identifiers are given below:

```
"+"      { yylval.Vopr = PLUSOP;  return(PLUSOP);   }
"-"      { yylval.Vopr = MINUSOP;  return(MINUSOP);   }
{idr}    { yylval.Vidr = evaluate_idr(); return(screen()); }
```

The full augmented lex specification is given in appendix A.

4.4.4 Augmented yacc specification for MSL

In order to be able to build a syntax tree, we must include appropriate data structure declarations for nodes of the tree. In general, each rule in the yacc specification will correspond to one tree node, and this node will contain pointers to other nodes for nonterminals on the RHS of the rule, and the values of certain terminals. Thus, since yacc generates C code, we need to specify a number of appropriate C struct data types. Each struct may be a union of different types when a rule contains more than one alternative in its RHS; for example given the following simple rule:

```
Stmt  : Assignst | Whilest | Ifst | Callst | Readst | Writest.
```

we will need a struct to hold a program node such as:

```
struct  Stmt_type {
  int     type;          /* tag to tell which union alternative */
  union {
  struct  Assignst_type  *Assignst; /* type=1, first alternative  */
  struct  Whilest_type   *Whilest;  /* type=2, second alternative */
  struct  Ifst_type      *Ifst;     /* type=3, third alternative  */
  struct  Callst_type    *Callst;   /* type=4, fourth alternative */
  struct  Readst_type    *Readst;   /* type=5, fifth alternative  */
  struct  Writest_type   *Writest;  /* type=6, sixth alternative  */
  }RIGHTSIDE;
};
```

Data type for "Stmt" node in MSL syntax tree

In order to actually build the syntax tree, given the necessary data structure definitions as described above, we need to insert actions into the yacc specification. These actions allocate the appropriate amount of dynamic store when a particular rule has been matched in the input stream. The rule then returns a pointer to this allocated store as its result through the yacc internal variable $$. In this way, since yacc produces an LR-parser, the syntax tree is created in a bottom-up manner. Thus for the `Stmt` rule given above, assuming we have a function `mknode()` which acquires the storage necessary for one tree node, the yacc rule for `Stmt` now becomes:

```
Stmt =    Assignst          {  $$ = mknode(2,1,$1);  }
        | Whilest           {  $$ = mknode(2,2,$1);  }
        | Ifst              {  $$ = mknode(2,3,$1);  }
        | Callst            {  $$ = mknode(2,4,$1);  }
        | Readst            {  $$ = mknode(2,5,$1);  }
        | Writest           {  $$ = mknode(2,6,$1);  };
```

The first parameter of the `mknode()` function is the number of store units to acquire; the other parameters are entered into these store units.

Once the syntax tree has been constructed in this manner, semantic analysis and code generation can be performed by traversing the tree left-to-right and depth-first, performing appropriate operations for each node as it is visited. Thus we require a number of "tree-walk" routines for the tree as described by the data structure definitions. For each node of the tree a walker routine is required. Consider the following rules:

```
Ifst        : IFSYM  Expr  THENSYM  Series  Op_Else  FISYM
                              { $$ = mknode(4,1,$2,$4,$5); };
Op_Else     : /* empty */
            | ELSE Series          { $$ = mknode(2,1,$2); };

Operand     : INT                  { $$ = mknode(2,1,$1); }
            | TEXT                 { $$ = mknode(2,2,$1); }
            | TRUESYM              { $$ = mknode(2,3,$1); }
            | FALSESYM             { $$ = mknode(2,4,$1); }
            | INDIRsym IDR         { $$ = mknode(2,5,$2); }
            | StoreAccess          { $$ = mknode(2,6,$1); }
            | OBsym Expr CBsym     { $$ = mknode(2,7,$2); };
```

The simple tree-walk routines that purely visit the nodes without performing any operations would be:

```
walk_Ifst(ptr)
IFST_TYPE  ptr;
{
      walk_Expr(ptr -> Expr);
      walk_Series(ptr -> Series);
      walk_Op_Else(ptr -> Op_Else);

}  /*  end of walk_Ifst()   */

walk_Op_Else(ptr)
OP-ELSE_TYPE  ptr;
{
```

```
        walk_Series(ptr -> Series);

} /* end of walk_Op_Else()  */


walk_Operand(ptr)
OPERAND_TYPE   ptr;
{
switch(ptr -> type) {
case  1  :  case  2  :  case  3  :  case  4  :  case  5  :
            break;
case  6  :  walk_StoreAccess(ptr->RIGHTSIDE.StoreAccess);
            break;
case  7  :  walk_Expr(ptr -> RIGHTSIDE.Expr);
            break;
default  :  printf("ERROR  -  wrong alternative number\n");
            break;
}/* end of  switch  */

}  /* end of walk_Operand()  */
```

However, if what we require is to perform the semantic analysis and code generation, then the walker routines must include code to do so. The following is the augmented walker routines for the above rules:

```
walk_Ifst(ptr)
IFST_TYPE  ptr;
{
int   cj, uj;

      walk_Expr(ptr -> Expr); cg2(JF,0); cj = PSused;
      walk_Series(ptr -> Series);
      if(ptr -> Op_Else)
      {
         cg2(J,0);
         uj = PSused;
      }
      PS[cj] = PSused+1;
      walk_Op_Else(ptr -> Op_Else);
      if(ptr -> Op_Else)
         PS[uj] = PSused+1;

}  /* end of walk_Ifst()  */

walk_Op_Else(ptr)
OP-ELSE_TYPE  ptr;
{
int   cj, uj;

      if(ptr) walk_Series(ptr -> Series);

}  /* end of walk_Op_Else()  */

walk_Operand(ptr)
OPERAND_TYPE   ptr;
{
BOOLEAN  onstack;
```

```
int      STpos;
int      DSloc;

switch(ptr -> type) {
case  1  :  cg2(LC, ptr->RIGHTSIDE.intsym); break;
case  2  :  textual = TRUE;
            cg2(LC,textaddress(ptr->RIGHTSIDE.text)); break;
case  3  :  cg2(LC, TMtrue); break;
case  4  :  cg2(LC, TMfalse); break;
case  5  :  idrchars = malloc(strlen(ptr->RIGHTSIDE.idr)+1);
            strcpy(idrchars, ptr->RIGHTSIDE.idr);
            checkdeclared(&STpos);
            cg2(LC, getRTSL(STpos)); break;
case  6  :  walk_StoreAccess(Rv,ptr->RIGHTSIDE.StoreAccess,
            &onstack, &DSloc); break;
case  7  :  walk_Expr(ptr -> RIGHTSIDE.Expr); break;
default  :  printf("ERROR - wrong alternative number\n");
            break;
         }/* end of  switch  */
}
```

During tree traversal, semantic analysis and code generation must be performed. Full detail of routines to achieve this can be found in appendix A. Routines which we added to perform semantic checking and to build the symbol table are:

`CSerror(), STlookup(), checkdeclared(), checkprocidr() and checkFPidr().`

Routine to produce a mnemonic code equivalent of the source program are:

`CGloadcontents(), getRTSL(), textaddress(), cg1(), cg2() and cg3().`

Also included are routines to output the symbol table and a generated code, these being:

`ListSymbTab() and ListTMcode().`

Manifest constants and symbol table data structure declaration are given in the header file `define.h`.

4.5 Summary

In this chapter we have given a more detailed account of the use of lex and yacc in the construction of compiler front-ends, since the system described in this thesis is largely based on these two tools. We have also seen in this chapter how lex and yacc specifications can be constructed for a simple programming language, namely MSL. We have shown informally the process of how a language implementor proceeds from a "reference manual" grammar to lex and yacc specifications. Although this process is straightforward for an experienced language implementor, it certainly requires a definite effort and is error-prone. A significant amount of training is required to be able to use them effectively. From these specifications we can obtain only a recognizer for a syntactically correct MSL program. Although the declaration of the syntax tree, and routines for building this tree and for visiting its nodes performing semantic analysis and code generation is mechanical, this must still be added when using lex and yacc. We maintain that much of this additional effort can be saved by providing a system which takes a single language specification using a notation convenient to the user, and which performs automatically the in-core manipulation and input/output of the tree. The aim is therefore to allow the user to input a "reference manual" grammar for a particular programming language, and to generate code for the front-end of a compiler for this language. In the next chapter we shall describe the design and implementation of such a system.

Chapter 5

The Problem and the Proposed Solution

5.1 Rationale and intended goals

We have seen in Chapters 3 and 4 that many systems have been developed which attempt to automate the construction of individual phases of the compilation process, or indeed the construction of an entire compiler. We note that these systems are deficient in either of two respects: they either try to provide a large number of facilities at the expense of clarity of the input specification, or they provide an easy-to-use interface, but do not include many of the features necessary for writing realistic software. The first of these two kinds of deficiency results in the user being given simply a "higher-level" programming language in which to write the compiler; the second results in a necessity to write many auxiliary routines to perform tasks not provided automatically by the system.

The most common approach to dealing with generation of a compiler front-end is to require the user to supply a separate specification for each component phase (eg. lexical and syntactic specification). This almost always involves a different form of input notation for the production of each compiler phase.

Having noted the above shortcomings of existing systems our intention is thus to develop a system which has a simple and clear user interface, but which provides most of the elements necessary for automatic compiler front-end construction. Our system, called CORGI (COmpiler-compiler from Reference Grammar Input) uses a single input specification to describe all the relevant aspects of the language to be compiled, and extracts from this, the information needed to build both the lexical

and syntax analysis phases of the compiler. Thus we have unified what previously required two specifications into a single entity. Additionally, we aim to eliminate much of the requirement for user-written auxiliary code. To increase the ease-of-use of the system this specification is given in a form similar to that found in many language reference manuals, namely EBNF. In addition to the basic functions for recognizing a correct program in the given language, the system should also automatically generate declarations for the data structures required to build an Abstract Syntax Tree (AST), and also a number of routines for manipulating these data structures and for storing and retrieving them from permanent storage devices.

5.2 Development environment

Many systems have been developed from scratch, without using existing software already available for fulfilling many of the requirements. Whilst this approach has its merits, we believe that a more reasonable alternative is to take advantage of existing software where possible, and avoid the overhead of "re-inventing the wheel". We therefore chose to adopt a "layered" approach, building additional functionality on top of the most popular currently existing tools, namely lex and yacc, but to hide this from the user, by providing a more convenient interface, and by automatically producing features not directly available with these two tools. We identified the following shortcomings of lex and yacc, which our system should address:

- to make best use of lex and yacc a detailed technical understanding of their internal operation is necessary
- preparation of the lex and yacc specifications is error-prone and tedious
- yacc requires a "low-level" specification (BNF), a backward step from EBNF
- lex and yacc deal with unduly limited aspects of compilation. Much user-written code is still needed (eg. syntax tree declaration and manipulation).

Hence our system was designed to free the user from the above constraints, by allowing him to write a single EBNF specification of the language to be compiled with the addition of some annotations which will be discussed later. From this specification is produced:

- a guaranteed valid lex input specification
- routines for determining values for attributed lexemes
- a guaranteed valid yacc input specification
- data structure declarations for the abstract syntax tree
- routines for tree manipulation in memory and input/output (I/O).

5.3 Functional specification of the CORGI system

As explained above, the CORGI system was not to produce a compiler-compiler from scratch, but to provide a more convenient interface to the user, and to automatically generate lex and yacc specifications from a grammar, including routines for tree building and manipulation; thus the system can be regarded as an enhancement of these two existing tools. CORGI is designed to be a fully integrated system which, when presented with an input specification, will produce an executable parser (including routines for building, manipulating, and performing input/output of the abstract syntax tree) for the given language.

5.3.1 CORGI input specification

In order to provide a convenient interface to the user, it was decided to use EBNF as an input notation for CORGI, since this is the most common notation used in language reference manuals. Spaces, newlines and comments may occur anywhere in the specification, between any grammar symbols, terminals or nonterminal. Comments in the input specification may be nested. They are any sequence of

characters enclosed between "/*" and "*/" as in C. Thus their syntax using EBNF is:

```
Comment  = "/*"  { item } "*/".
item     = Comment | anyprint-char-but-commentstartsymbols.
```

Following the style of lex and yacc, a full CORGI specification for a language for which a compiler is being produced consists of three sections separated by "%%". These are:

```
{ Annotation }              ==> section 1
%%
{ Rules }                   ==> section 2
%%
{ User's routines }         ==> section 3
```

Each of these sections is described in detail in the following sections.

5.3.1.1 Annotation section

The annotation section describes aspects of the language which we would not normally wish to treat as part of its syntax (eg. comments) or where description is tedious using a context-free grammar notation (eg. case sensitivity of keywords).

In order to produce the necessary input for lex to perform lexical analysis, lexemes of the language are extracted by the CORGI system from the EBNF syntax description. The user is therefore not required to write regular expressions to describe these lexemes.

Certain information must however be provided in this section for some lexemes. This information is divided into four major sections namely Comments, Key-case, Lexemes and Operators. These sections are described in detail below.

a) <u>Comments</u>: Comments are an unpleasant exception in the vocabulary of most programming languages. In fact they are not part of the syntax and should be deleted by the lexical analyser. A method is needed to allow the user to specify the symbols which start and end comments. An approach taken by Mössenböck (1986) was to introduce these symbols using the keywords FROM and TO. However this does not cater for a language where a comment can be closed by more than one closing symbol (eg. Pascal).

Hence in the CORGI system we have allowed the user to give a list of alternative symbols to start and end comments, thus:

```
STARTCOMMENT          "{"  |  "(*".
ENDCOMMENT            "}"   "*)"  |  "*)"   "}".
```

would be used to describe Pascal comments. The above states that starter "{" can be matched by "}" or "*)", and that starter "(*" may be matched by "*)" or "}".

In order to deal with languages where there is no explicit closing comment symbol as such, but a comment ends after a newline is found, CORGI provides the keyword `NEWLINE` ; for example for Ada, this specification would be:

```
STARTCOMMENT          "--".
ENDCOMMENT            NEWLINE.
```

For Mesa (Mitchell, 1979), the specification would be:

```
STARTCOMMENT          "--".
ENDCOMMENT            "--"   NEWLINE.
```

Some languages permit nested comments, eg. Modula-2, and we should allow the user to specify this in CORGI. This facility cannot be described using regular expressions, and is not supported by lex, however CORGI allows such a feature using the keyword `NESTED` to indicate that the comment symbols given can be nested. For example the Modula-2 specification would be:

```
STARTCOMMENT          "(*"   NESTED.
```

```
ENDCOMMENT          "*)".
```

In chapter 6 we give details of how nested comments are dealt with in the generated lex input specification.

If this part of the annotation section is omitted, then the language is assumed not to allow comments at all.

b) Case sensitivity: The case sensitivity of keywords in a language can be difficult to describe using a normal context-free grammar notation. If the language allows keywords in lower case only, then these can be given directly in the grammar as in for example:

```
whilest = "while"  expression  "do"  statement.
```

A similar rule would be given for upper case keywords only:

```
whilest = "WHILE"  expression  "DO"  statement.
```

The specification becomes more tedious and redundant, if keywords can appear in lower case or upper case, but not allowing case to be mixed, as in:

```
whilest  =  "while"  expression  "do"  statement
         |  "WHILE"  expression  "DO"  statement.
```

The situation is even more difficult if case can be mixed, thus rendering a concise specification impossible using the above methods; it would not be reasonable to expect the user to state all combinations of upper and lower case in keywords.

For example given the CORGI rule for a while statement:

```
whilest  =  "while"  expression  "do"  statement.
```

then the annotation section may contain one of the following, where we explain their meaning:

```
KEYWORD_CASE    CASE-SIG.
```

means only "while" in lower case is allowed.

```
KEYWORD_CASE    CASE-NONSIG.
```

means any combination of upper and lower case letters is allowed.

```
KEYWORD_CASE    CASE-NOTMIXED.
```

means either `while` or `WHILE` is allowed, but a lexeme such as `WhIlE` would be taken as an identifier.

If the case sensitivity part of the annotation section is omitted then the default value taken is `CASE_NONSIG`.

c) Lexemes: As previously mentioned, CORGI extracts the lexemes of a language from its context-free grammar. As discussed in chapter 3 we define a lexeme to be a sequence of characters from the source text grouped together in a particular structure. Krzemien & Kukasiewicz (1976) developed an algorithm which extracts from the original BNF grammar all so-called "quasi-regular" subgrammars generating regular languages. However it sometimes extracts certain subgrammars which are either not normally treated by the lexical analyser (see rule 1) or for which a finite state automaton (FSA) cannot readily be built (see rule 2).

```
idr-list    =   idr  { "," idr }.                              (1)
binary-num  =  0 | 1 | 0 binary-num | 1 binary-num.            (2)
```

Rule (1) is regular and hence theoretically an FSA can be built from it, however one would not wish to use a lexical analyser on such a rule. The rule does not describe a single indivisible (atomic) unit, instead it describes a group of such units and hence to treat such a group as a single entity makes the semantic analysis very complex. Rule (2) is also regular, but since it is not given in regular expression form (it contains a direct recursive definition) a FSA cannot be built directly from it; therefore some changes have to be made such as:

```
binary-num   =  binary-digit  { binary-digit }.
```

```
binary-digit =  0 | 1.
```

Thus in CORGI, lexemes are described using EBNF notation; however this description must not contain any direct or indirect recursive definitions to avoid the problem found in rule (2). Violation of this constraint will lead to a circular definition in the generated lex specification, which will subsequently be faulted by lex.

In order to isolate the lexemes which lex should be used to detect, and which may require evaluation (eg. identifiers, strings, numeric constants, etc), the user is required to list the nonterminal symbols which describe these (usually attributed) lexemes. This is done by using the keyword LEXEME followed by a list of the appropriate nonterminals or literals. For example, in a language where the only attributed lexemes are `identifiers`, `strings` and `integers`, the LEXEME part of the annotation section would contain:

```
LEXEME   identifier  string  integer.
```

where these might be further described in the rules section as:

```
identifier   =  letter  { letter  |  digit  }.
string       =  """" { anybut_DQ_NL } """".
integer      =  digit  { digit  }.
```

Note that in some programming languages certain literals are treated in a very special way. For example, the delimiters "#" and "+" used in read and write statements or the keywords "TRUE" and "FALSE" used in MSL language. These literals are now important and needed during the code generation, unlike literals such as "!", "@" which are needed only for the context-free syntax. These literals may be treated by the lexical analyser, and therefore may be given in the LEXEME list as follows:

```
LEXEME  identifier string integer "TRUE" "FALSE" "#" "+".
```

where these might be used in the grammar section as:

```
identifier      = letter  { letter  |  digit  }.
string          =  """"  { anybut_DQ_NL }  """".
integer         =  digit  { digit  }.
Operand         = int | text | "TRUE" | "FALSE"
                | "@" idr |StoreAccess |"(" Expr ")".
Optionalhash    =  ["#"].
Optionalplus    =  ["+"].
```

This section is a mandatory part of the annotation section.

c) Operators: In many programming languages, operators have differing precedence and associativity and a common problem found in such languages is specifying expressions. The ideal is to have a formal notation to describe both operator precedence and asociativity, however such notation is usually only used in standard BNF and not in the clearer, more natural formulation preferred in EBNF. For example the syntax of a Pascal expression as given in BSI 6192 is:

```
expression       = simple-expr[ relational-oprs simple-expr ].
simple-expr      = [ sign ] term { adding-oprs term }.
term             = factor { multip-oprs factor }.
factor           = variable-access | ... | "not" factor.
relational-oprs  = "<"|"<="|">"|">="|"<>"|"="|"in".
adding-oprs      =  "+"|"-"|"or".
multip-oprs      =  "*"|"/"|"div"|"mod"|"and".
```

Grammar 1

From this set of rules only the operator precedences can be determined; we can deduce that the operator `not` has the highest precedence, followed by the `multiplying-operators`, then the `adding-operators` and `sign`, and finally with the lowest precedence, the `relational-operators`; but we can say nothing about associativity. However from the syntax of an expression given below, even this deduction is not possible.

```
expression      =   expression operators expression
                |   unary-oprs expression
                |   identifier.
operators       =   adding-oprs | logical-oprs | rela-oprs
                |   multip-oprs | exponen-oprs.
adding-oprs     =   "+"   |   "-"   |   "&".
logical-oprs    =   "and"   |   "or"   |   "xor".
rela-oprs       =   "<"   |   "<="   |   ">"   |   "="   |   "/=".
multip-oprs     =   "*"   |   "/"   |   "mod"   |   "rem".
exponen-oprs    =   "**".
```

Grammar 2

Passing grammar 1 or grammar 2 to yacc would cause a large number of shift/reduce and reduce/reduce conflicts.

It would not be reasonable, however, to expect the user to express expression rules such that operator precedence and associativity can be deduced. For this reason, CORGI provides an OPERATORS part of the annotation section, where the user lists operators of the language in ascending order of precedence, together with an indication as to whether they are left-, right- or non-associative (using the notation \L, \R, \N respectively). For example, in a language where "*" and "/" have higher precedence than "+" and "-", and all are left-associative, the OPERATORS section would be:

```
OPERATORS
operators    =    \L   { "+"    "-"    }
                  \L   { "*"    "/"    }.
```

If all operators had the same precedence, as in MSL, we would have:

```
OPERATORS
operators    =    \L   { "+"    "-"    "*"    "/"    }.
```

The nonterminal symbol `operators` can then be used on the right hand side of the expression rule in the grammar without needing to redescribe it. This removes the

requirement for the redundant information, which needs to be included to achieve the same results in other systems, notably yacc (Johnson, 1975) and Early's method (Early, 1975). For example in yacc one would give a specification such as:

```
...
%left   '+'    '-'
%left   '*'    '/'
...
%%
expression    : expression  '+' expression
              | expression  '-' expression
              | expression  '*' expression
              | expression  '/' expression
              | identifier;
```

instead of the following CORGI input:

```
OPERATORS
operators   =      \L  { "+"    "-"  }
                   \L  {  "*"    "/"   }.
%%
expression     = expression operators expression.
...
```

For completeness, the following is the syntax of the CORGI annotation section in EBNF.

```
Annotation = [Comments] [Key-case] Lexemes [Operators].
Comments   = "STARTCOMMENT" S-sym {"|"S-sym}["NESTED"] "."
              "ENDCOMMENT" E-sym{E-sym}{"|"E-sym{E-sym}} ".".
S-sym      = Literal.
E-sym      = S-sym  |  "NEWLINE".
Key-case   = "KEYWORD-CASE" ( "CASE-SIG" | "CASE-NONSIG"
             | "CASE-NOTMIXED"  )  ".".
Lexemes    = "LEXEME" (Identifier | Literal)
              { ( Identifier | Literal) } ".".
Operators  = "OPERATORS"  ProdS  ".".
ProdS      = Production ProdS | Production.
Production = Identifier "=" PrecedS ".".
PrecedS    = Preced  PrecedS | Preced.
Preced     = [ Assoc ] "{" Op-sym { Op-sym  } "}".
Assoc      = "\L" | "\R" | "\N".
```

```
Op-sym      = Literal | Identifier.
Identifier = letter { letter | digit | "_" }.
```

Figure 5.1 The syntax of the annotation section in EBNF

5.3.1.2 Rule section

The rule section is mandatory, and in it the user describes the context-free syntax of the language for which a compiler is being developed, using an EBNF notation.

As previously mentioned, descriptions of lexical constructs, such as identifiers, keywords, numeric constants, literal constants etc, are given in this section as found in a "reference manual" grammar. This contrasts with other systems such as DELTA (Lorho, 1977) and MUG1(Wilhelm *et al.*, 1976) where grammar rules are given in two separate parts:

- the first part being a set of regular expressions for a lexical analyser
- the second part being a set of BNF rules for constructing a parser.

In such systems, the user must be familiar with both regular expression and BNF notation and must be able to translate from the typical reference manual form of a language's grammar to the form required by lex (eg. RE) and yacc (eg. BNF).

Although CORGI accepts a general EBNF grammar, there are some restrictions which have to be observed. For the description of lexemes, as mentioned in section 5.3.1, the grammar should allow the construction of a FSA, containing neither directly nor indirectly recursive definitions.

A further restriction is that the grammar describing the language must be in well-formed EBNF. A well-formed EBNF grammar must follow context-free and context-sensitive rules. The context-free rules are laid down by the syntax of

CORGI given in EBNF as shown in figure 5.3; in order to comply with the context-sensitive rules, it must satisfy the constraints of an LALR(1) grammar (after allowing for augmentation by operator precedence and associativity annotations). Compliance with the context-free rules is enforced by CORGI. Where practical, context-sensitive constraints are also enforced by CORGI, but to avoid extensive duplication of the work of yacc, the fundamental checking of LALR(1) parsability must be left to yacc. Further details are given in section 6.2.

In the rule section the user gives a description of his language by means of grammar rules using EBNF notation with some additional information. This grammar is a sequence of one or more grammar rules where each rule has the following form:

```
Rule-name   =   Rule-body.
```

Rule-name is an identifier that starts with a letter and continues with either a letter, digit or underscore. Rule-body, which represents the RHS of the rule, may contain one or more alternatives which defines the form of the statement of the given language. Actions may be attached to the RHS of rules, but only for rules that describe lexemes; in the following section a detailed description of actions is given.

The associated actions

Actions in grammar rules are intended to provide a means of evaluating attributed lexemes; this will normally result in a call to an evaluation routine. For example, given the lexemes `identifier`, `string`, `character` and `integer`, one might write the following rules:

```
identifier  = letter { letter | digit }   ==>  evaluate_idr(0).
string      = """" anychar1 { anychar1 } """" ==>  evaluate_Cstr.
character   = "'" anychar2 {anychar2 } "'"    ==>  evaluate_Cchar.
integer     = digit { digit }      ==>  evaluate-denary.
```

```
anychar1      = anybut_DQ_NL.
anychar2      = anybut_SQ_NL.
```

where `evaluate_idr`, `evaluate_Cstr`, `evaluate_Cchar` and `evaluate_denary` are predefined evaluation routines. A library of evaluation routines is provided for this purpose, but the user may override this by inserting his own routines, whose correctness now becomes the user's responsibility.

Routines contained in the library fall into three categories: those that deal with the evaluation of integers, strings and characters. The routine `evaluate_idr(0)` is an exception since it does not belong to any of the three types given above. The `(0)` is used to indicate that the case in this particular lexeme is not significant; case-significance would be denoted by `(1)`. The library contains the following routines:

Routines to evaluate numbers:

1. evaluate_binary()
2. evaluate_denary()
3. evaluate_Hex()
4. evaluate_Octal()
5. evaluate_C_Hex()
7. evaluate_C_Octal()
8. evaluate_Mod2_Hex()
9. evaluate_Mod2_Octal()
10. evaluate_Ada_int(delimiter, separator)
11. evaluate_token()
11. evaluate(<integer>)

Routines to evaluate strings:

13. evaluate_Cstr()
14. evaluate_Pstr()
15. evaluate_Ostr(escapechar, specialcase, translation)

Routines to evaluate characters:

16. evaluate_Cchar()
17. evaluate_Pchar()
18. evaluate_Ochar(escapechar, specialcase, translation)

As the evaluation of real literals is generally implementation dependent, provision for this task is left to the user of CORGI. As it stands, CORGI will maintain the attribute associated with a real literal as a string, which may be evaluated by a user-provided routine.

The routines 1 to 4 deal with languages where the radix is not part of the number; it is known from the language features. These languages usually allow only a single representation of numbers; binary, denary (eg. Fortran, Cobol, Miranda, Prolog and Pascal), hexadecimal or octal (eg. Maclisp).

Routines 5 to 10 deal with languages which use a C-like, Modula2-like, or Ada-like representation respectively. In the C-like languages a number can be denoted as follows:

`31`	`2453`	- denary integers in C; `evaluate_denary()` can be used in this case.
`037`	`0265`	- octal integers in C (0 is the octal specifier); `evaluate_C_Octal()` can be used in this case.
`0X1F`	`0x1ff`	- hexadecimal integers in C (0X is the hex specifier); `evaluate_C_Hex()` can be used in this case.

The Modula2-like languages allow numbers to be suffixed with a base specifier such as the following representations:

`672C`	`146C`	- octals, type char in Modula2 (C is the octal specifier); `evaluate_Mod2_Octal()` can be used in this case.
`123B`	`675B`	- octal integers in Modula2 (B is the octal specifier); `evaluate_Mod2_Octal()` can be used in this case.

`24AH 76EH`	- hexadecimal integers in Modula2 (H is the hex specifier); `evaluate_Mod2_Hex()` can be used in this case.

The Ada-like languages include Ada, Pop-11 and Algol-68. In all the following cases `evaluate_Ada_int()` can be used. For example

`12 123_456`	- denary integers in Ada
`2#1111_1101#`	- a binary integer in Ada ('#' is the separator)
`16#FF#`	- a hexadecimal integer in Ada
`66 1234`	- denary integers in Pop-11
`2:101`	- a binary integer in Pop-11 (':' is the separator)
`8:101`	- an octal integer in Pop-11
`123 54`	- denary integers in Algol-68
`2r10001000`	- a binary integer in Algol-68 ('r' is the separator)
`4r200`	- a base 4 integer in Algol-68 (the base can be of any value up to a maximum of 36)

From the above examples, we notice that the base is given first followed by a separator (eg. `#`, `:` or `r`) and may be terminated by the separator as in Ada. Thus two parameters are needed in this case; the first parameter (1 or 0) indicates whether it is delimited or not and the second parameter gives the separator which may be one or more characters (eg. `"#"`, `":"`, `"r"` etc). Note that in this type of representation the evaluation of the base must be performed first.

`evaluate_token()` overrides the default action (which is to evaluate the lexeme to its text value); with this function the attribute value of the lexeme in question gets the value of its associated token number. `evaluate(<integer>)` provides the lexeme with an integer value; this may be used to associate 0 or 1 to literal such as FALSE or TRUE found in MSL.

For the evaluation of characters and strings, we provide `evaluate_Cchar()` or `evaluate_Cstr()` which deal with C-like languages and `evaluate_Pchar()` or `evaluate_Pstr()` for Pascal-like languages and which also cater for characters and strings where the delimiter is not (') such as Modula-2 and MSL. We also supply `evaluate_Ochar()` and `evaluate_Ostr()` that cater other cases and which require three parameters:

- An escape character if one is used, eg. `"*"` or `"'"`.
- Special characters eg. tab, newline, backspace characters etc, which must be given via a string such as `"t, n, b"`.
- A string of ASCII equivalents of the above special characters, such as `"13, 20, 21"`.

When using an EBNF notation to describe both the syntactic and lexical elements of a language, the use of white space represents a problem. Consider the following example rules for a Pascal identifier list.

```
idr-list = idr { "," idr }.                   (1) describes an identifier list
idr      = letter { letter | digit }.  (2) describes an identifier
```

Clearly in the first rule, we wish to allow spaces between identifiers in the list. However in the second rule, spaces should not be permitted within an identifier. Alternatively if the spaces are allowed then they must be explicitly manifested in the grammar rule but only in those rules which describe lexemes of the language. For example in Algol-68 identifiers are allowed to contain spaces, their specification would be:

```
idr  =  letter  { letter  | digit  |  " "  }.
```

Another problem which arises, is whether characters appearing in the lexemes are significant; spaces for example in Algol-68's identifiers are allowed but not

significant, also underscore in Ada's numbers is allowed but not significant. The allowable space in Algol-68 is usually used to give a special meaning to the usage of this particular identifier. eg. the identifier "carpet" is equivalent to the identifier "car pet", although their intuitive meanings are different. Also the allowable underscore in Ada is used only for clarity, for example the number 1000 is equivalent to 1_000. The syntax of such lexemes would be:

```
identifier = letter { letter | digit | " " }.       (1) Algol-68
number     = digit { digit | "_" }.                  (2) Ada
```

The problem with these two rules, is that we require no significance to be attached to the allowed space and the underscore in identifiers and numbers respectively. Therefore we introduce the symbol "#" which may precede factors of a rule. The presence of this symbol indicates that the following factor is not significant. Rule (1) and (2) may now be rewritten as follows:

```
identifier = letter { letter | digit | #" " }.      (1') Algol-68
number     = digit { digit | #"_" }.                 (2') Ada
```

A number of predefined rules are provided by the CORGI system for certain commonly found entities, particularly concerning character classes; these are:

```
upper_letter  = [A-Z]
lower_letter  = [a-z]
uplow_letter  = [A-Za-z]
octdigit      = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
digit         = octaldigit | 8 | 9
hexdigit      = digit | A | B | C | D | E | F
any_PR_char   = any printable ASCII character
anybut_NL     = any printable ASCII character except newline
anybut_DQ     = any printable ASCII character except (")
anybut_SQ     = any printable ASCII character except (')
anybut_DQ_NL  = any printable ASCII character except (") and
                newline
```

```
anybut_SQ_NL  = any printable ASCII character except (') and
                newline
```

For completeness the EBNF description of the rules section is given below:

```
Grammar    = Rule { Rule } ".".
Rule       = identifier "=" Expression ".".
Expression = Term { "|" Term }.
Term       = Factor { Factor } [ Action ].
Factor     = ["#"] ( Identifier | Literal )
           | "(" Expression ")" | "[" Expression "]"
           | "{" Expression "}".
Literal    = """"  Anychar { Anychar } """".
Action     = "==>" Identifier [ "(" ParamS ")" ].
ParamS     = Param { "," Param }.
Param      = Flag | Literal.
Flag       = "0" | "1".
Identifier = letter { letter | digit | "_" }.
Anychar    = any_PR_char. /* CORGI predefined */
```

Figure 5.2 The syntax of the rule section in EBNF

5.3.1.3 User routine section

The third and final section, which is optional, is the user routine section. Here the user declares any routines which he has used instead of the standard library routines for evaluating lexemes. Such routines should be coded in the C programming language. If this section is empty, then it is assumed that only the library routines are to be used for this purpose.

Figure 5.3 on page 117 gives the EBNF syntax of the complete CORGI input specification.

5.4 Overall structure and use of the CORGI system

Having reviewed the input specification we now address the overall organization of the CORGI system, and how it fits in the use of the system. CORGI is a system built on top of two existing tools, namely lex and yacc. It operates directly on "reference manual" grammars which are based on EBNF notation. The main facility this system provides is the automatic generation of lex and yacc specifications from one single input presented in EBNF.

For the generation of the lex specification, CORGI automatically establishes the relationship between keywords and identifiers of the language and uses the most appropriate method for dealing with them.

For certain types of tokens, additional information must be made available, their attribute values for example. This additional information will be held at appropriate points in the syntax tree to be used by semantic analysis and code generation routines. CORGI inserts lex actions automatically to deal with nested comments; it also supports tracing facilities via a flag to CORGI, which will cause the current lexeme and the rule which recognized it to be printed out.

For the generation of the yacc specification the system converts the EBNF notation into its BNF equivalent, suitable for input to the syntax analyser generator yacc. yacc error productions are also inserted according to an algorithm similar to the one given by Schreiner & Friedman (1980) to aid error reporting and recovery.

In addition, our system also automatically generates the necessary C data structure declarations in order to specify the organization of the abstract syntax tree. For each rule in the grammar (ie. for each nonterminal), CORGI produces a C struct, which contains fields to hold pointers to other nonterminals and the attribute values of certain terminals. Yacc actions are then inserted into the generated yacc input,

which allocate dynamic storage for nodes of the abstract syntax tree (AST) as the syntax analyser parses its input stream. Thus when syntax analysis is completed, the entire AST has been built.

To enable semantic analysis and code generation to be performed, CORGI generates a number of "walker" routines, which traverse the AST in a left-to-right depth-first manner. The user can then insert code into these routines for semantic analysis and code generation, in the knowledge that tree traversal will be bug-free. He is assisted in this task in that the names of walker routines correspond to the rule names in the generated yacc input.

Phases of the compiler following syntax analysis may be performed by a totally separate process. It might be desirable to direct the tree to backing storage to be used by separate processes, alternatively it might be that all that is required is the immediate application of semantic analysis and code generation; these two methods will be known as deferred semantic reduction and direct semantic reduction, respectively. Deferred semantic reduction is provided for organizational clarity which is achieved by system modularization; it also provides a means for debugging which is a very important stage in the software engineering life cycle. The CORGI system however allows both approaches which are described below.

5.4.1 Deferred semantic reduction

If an intermediate representation of the program is required then the CORGI system functions as shown in Figure 5.4. The CORGI grammar processor takes the user's input specification and produces a file containing a lex specification of the language's lexemes `lex.spec` and a yacc specification of its syntax `yacc.spec`. Lex and yacc are then invoked to form the lexical and syntax analyser phases of the compiler in `lex.yy.c` and `y.tab.c`. Also produced is a series of recursive writer

routines in `writer.c` for writing the AST onto permanent storage, together with the data structure definitions for the abstract syntax tree `syntree.h`. When the AST is written to a file `tree.dat`, it is first flattened into a linear representation, which is human-readable and includes information available to the compiler-writer for debugging and tracing purposes.

All these generated files are then passed to the C compiler to produce a parser which will emit a flattened tree representation of the parsed program. The intermediate representation for the tree can then be read in by the generated reader routines held in `reader.c` which reconstruct the AST into its original form, to be processed by the semantic analyser and code generator using the generated walker routines `walker.c`. User-written code is inserted at appropriate points in the walker routines. Such code will then be applied to the corresponding tree nodes when the walker routine is executed. These are then passed to the C compiler to produce the semantic analysis and code generation phase of the compiler, which take as input the flattened tree produced by the parser. Hence when parsing is complete, a call will be made to the top-level writer routine to produce the flattened tree. This call is automatically inserted into the yacc specification by CORGI. The user will then invoke the executable file `run.x` to process this tree and produce the machine code version of the parsed program.

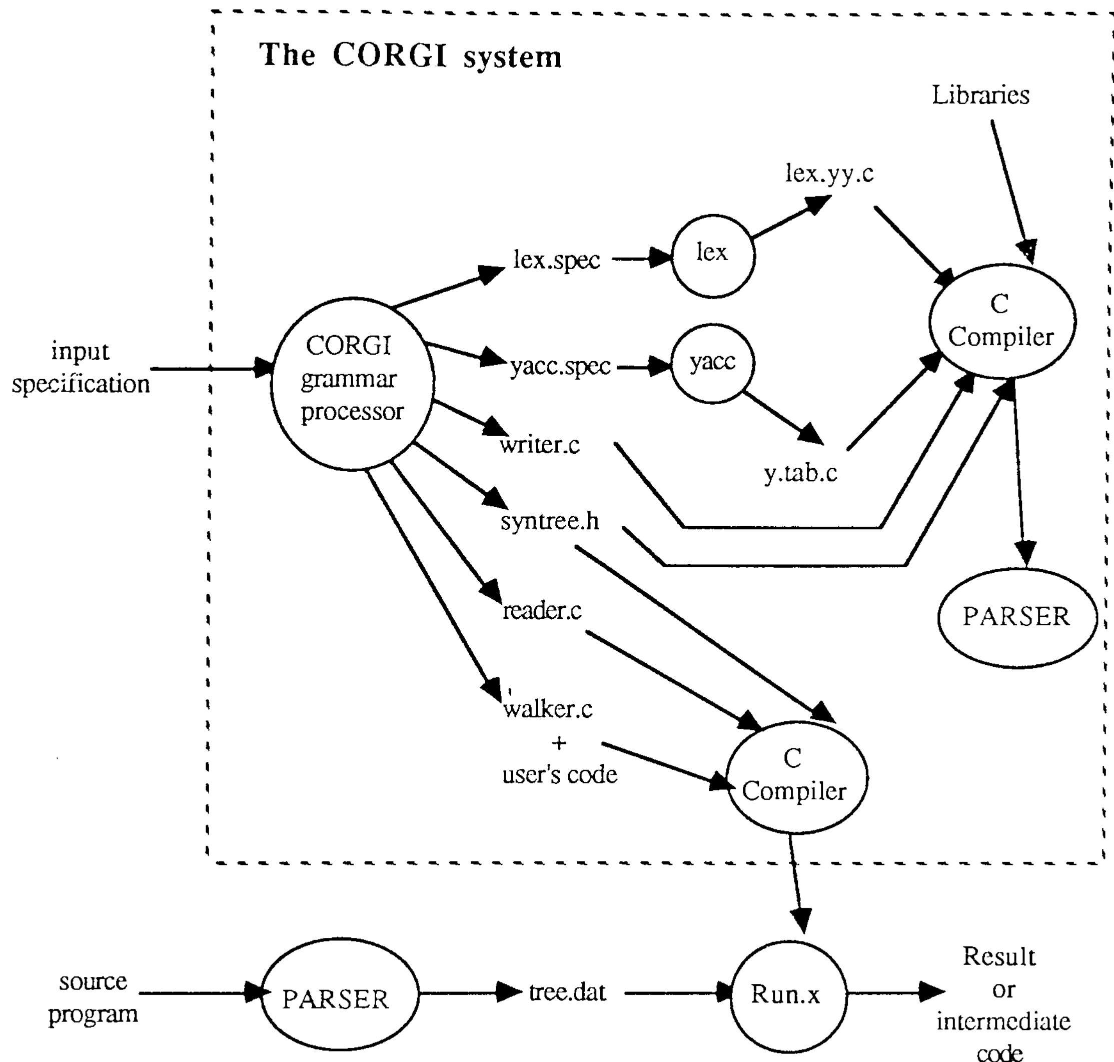


Figure 5.4 Deferred semantic reduction approach of the CORGI system

5.4.2 Direct semantic reduction

Alternatively, the user can specify via a flag that he does not wish to separate these phases. The CORGI system in this case will work as follows (see Figure 5.5). Again lex and yacc specifications are formed and passed through the lex and yacc processors to create `lex.yy.c` and `y.tab.c`. The user should then augment the generated walker routines to complete the compiler. Hence immediately after

parsing is complete, a call will be made by CORGI to the top-level walker routine, which will output the machine code representation of the parsed program. The user may still use the writer and the reader routines to generate debugging information.

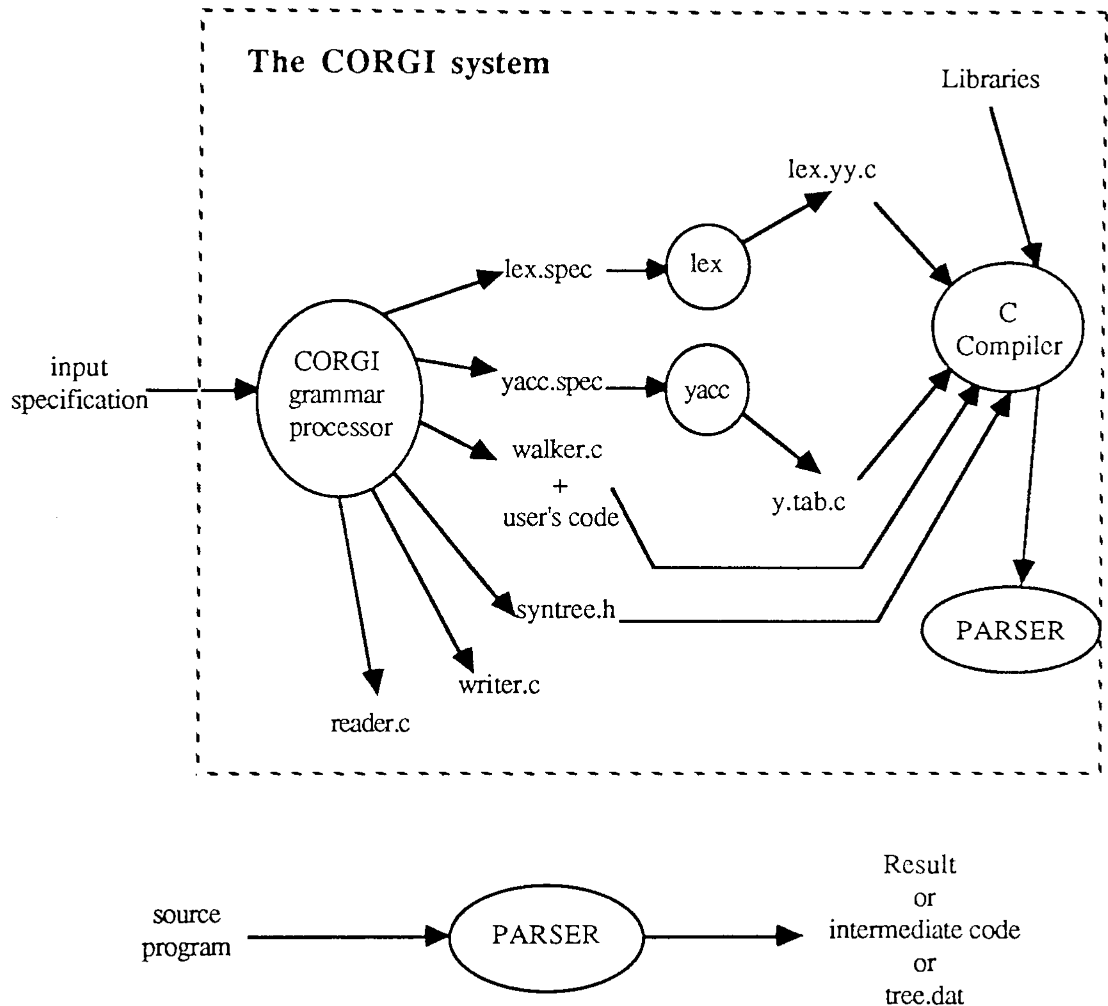


Figure 5.5 Direct semantic reduction approach of the CORGI system

In this approach `reader.c` and `writer.c` are not immediately required but are generated in case the user wishes to transfer the tree to backing storage at a later stage.

Annotation :

- `lex.spec` - lex specification.
- `yacc.spec` - yacc specification.
- `reader.c` - the reader of the tree.
- `writer.c` - the writer of the tree.
- `walker.c` - the walker of the tree.
- `syntree.h` - the data structure of the tree.
- `tree.dat` - the flattened tree (Intermediate Representation)
- `lex.yy.c` - the C program generated by lex (contains the lexical analyser).
- `y.tab.c` - the C program generated by yacc (contains the syntax analyser).
- `libraries` - include evaluation library, make nodes library, augmented yacc library.

5.5 Summary

In this chapter, we have identified the need for a compiler-writing system which uses a "reference manual" grammar as its input. We have stated that we do not intend to "re-invent the wheel" by designing such a system from scratch, but instead we propose using existing tools, lex and yacc, with an improved interface and additional features.

We have described, in broad terms, the nature of such a system, its functional specification and overall structure, operating in two possible modes: either producing an intermediate representation on permanent storage, to be processed later, or performing further processing directly.

In chapter 6, we shall describe in more detail the design and construction of the CORGI system as briefly introduced in this chapter.

In figure 5.3, we give a collected syntax for the whole of a CORGI specification.

```
InputSpec    = Annotation "%%" Grammar [ "%%" Routines ].
Annotation   = [Comments] [Key-case] Lexemes [Operators].
Comments     = "STARTCOMMENT" S-sym {"|"S-sym}["NESTED"] "."
               "ENDCOMMENT" E-sym{E-sym}{"|"E-sym{E-sym}} ".".
S-sym        = Literal.
E-sym        = S-sym  |  "NEWLINE".
Key-case     = "KEYWORD-CASE" ( "CASE-SIG" | "CASE-NONSIG"
               | "CASE-NOTMIXED"  )  "."
Lexemes      = "LEXEME" ( Identifier | Literal )
                     { ( Identifier | Literal ) } ".".
Operators    = "OPERATORS"  ProdS  ".".
ProdS        = Production ProdS | Production.
Production   = Identifier "=" PrecedS ".".
PrecedS      = Preced  PrecedS | Preced.
Preced       = [ Assoc ] "{" Op-sym { Op-sym  } "}".
Assoc        = "\L" | "\R" | "\N".
Op-sym       = Literal | Identifier.
Grammar      = Rule { Rule } ".".
Rule         = Identifier "=" Expression".".
Expression   = Term { "|" Term  }.
Term         = Factor {Factor } [ Action ].
Factor       = ["#"] ( Identifier |  Literal )
               |"(" Expression ")" | "[" Expression "]"
               | "{" Expression "}".
Literal      = """" Anychar {Anychar} """"./* CORGI predefined */
Action       = "==>" Identifier [ "(" ParamS ")" ].
ParamS       = Param { "," Param }.
Param        = Flag | Literal.
Flag         = "0" | "1".
Identifier   = letter { letter | digit | "_" }.
Routines     = { Anychar }.
Anychar      = any_PR_char.
```

Figure 5.3 The EBNF syntax of a CORGI specification

Chapter 6

The Realisation of the CORGI System

6.1 Introduction

We saw in the previous chapter a need for a compiler-compiler which works from a "reference manual" grammar input. In this chapter we shall describe in more detail the design and construction of the CORGI system.

CORGI's structure is similar to that of a traditional compiler: it has a front-end and a back-end. The front-end (or grammar processor) builds an abstract syntax tree of the input specification, to be used by the back-end. The back-end takes this tree and uses it to generate a number of files containing the following:

- lex and yacc specifications
- declaration of the necessary data structures for the construction of the AST for any particular program in the given language
- reader/writer/walker routines for tree manipulation.

The two phases of the CORGI system are described in detail in the following sections, and Figure 6.1 shows a diagrammatical overview of their interaction.

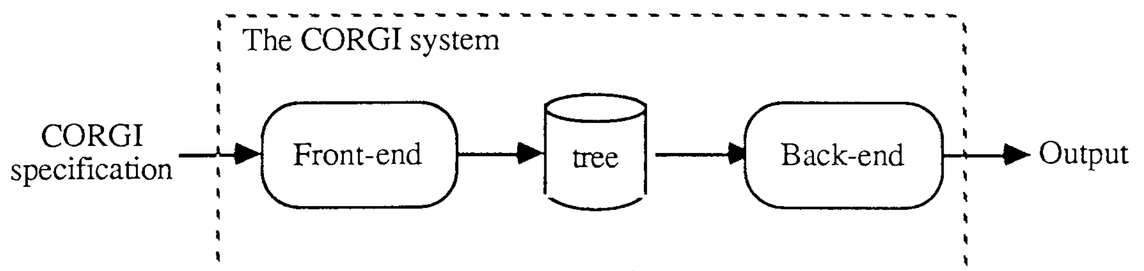


Figure 6.1 CORGI overview

6.2 The front-end

In the front-end, the text of a CORGI specification is parsed, following the syntax rules for such a specification, and its corresponding abstract syntax tree is constructed. Figure 5.3 gives an EBNF syntactic description of the CORGI input; this description is self-describing unlike the grammar of yacc which is LALR(2), as opposed to LALR(1) which is required for processing by yacc. The syntax of the language accepted by CORGI, given in the form required for processing by CORGI is given in appendix F. The front-end is itself composed of two phases: one performing lexical analysis and the other performing syntax analysis. The syntax analyser repeatedly requests single tokens from the lexical analyser. We chose to implement these two phases using hand-coded routines; syntax analysis is carried out in a top-down, recursive descent manner.

During this phase CORGI performs several checks on the input specification. It ensures that the context-free syntax is fully conformant with that of the CORGI EBNF as discussed in section 5.3.2.1.

Context-sensitive checking consists of ensuring that the grammar is LALR(1), and that all the nonterminals are uniquely defined. A non LALR(1) grammar will result in a parser generating shift/reduce or reduce/reduce conflicts; these are discussed in section 4.3.1. Since context-sensitive checking is performed by yacc, we believe that it would be redundant for CORGI to enforce strict LALR(1) constraints. However, CORGI does check that all nonterminals are uniquely defined, thus leading to any duplicate or missing definitions being reported before yacc is invoked.

During this phase, a number of data structures are built and maintained, the principal ones being:

- a lexeme list that holds the lexeme names given by the user in the LEXEME section
- an operator table which holds each operator given in the OPERATOR section together with its precedence and associativity
- an abstract syntax tree representing the structure of the rules section from the CORGI specification
- a symbol table holding the nonterminal symbols used in the input specification
- a keyword table for the given language
- a table of delimiters for the given language (eg, "!", ",", "@").

Building the abstract syntax tree

As previously mentioned, CORGI builds an abstract syntax tree of the rule section of the input specification (in a similar manner to a compiler). We shall now examine precisely how each rule from this section is represented in the tree. For clarity and convenience we have reproduced below the EBNF syntax of the rule section given in figure 5.2.

```
Grammar    = Rule { Rule } ".".
Rule       = Identifier "=" Expression ".".
Expression = Term { "|" Term }.
Term       = Factor { Factor } [ Action ].
Factor     = ["#"] ( Identifier | Literal )
           | "(" Expression ")" | "[" Expression "]"
           | "{" Expression "}".
Literal    = """" Anychar { Anychar } """".
Action     = "==>" Identifier [ "(" ParamS ")" ].
ParamS     = Param { "," Param }.
Param      = Flag | Literal.
Flag       = "0" | "1".
Identifier = letter { letter | digit | "_" }.
```

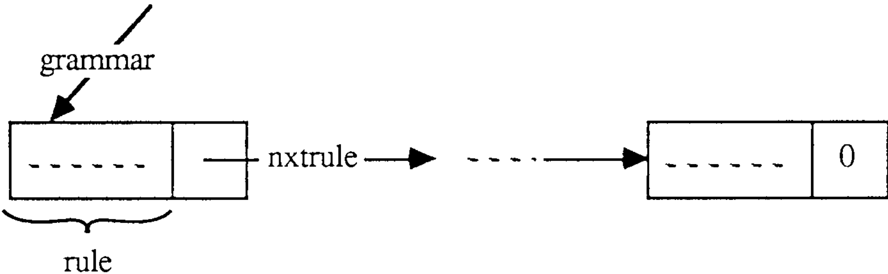
```
Anychar      = any_PR_char.
```

The syntax of the rule section in EBNF

The entire input grammar is held in a linked list where each node holds details of a single rule, eg.

```
Grammar      = Rule { Rule } ".".
```

is stored as:

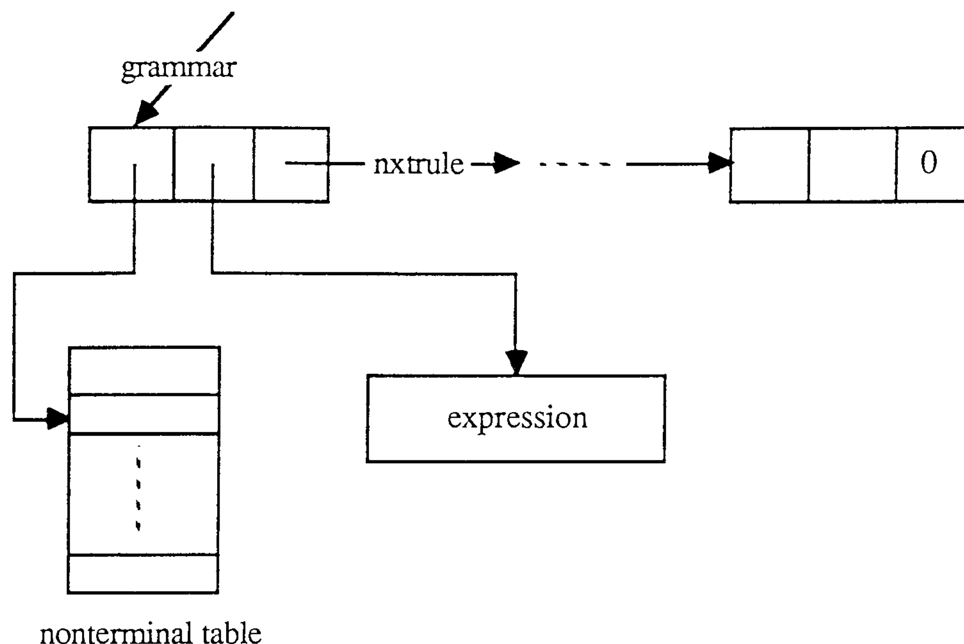


Thus in addition to its contents, a rule has a pointer field to the next rule in the grammar (nxtrule), with a null pointer ('0') denoting the end of the list.

Each rule of the grammar has the following format:

```
Rule = identifier "=" Expression ".".
```

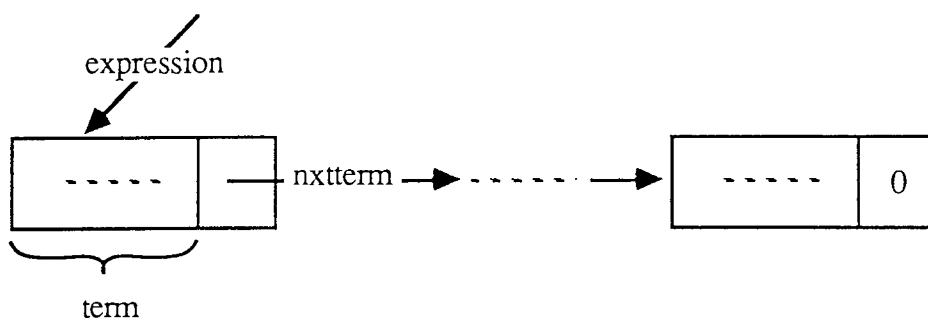
and is thus held as



where `identifier` is a pointer to a symbol table entry for the relevant identifier, and "expression" is a pointer to the rule's right-hand-side (RHS). The RHS, classed as an expression, has the following syntax:

```
Expression  = Term { "|" Term  }.
```

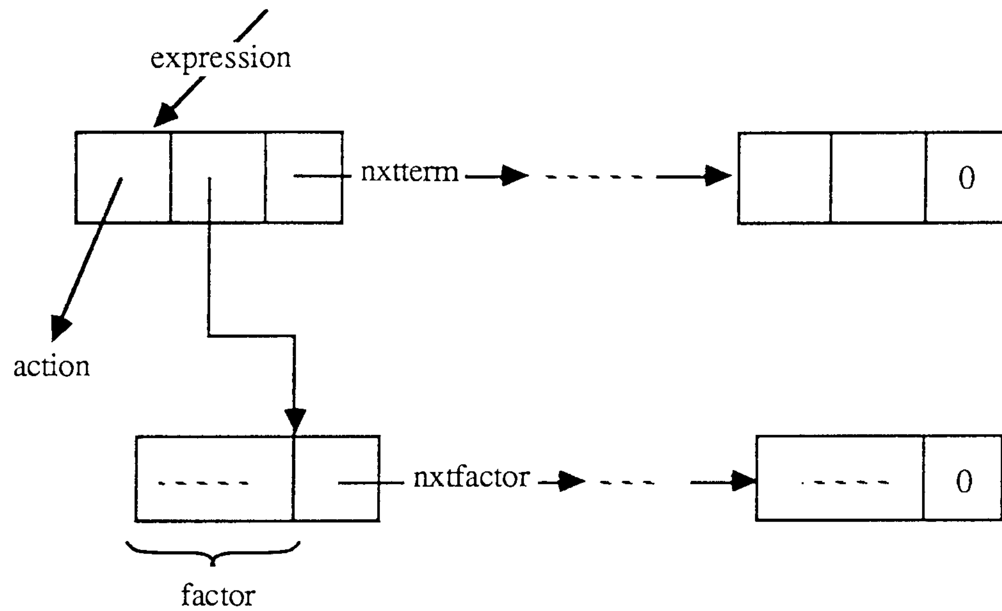
and is held in a data structure as:



where terms are held in a linked list. Each node in the linked list of terms is defined by:

```
Term        = Factor { Factor } [ Action ].
```

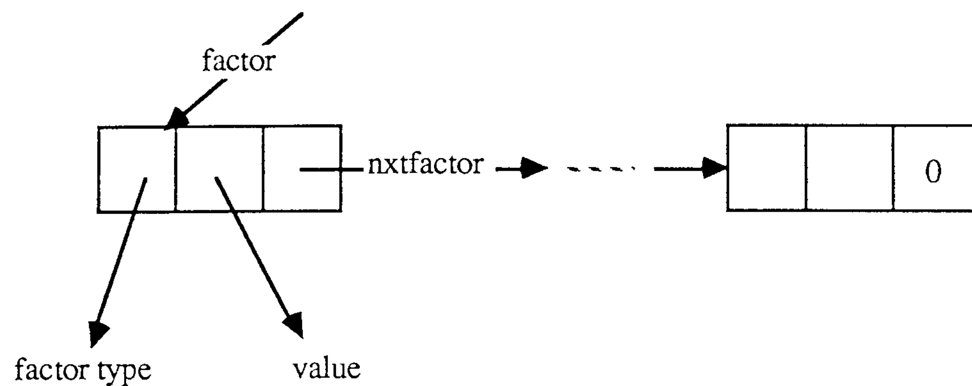
and is thus held as:



The "action" field of a term is simply a pointer to a string of characters representing a call to a C function to be performed when that term is matched in the parsed input. `factor` points to a further linked list of factors, each of which is defined as:

```
Factor      = ["#"] ( Identifier | Literal )
            | "(" Expression ")" | "[" Expression "]"
            | "{" Expression "}".
```

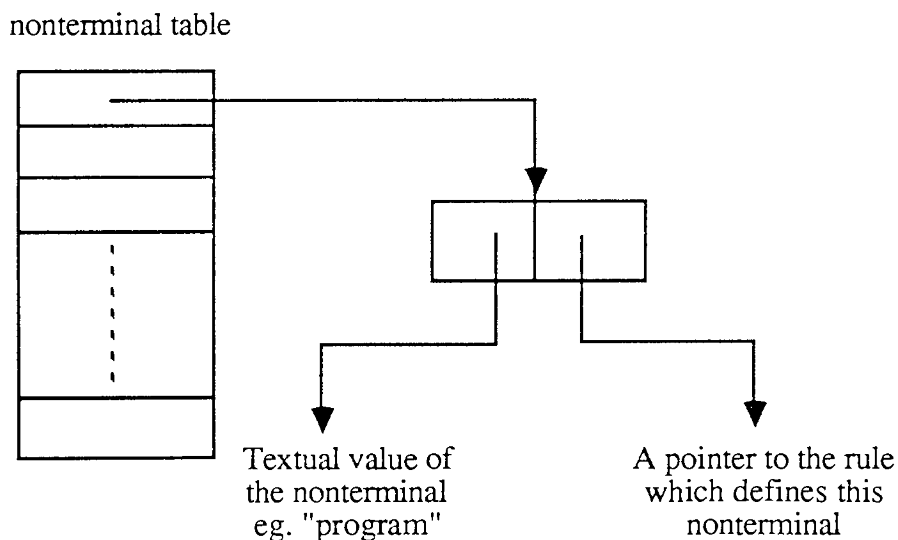
A factor is stored in the following structure:



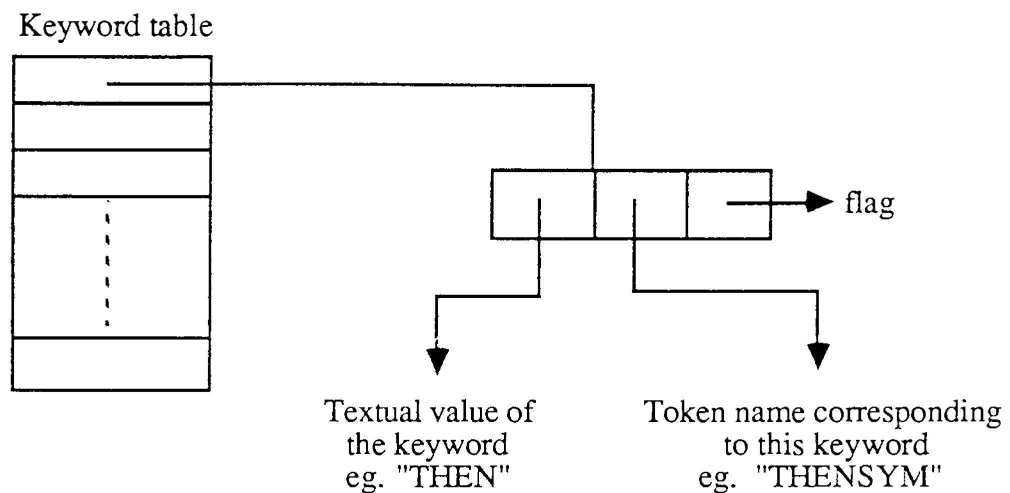
A factor can be one of a variety of types namely identifier, literal, repeated, optional or grouped expressions. An identifier is further categorized as being either a lexeme name or a nonterminal name in which case the value of the factor is a pointer to the symbol table entry for that identifier. Literals can also be further categorized as keywords, operators or delimiters, where the factor's value is a pointer to a character string. A repeated, optional or grouped expression has a value being a pointer to an expression discussed earlier.

Each of the tables which are produced from CORGI input and which are used to generated lex and yacc specifications is held as an ordered linear array. This ordering enables us to search these tables using a binary chop algorithm.

The nonterminal table has the form:



The keyword table being:

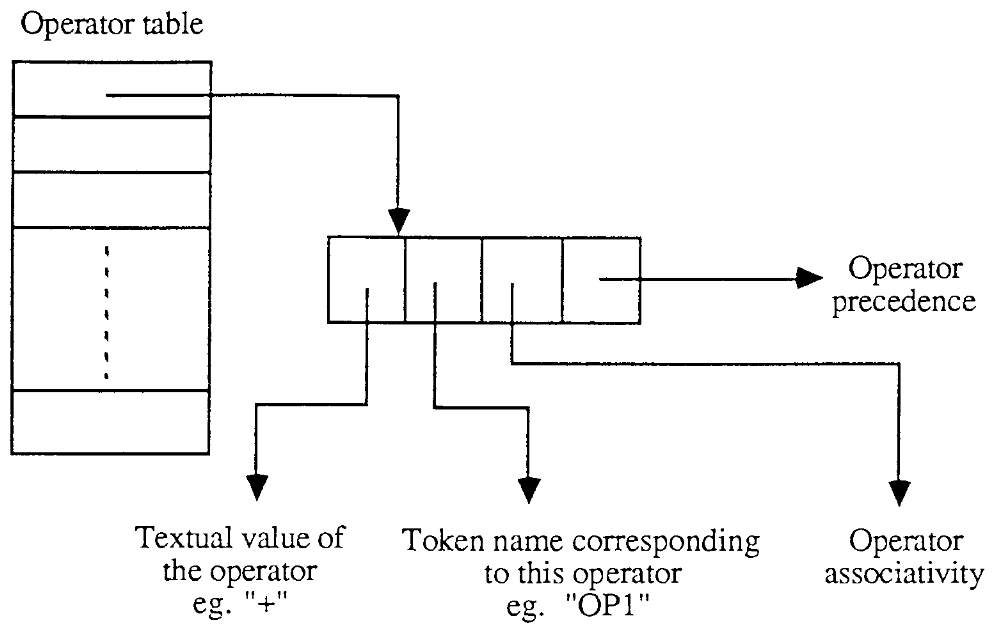


where flag is an integer that takes value 0 or 1, it indicates whether this keyword is declared in the annotation section or not.

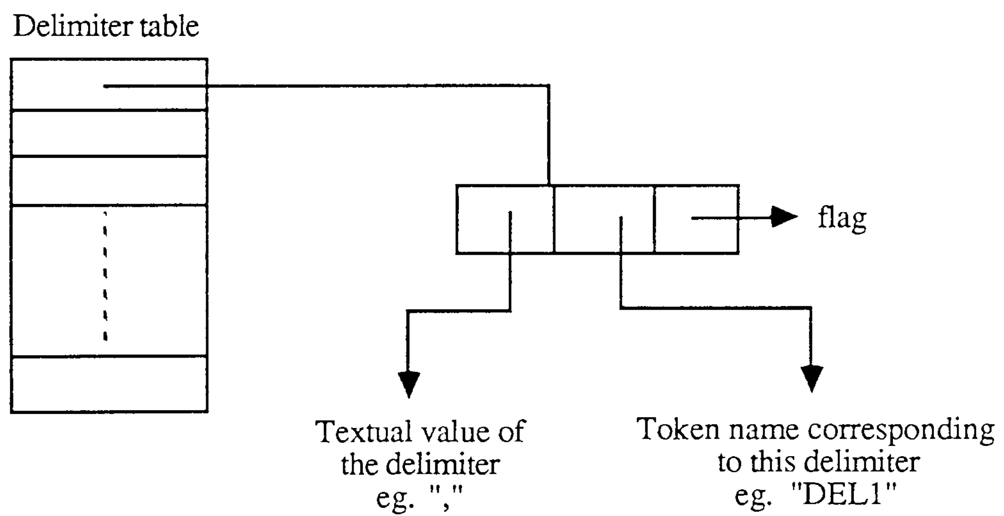
Any literal given in the CORGI specification that starts with a letter is considered to be a keyword and is hence stored in the keyword table. Following the convention of most programming languages, keywords are assumed to start with a letter, and this can be used to distinguish them from other literals (eg. delimiters). However, if in some PL this assumption is not met, and the keywords do start with a character other than a letter, the system will still function correctly but with reduced efficiency since the keywords will be listed in the lex specification which may increase the execution time of the lexical analyser.

Operators such as "MOD", "DIV" in Pascal or Modula-2 are also considered as keywords. This assumption is made in order to deal correctly with their case sensitivity.

The operator table is held as:



and finally the delimiter table:



where flag is an integer that takes value 0 or 1, indicating whether this delimiter is declared in the LEXEME list or not.

6.3 The back-end

The purpose of the back-end of CORGI is to take the tree generated by the front-end, and to use it to construct the necessary input to the lex and yacc tools, together with data structure definitions and manipulation routines. In the following sections, we shall describe how this is achieved.

6.3.1 Generating the lex specification

In order to generate a lex specification which is guaranteed to be valid for the given language, we need to extract the information necessary for each of lex's input sections, ie, the:

- definition section
- rule section
- user routine section

6.3.1.1 The lex definition section

Since the lex definition section requires details of character classes and of certain intermediate regular expressions (REs) used to define the lexemes, these must be extracted from the EBNF grammars, and then an EBNF-to-RE translation must be performed. When this extraction is complete, the rules corresponding to lexemes must be removed from the grammar to avoid them being passed on to yacc.

The first step in this process is to extract the definitions of lexemes which are listed in the annotation section using the keyword `LEXEME`, and which are held in the data structure `lexeme` built by the front-end as described earlier. Thus the definition for such a rule is now pointed to by its corresponding node in this linked list, and the rule is removed from the grammar. The rules for these lexemes may refer to other rules, in which case these rules are also added to the `lexeme` data structure, and their rules are discarded. If any non-lexeme rule in the grammar refers to one of

these discarded rules, CORGI arranges that the apparently discarded symbols, which it will always be possible to treat as lexemes, are returned as a new tokens. For example, suppose we have the following rules as given in BS6192 (1982) for Pascal:

```
simple-expression   = [ sign ] term { adding-operator term }.
signed-number       = signed-integer  |  signed-real.
signed-integer      = [ sign ] unsigned-integer.
signed-real         = [ sign ] unsigned-real.
sign                = "+"  |  "-".
```

and in the annotation section we have:

```
LEXEME      ... signed-number ...
```

In this case the rule `sign` will be extracted by our algorithm, since it is indirectly part of the lexeme `signed-number` definition; however, it is also part of the nonterminal definition `simple-expression`, so CORGI will generate the following lex input fragment:

```
signed-number       ({signed-integer}|{signed-real})
signed-integer      ({sign})?{unsigned-integer}
signed-real         ({sign})?{unsigned-real}
unsigned-integer    ({digit})({digit})*
sign                ("+"|"-")
%%
...
{signed-number}     { return (SIGNED-NUMBER); }
{sign}              { return (SIGN); }
...
```

Lex's "longest match" rule ensures that the required behaviour is obtained.

When the lexemes have thus been extracted from the grammar, then the following transformations are performed to convert their form from EBNF to lex's regular expression notation:

- The definition of an item is surrounded by "{" and "}"; for example a nonterminal in EBNF `signed-number` becomes `{signed-number}` in the generated lex specification.
- An optional item eg. `[sign]` in EBNF becomes `({sign})?`
- A repeated item eg. `digit{digit}` in EBNF becomes `({digit})({digit})*`
- Grouped items eg. `(signed-integer|signed-real)` in EBNF becomes `({signed-integer}|{signed-real})`.

For example, consider the following EBNF rules:

```
identifier    = letter { letter | digit }.
integer       = digit  { digit }.
string        = """"   { stringchar }   """".
letter        = uplow_letter.
stringchar    = anybut_DQ_NL.
```

After translation using the above rules, the definition section of the generated lex specification would contain:

```
stringchar      {anybut_DQ_NL}
letter          {uplow_letter}
string          "\""({stringchar})*"\""
integer         ({digit})({digit})*
identifier      ({letter})(({letter})|({digit}))*
```

where `anybut_DQ_NL`, `uplow_letter` are predefined by the CORGI system and their definition would appear in the generated lex specification as:

```
anybut_DQ_NL    [\n\"]
uplow_letter    [a-zA-Z]
digit           [0-9]
```

6.3.1.2 The lex rule section

CORGI generates the necessary rules in lex's rule section to deal with operators, delimiters, lexemes, keywords and comments; we discuss these separately below.

Operators: For each operator from the operator table, CORGI generates a rule of the form:

```
"operator"          { return ( OP-NUMBER);}
```

For example, if the language contains the operators "*", "<>" and "<=", then the following will be generated:

```
"*"                 { return (OP1); }
"<>"                { return (OP2); }
"<="                { return (OP3); }
```

where `OP1, OP2, OP3` denote the token numbers for `"*", "<>"` and `"<="` respectively. It would be clearer if we were able to write:

```
"*"                 { return ('*');  }
"<>"                { return ('<>'); }
"<="                { return ('<='); }
```

however, the C function is only able to return a single ASCII value, thus the single quotes may only contain one character or a 3-digit octal number. Hence we are obliged to generate the more obscure code shown above using `OP1, OP2` and `OP3`. To aid readability, CORGI includes a comment next to each token declaration in the yacc definition section, eg:

```
%token   OP1   /*  OP1 = "*"    */
%token   OP2   /*  OP2 = "<>"   */
%token   OP3   /*  OP3 = "<="   */
```

Delimiters: For each delimiter in the delimiter table, a rule of the following form is generated:

```
"delimiter"                { return ( DEL-NUMBER);}
```

For example, given the delimiters ",", "@" and "!" then the following will be generated:

```
","                { return (DEL1); }
"@"                { return (DEL2); }
"!"                { return (DEL3); }
```

where `DEL1, DEL2, DEL3` are the token numbers for `","`, `"@"` and `"!"` respectively, and are commented in a manner similar to that of the operators as described above.

Lexemes: For each lexeme in the lexeme list, we generate a rule similar to the following:

```
{lexeme}                { return (LEXEME-NUMBER); }
```

For example, if we have lexemes `identifier`, `string` and `integer`; then a possible series of rules would be:

```
{identifier}            { return (IDENTIFIER); }
{string}                { return (STRING); }
{integer}               { return (INTEGER); }
```

Example 6.1

As discussed earlier, if a new lexeme is extracted from the grammar then a rule is also generated for that new lexeme of the form:

```
{newlexeme}             { return (NEWLEXEME); }
```

For example if `sign` is a new lexeme, then it will appear as:

```
{sign}                  { return (SIGN); }
```

However this solution is not able to accommodate the keywords which are commonly treated by the `identifier` RE for efficiency reasons. In section 4.1 we identified three possible cases which are commonly found in most programming languages, these are the following:

i) $I \cap K = \emptyset$ - the keywords and identifiers are two disjoint sets
ii) $I \supseteq K$ - the keywords are a subset of the identifier set
iii) $I \cup K \neq I$ - the keywords and identifiers are two overlapped sets

where I is the set of identifiers and K is the set of keywords.

In order to establish the exact relationship of two regular sets I (for identifier) and K (for keywords), the following method is used. If S denotes a string, we find the set of all strings K = $\{S_i \mid i=1..N$ where N is the number of keywords$\}$ and then for each S_i, test if it can be a member of the set I. Let the number of strings which satisfy this condition be n, then :

1) $n = N \Rightarrow I \supseteq K$ (ie. K is a subset of I).

For this case the specification shown in Example 6.1 may be given as follows:

```
...                 /*  definition section  */
%%
...
{identifier}        { return (screen()); }
{string}            { return (STRING); }
{integer}           { return (INTEGER); }
...
```

Example 6.2 Keywords and identifiers as disjoint sets

The function `screen()` is a routine that checks the input which has been recognized against a keyword table and returns a code indicating whether it was a keyword (and if so, which keyword) or identifier.

2) $n = 0 \Rightarrow I \cap K = \emptyset$ (ie. I and K are two disjoint sets).

For this case the specification shown in Example 6.1 may be given as follows:

```
...                 /* definition section  */
kwd                 ({up_letter})(({up_letter})|({digit}))*
...
%%
{kwd}               { return(look_up());}
{identifier}        { return (IDENTIFIER); }
{string}            { return (STRING); }
{integer}           { return (INTEGER); }
```

Example 6.3 Keywords as a subset of identifiers

3) $n < N \Rightarrow I \cap K \neq \emptyset$ (ie. I and K are two joint sets).

For this case the specification shown in Example 6.1 may be given as follows:

```
...                 /* definition section  */
kwd                 ({up_letter})(({up_letter})|({digit}))*
...
%%
{kwd}               { val = look_up();
                      if(val != 0)
                          return (val);
                      else
                          REJECT;   }
{identifier}        { return (IDENTIFIER);}
{string}            { return (STRING); }
{integer}           { return (INTEGER); }
```

Example 6.4 Keywords and identifiers as joint sets

The function `look_up()` is a routine that checks whether the input which has just been recognized by the keyword rule, is indeed a keyword (which may not be the case if keywords and identifiers are joint sets). If the input is in fact a keyword then the `look_up()` function returns the token number associated to that particular keyword, otherwise it returns a zero in which case the input is passed to the next rule using the lex facility REJECT.

In order to implement this solution, we need to determine which rules of the lexemes match each of the keywords from the keyword table. We found that the best way of implementing such a scanning problem is to generate a further lex specification. This specification will only contain rules concerning the lexemes, where each rule returns a integer value starting from 1 as follows:

```
...                 /* definition section */
%%
{identifier}        { return (1); }
{string}            { return (2); }
{integer}           { return (3); }
...
```

CORGI then runs lex to generate a scanner for the keywords, and calls `yylex()` for each of the keywords from the keyword table. A record is kept of how many keywords are recognized by each rule (this corresponds to the use of `n` in the discussion of the three cases above) and is used to distinguish between the three possible cases mentioned earlier.

Note that in Example 6.3 and Example 6.4, a method is required for generating a regular expression `{kwd}` for the keywords which form a finite set. CORGI uses the following algorithm:

- As already mentioned in chapter 5, any literal which starts with a letter is taken as a keyword and is hence stored in the keyword table. Having noted that keywords start with a letter, we then consider subsequent characters in a keyword to fall into one of the classes `letter` or `digit`, or to form a single character class of their own. Such characters are termed "continuation characters".
- for each keyword in the table:
  - for each continuation character in the keyword:

- ••• if its character class is not already in continuation character list, add it to the list.

Consider the following examples of languages with different types of keywords:

```
IF   THEN   ELSE   WHILE              RE = letter(letter)*
IF12   THEN4   ELSE   WHILE           RE = letter(letter|digit)*
```

Comments: In order to deal with comments in a thorough manner, we must consider two possible complicating factors:

- can comments be nested?
- for each symbol starting a comment, is there a unique symbol that may close that comment?

Comments are dealt with in CORGI by inserting a purpose-built function `comment()` into the lex specification. When a comment start symbol is recognised this function repeatedly calls `yylex()`, consuming the input text until a valid end-comment symbol is returned by `yylex()`; fortunately lex is written in such a way that recursive calls of this form are acceptable. `comment()` ensures that each opened comment is closed by a valid symbol if there is more than one possibility. This mechanism would deal with any arbitrary nested comments.

For example, Pascal comments can be specified in CORGI as:

```
STARTCOMMENT        "(*"   |   "{".
ENDCOMMENT          "*)"   "}"   |   "}"   "*)".
```

for which CORGI will generate the following lex rules:

```
"(*"|"{"      {  if(NESTED || count != 0)
                 {  count++;
                    if (comment()!= 0)
                    {
```

```
                    printf("Premature EOF\n");
                    exit;
                }
                count--;
            }
        }
"*)"|"}"    {   return(ENDCOMMENT); }
.           {   if(count == 0)
            printf("!!! char <%c> is illegal here\n",yytext[0]); }
```

As can be noted from the above lex rules, if a symbol which may start a comment is encountered in the input stream (eg. "(*" or "{") the generated lexical analyser makes a call to the function `comment()` which will return 0 if a comment has been successfully found, and 1 otherwise. If in a language the end of line character ends a comment (eg. MSL, Ada) then the highlighted code in the following lex rule is added.

```
[\n]        {   linenumber++;
                if (count != 0) return(NEWLINE); }
```

The final rule included in the lex rule section is intended to intercept the case where the input does not match any of the supplied rules, indicating a lexical error. This rule is also used to consume any character from the input, not found by the previous rules, inside comments as shown above. This rule has the following form:

```
.           {   if(count == 0)
            printf("!!! char <%c> is illegal here\n",yytext[0]); }
```

6.3.1.3 The lex user routine section

The user routine section is generated with the following contents:

- declaration and initialisation of the keyword table
- definition of a `look_up()` function, if the keywords and identifiers are joint or disjoint sets.
- definition of a `screen()` function, if the keywords are a subset of identifiers

- any routine which the user has placed in CORGI's user routine section.

Note: For operators and attributed lexemes, additional actions must be generated. These actions deal with maintaining information required for semantic analysis and code generation. They usually involve saving values (which may need processing first, eg. evaluation of strings, numbers etc) in the AST. This additional task is explained in more detail in section 6.3.4, where the abstract syntax tree is studied in more detail.

6.3.2 Generating the yacc specification

The generation of a yacc specification which is guaranteed to be valid for the given language (provided that the user supplied grammar is well-formed and LALR(1)) involves extracting information from the user-supplied grammar for each of the yacc specification sections, ie. the:

- definition section
- rule section
- user routine section

6.3.2.1 The yacc definition section

The yacc definition section should include the declaration of tokens used in the yacc grammar. This applies to all the tokens which were referred to in the lex specification, namely delimiters, operators, lexemes and keywords. For each of these token types, CORGI generates a yacc specification as follows:

Given delimiters, say `","`, `"@"` and `"!"`:

```
%token      DEL1        /*  DEL1 = ","    */
%token      DEL2        /*  DEL2 = "@"    */
%token      DEL3        /*  DEL3 = "!"    */
```

Given operators, say `"*"`, `"<>"` and `"<="`:

```
%token    OP1        /*  OP1 = "*"    */
%token    OP2        /*  OP2 = "<>"   */
%token    OP3        /*  OP3 = "<="   */
```

Given lexemes, say `identifier`, `string` and `integer`:

```
%token    IDENTIFIER
%token    STRING
%token    INTEGER
```

Given keywords, say `"IF"`, `"THEN"` and `"ELSE"`:

```
%token    IFSYM
%token    THENSYM
%token    ELSESYM
```

Also CORGI must generate the appropriate yacc statements to attach associativity and precedence to operators. For example in the CORGI specification of MSL, we may have:

```
OPERATOR
operators = \L { "*" "<>" "<=" ... }
```

which will be transformed by CORGI into:

```
%left  OP1  OP2  OP3
```

6.3.2.2 The yacc rule section

Recall that CORGI allows the user to express his rules in EBNF, but yacc requires its rule section to be written in BNF. Hence we need a means of translating EBNF to yacc BNF, and to do this we use the following algorithm:

Optional items: For each rule of the form $A = \alpha\ [\ X_1\ \ X_2\ \ \ldots\ldots\ \ X_n]\ \beta$.

- create a new nonterminal N.

- replace A by $A = \alpha N \beta;$
- create a new rule $N = \varepsilon \mid X_1\ X_2\ \ldots\ldots\ X_n;$

where ε denotes an empty rule

Repeated items: For each rule of the form $A = \alpha \{ X_1\ X_2 \ldots\ldots\ X_n \} \beta.$

- create a new nonterminal M.
- replace A by $A = \alpha\ M\ \beta;$
- create a new rule $M = \varepsilon \mid M\ X_1\ X_2\ \ldots\ldots\ X_n;$

where ε denotes an empty rule

Note that we use left recursion in the yacc specification since this is preferred by the LR parser generated by yacc to avoid any risk of possible parse stack overflow.

Grouped items: For each rule of the form $A = \alpha\ (X_1 \mid X_2 \mid \ldots\ldots \mid X_n)\ \beta.$

- create a new nonterminal Y.
- replace A by $A = \alpha\ Y\ \beta;$
- create a new rule $Y = X_1 \mid X_2 \mid \ldots\ldots \mid X_n;$

According to the above algorithms the following two MSL rules:

```
Program = ["RESERVE"  int ] { ProcDec } Series "."
ProcDec = "PROC"  idr  [ "(" Idrlist ")" ] Series  "END".
```

are in effect transformed into the following set of rules:

```
Program       = Op-reserve ProcDecs Series ".";
Op-reserve    = /*  empty alternative  */
              | "RESERVE"  int;
ProcDecs      = /* empty alternative  */
              | ProcDecs ProcDec;
ProcDec       = "PROC" idr Op-Idrlist Series  "END";
Op-Idrlist    = /*  empty alternative  */
```

```
                    | "(" Idrlist ")";
```

The full yacc BNF grammar for MSL is given in appendix B.

Further changes to the specification must be made in order to reduce the number of shift/reduce conflicts which occur when dealing with expressions and operators. Shift/reduce conflicts arise when a parser has to decide between two legal actions: a shift, in which case the parser accepts the next lookahead symbol and shifts to the next state; or a reduction, where the parser replaces the right hand side of a grammar rule by the left hand side. For example if we have the following rules:

```
Expression     :   Expression  adding-ops Expression
               |   Expression  multip-ops Expression
               |   Expression  relati-ops Expression
               ...

adding-ops     :  "+"   |  "-".
multip-ops     :  "*"   |  "/".
relati-ops     :  "<="  | "<>".
```

then we can remove shift/reduce conflicts by first expanding the occurrences of `adding-ops`, `multip-ops` and `relati-ops` appearing in the rule for expression, then giving the operator precedence and associativity in the definition section of the yacc specification. This transformation would result in the following:

```
%left  ADDOP  MINUSOP MULTOP DIVOP LEQOP NOTEQOP
...
%%
Expression     :   Expression  ADDOP   Expression
               |   Expression  MINUSOP Expression
               |   Expression  MULTOP  Expression
               |   Expression  DIVOP   Expression
               |   Expression  LEQOP   Expression
               |   Expression  NOTEQOP Expression
               |   identifier;
```

6.3.2.3 The yacc user routine section

All that is required in the yacc user routine section is a main function whose sole purpose is to make the initial call to yacc's parser function `yyparse()`, and to

inform the user of the return status from this function. Hence CORGI inserts the following code into the user routine section :

```
%%
main()
{
#ifdef LEXDEBUG
  debug = 1;    /* debug = 1 for debugging and 0 otherwise */
#endif
extern  int yynerrs;
int flag = 0;

   flag = yyparse();
   printf("Compilation error(s): %d\n", yynerrs);
   if(flag)
       printf("Compilation aborted\n");
   else
       printf("Compilation terminated\n");
}
```

`yyparse()` returns zero if the submitted source program does not contain syntax errors, and returns one otherwise.

6.3.3 The abstract syntax tree declaration

Normally when using lex and yacc, the compiler-writer has to add C declarations of the data structures needed to construct an abstract syntax tree during parsing, and routines to manipulate this tree. We consider that this process should be automated since it is largely mechanical. These are two possible methods for achieving this: either to generate the necessary input specification for a tool such as IDL (Lamb, 1987), which translates high-level descriptions into data type definitions in C, or to generate these abstract data types directly within CORGI.

In order to evaluate the use of a tool for the automatic construction of the AST, we shall review briefly the operations of the IDL system.

6.3.3.1 The IDL system

IDL is a tool designed for supporting the construction of medium to large software systems. It provides the user with a facility for specifying data structures at a higher level of abstraction than most conventional programming languages.

The IDL translator takes a user-specified collection of data structures expressed in terms which relate closely to the problem to be solved, and transforms them into concrete declarations in a target language and routines for input/output of instances of these data structures. Instances of these data structures are stored on external storage devices in an auxiliary description language known as the ASCII External Representation Language (ERL). Figure 6.2 shows an IDL model of a process.

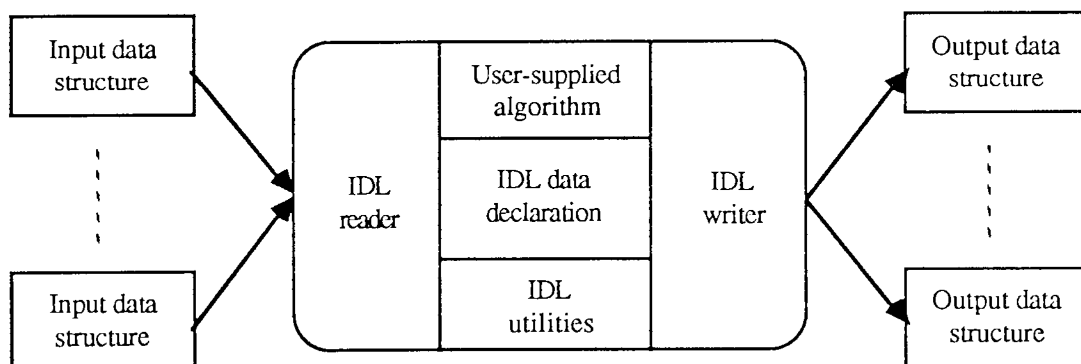


Figure 6.2 IDL model of a process

A user-supplied algorithm is written in terms of IDL data declarations and code which can make use of IDL-generated readers and writers for I/O of instances of data structures, and utilities for manipulating these instances in memory.

Expected benefits of IDL.

The philosophy behind IDL is that its data abstraction features allow the user to reason about his problem mainly in terms of useful data types, rather than in terms of implementation details. The resulting manipulation and I/O routines are guaranteed to be bug-free and are accurately and concisely documented by the IDL specification itself.

IDL Specification.

The basic building blocks of IDL specification are nodes and classes.

Nodes and Attributes

IDL has 4 basic types: integer, rational, boolean, string and 2 structured types sequence (ordered list) and set (unordered list).

A node is a named collection of zero or more named values called attributes. So a node corresponds to a C struct, and attributes to members of that struct. For example:

```
person   ====>  name      : string,
                address   : string,
                age       : integer;
```

would be equivalent to

```
struct person {
     char     *name;
     char     *address;
     int      age;
};
```

Type names for attributes of a node may also be names of nodes . For example:

```
binary-tree     ===>  name           :  string,
                      left-branch    :  binary-tree,
                      right-branch   :  binary-tree;
```

Classes

A class is name for a variable that can hold a reference to one of a set of nodes or other classes; thus it is equivalent to a C union. The possible members of this set are given, separated (as in BNF) by a vertical bar. For example:

```
student :: = research | undergrad | msc
```

Each of these members could itself be a class or node. To simplify let us consider them as the following nodes:

```
research   ===>   name          : string,
                  dept-code     : integer,
                  supervisor    : string,
                  funding       : string;

undergrad ===>    name          : string,
                  dept-code     : integer,
                  pers-tutor    : string,
                  yr-of-course  : integer;

msc        ===>   name          : string,
                  dept-code     : integer,
                  IT-or-SEA     : boolean,
                  proj-supev    : string;
```

Structures

All nodes and classes are grouped into named collections called structures. Each structure has a ROOT followed by a list of nodes and classes. This is usually employed to construct a structured data type defined by its nodes and classes. For example:

```
Structure students Root student-list is
student-list   ====> list : seq.of student
student ::= research | undergrad | msc
end
```

All nodes and classes of a structure must be "reachable" from the root of that structure.

Processes and ports

In order for IDL to create readers and writers for instances of IDL-specified data structures, the user must give input and output ports (which are typed), collected together under a process. A process is the IDL model of a computation, and is therefore equivalent to a C program. A process declaration specifies :

- the name of the process.
- a list of IDL data structures read and written by the process.
- the target language for the process.

For example:

```
process exams  is
target  C
pre   students-in    : students;
pre   marks-in       : marks;
post  results-out    : results;
end
```

This may be used in a program to process exam marks and produce a results list. The prefixes pre and post are used to specify whether a port is used for input or output respectively.

Assertion

Assertions permit the user to ensure that certain properties are true of his data structures; these are then used by a program called `idlcheck`, which carries out the checking of these properties. For example:

```
Assert Forall S in students do s.dept-code > 0 od
```

will check that each element in the list of students has a department code greater than zero.

Use of IDL in C

The user of IDL must submit a specification to the IDL translator which produces the necessary data definitions and functions to manipulate instances of these definitions. The output of the translator consists of two files for each process.

- The first is a `.c` file which contains the readers and writers and manipulation routines for that process. This is compiled into a `.o` file and the original source file is deleted.
- The second file is a `.h` file which the user must `#include` in his algorithm; this consists of all the typedef statements for his data types and some useful `#define` constants and macros.

The user then writes his algorithm in terms of variable declarations using type names given in the include file and manipulates these variables using the provided routines. I/O is performed by first opening a file in C's usual fashion (`fopen(...)`) and passing the resulting file pointer to the IDL generated reading and writing routines.

Finally the user compiles his algorithm together with the `.o` file produced by IDL. When running the executable version produced from compilation, the user must give all input in ASCII ERL, and will expect output in the same form. IDL also produces a `makefile` to maintain the files which it has created.

The scope of IDL

The use of IDL to deal with construction and manipulation of the AST has certain advantages. It provides bug-free data manipulation and I/O routines, and makes programs shorter and faster to write. The user can think of the problem in more abstract terms than getting involved in implementation details. Also, modules of a

large software project need not be written in a single language (due to use of ASCII ERL for external data representation). For example, output from one C program can become input to a Cobol program as long as both programs have been given the same IDL data specification, targeted to the appropriate language. The IDL data specification facilitates communication between members of a project team and provides language independent documentation of data structures used. However the tool does have distinct disadvantages listed below:

- Due to the use of ASCII ERL, modules written using IDL cannot appear in a chain of processes unless the non-IDL processes also use ASCII ERL.
- Although IDL attempts to free the user from implementation details, we believe that the user must become involved in the intricacies of IDL outputs.
- It is a very large system, which needs at least 10Mbytes of storage for installation, and does not run on all Unix machines. We attempted to run it on HLH ORION under the 4.3 BSD Unix operating system with no success. However on a SUN 3/160 workstation the IDL tool compiled and ran successfully. But even in the latter case, many changes were required to the Pascal sources supplied, and we were also obliged to subdivide some of the C files since the assembler was unable to cope with their size.
- The installation of this tool is obviously non-trivial, and in our case required approximately two months. Installation appears to be subject to the vagaries of different C and Pascal systems found with different Unix systems and suppliers.
- IDL is not a standard Unix tool and must therefore be provided separately; it is designed to be multi-purpose and is hence large for our purpose.

For these reasons, we decided that despite its attractions and the fact that the use of IDL would be consistent with our "layered" approach, the disadvantages of IDL are (at least for our application) overwhelming and therefore render it unsuitable for use with the CORGI system. The solution we have adopted is discussed in the section below.

6.3.3.2 The CORGI approach to generating the AST

In order to allow easy construction of an abstract syntax tree during parsing, CORGI generates the necessary data structure declarations, corresponding to rules in the generated yacc rule section. Thus for each yacc rule, CORGI produces an appropriate C struct definition. Such structs have two members. The first member is an integer which, since a yacc rule may consist of more than one legal alternative, is used to indicate which alternative has been reduced. The second member is a union of all of these possible alternatives, where each union member corresponds to one alternative. We illustrate this point using the following general examples:

(1) For each rule of the form

```
a : x1  x2......xn
  | y1  y2......ym         where xi, yj and zk are nonterminal
  | z1  z2...... zp;       symbols. i = 1..n, j = 1..m, k = 1..p
```

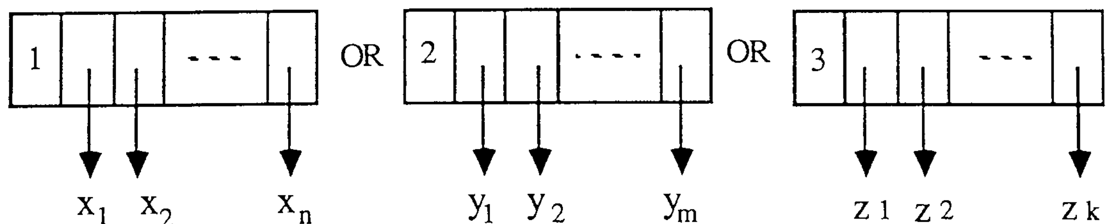
create a new type named A_TYPE as follows

```
typedef struct  a_type    *A_TYPE;      (*)
typedef struct  x1_type   *X1_TYPE;
typedef struct  x2_type   *X2_TYPE;
...
typedef struct  xn_type   *Xn_TYPE;
typedef struct  y1_type   *Y1_TYPE;
typedef struct  y2_type   *Y2_TYPE;
...
typedef struct  ym_type   *Ym_TYPE;
typedef struct  z1_type   *Z1_TYPE;
typedef struct  z2_type   *Z2_TYPE;
...
```

```
typedef struct  zp_type    *Zp_TYPE;
struct  a_type {
     int   type;
     union {
           struct alter_a_1 {
                 X1_TYPE         x1_1;
                 X2_TYPE         x2_2;
                 ...
                 Xn_TYPE         xn_n;
           }ALTER_a_1;
           struct alter_a_2 {
                 Y1_TYPE         y1_1;
                 Y2_TYPE         y2_2;
                 ...
                 Ym_TYPE         ym_m;
           }ALTER_a_2;
           struct alter_a_3 {
                 Z1_TYPE         z1_1;
                 Z2_TYPE         z2_2;
                 ...
                 Zp_TYPE         zp_p;
           }ALTER_a_3;
     }RIGHTSIDE;
}; /* end of node a */
```

lines such shown in (*) are used to overcome the declare before use principle in the definition of C data structure definition. Thus a value of type `A_TYPE` will be a pointer at an `a_type` structure value.

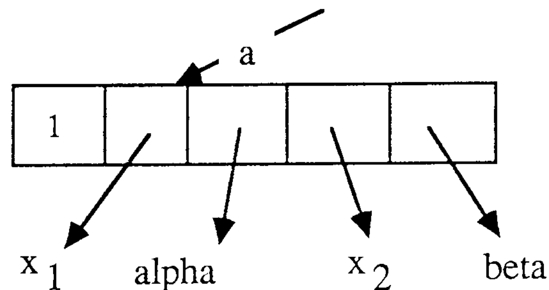
The field `type` will indicate which of the alternatives (eg. x_1 x_2......x_n or y_1 y_2......y_m or z_1 z_2...... z_p) has been parsed. Thus, depending on the structure of the source program, subsequent routines inserted into the yacc rule section by CORGI will construct one of the following nodes in the AST:



(2) For each rule which contains terminal symbols, additional facilities are required. Consider for example the rule:

```
a :  x1 alpha x2 beta
```

where x_1 and x2 are nonterminal symbols, and `alpha` and `beta` are either LEXEME or OPERATOR symbols. CORGI will generate a data structure with the following representation:



This representation is given by the following C data declaration.

```
typedef struct  a_type    *A_TYPE;
typedef struct  attribute_type   *ATTRIBUTE_TYPE;
struct  a_type {
   int      type;
   union {
      struct alter_a_1 {
            X1_TYPE             x1_1;
            ATTRIBUTE_TYPE      alpha_2;
            X2_TYPE             x2_3;
            ATTRIBUTE_TYPE      beta_4;
      }ALTER_a_1;
   }RIGHTSIDE;
};  /* end of a node  */

struct  attribute_type {
   int      type;
   UVAL     textvalue;
};
typedef union {
   int      d;    /* the lexeme's attribute is an integer  */
   char     c;    /* the lexeme's attribute is a character */
   char     *s;   /* the lexeme's attribute is a string    */
}UVAL;
```

Note that the names used for the node type declaration and for members of the resulting struct are generated directly from nonterminal and terminal names in the grammar specification, augmented with either a suffix or prefix. This allows the user to clearly see which data structure declarations refer to which rules. This will

be useful when inserting code to perform semantic analysis and code generation when walking through the tree. For example, given the following rule:

```
program:"PROC" procname "("param_list")" declaration proc_body "."
       |"PROC" procname declaration proc_body ".";
```

CORGI will generate a declaration for a node in the AST as shown below:

```
typedef struct  program_type      *PROGRAM_TYPE;
typedef struct  param_list_type   *PARAM_LIST_TYPE;
typedef struct  declaration_type  *DECLARATION_TYPE;
typedef struct  proc_body_type    *PROC_BODY_TYPE;
struct  program_type {
   int      type;
   union {
      struct alter_program_1 {
            PROCNAME_TYPE       procname_1;
            PARAM_LIST_TYPE     param_list_2;
            DECLARATION_TYPE    declaration_3;
            PROC_BODY_TYPE      proc_body_4;
      }ALTER_program_1;
      struct alter_program_2 {
            PROCNAME_TYPE       procname_1;
            DECLARATION_TYPE    declaration_2;
            PROC_BODY_TYPE      proc_body_3;
      }ALTER_program_2;
   }RIGHTSIDE;
};  /* end of program node  */
```

Code 6.1 The C struct associated with rule `program`

Node type names are generated by converting the original rule name to upper case and adding the suffix `_TYPE`. Members of the generated C struct (which correspond to the entries in the RHS of a rule) have names as they appear in the rule, suffixed by "_" followed by an identifying digit. This is required in order to deal with rules such as:

```
expression : expression operators expression;
```

where it would be illegal to generate a C struct with two members having the same name, hence the digit suffix is used to differentiate between them.

6.3.4 Augmenting lex and yacc specifications

Once the data structure definitions have been completed, the lex and yacc specification can be augmented with actions, which are created automatically by CORGI, to build the AST, evaluate lexemes, and store information in the tree for use by semantic analysis and code generation.

For the yacc specification, CORGI performs the following:

• Once all the necessary data types are known together with those symbols which need to be typed, we can proceed to augment the yacc specification. First the data types of the value stack must be defined (the default type of the value stack is integer) which is done in the declaration section of a yacc specification using a "%union" clause, as shown by the following example which corresponds to the `program` rule given above:

```
%union{
    ATTRIBUTE_TYPE      Vattribute;
    PROCNAME_TYPE       Vprocname;
    PROGRAM_TYPE        Vprogram;
    PARAM_LIST_TYPE     Vparam_list;
    DECLARATION_TYPE    Vdeclaration;
    PROC_BODY_TYPE      Vproc_body;
    ...
    };
```

• Then specific types must be associated with those terminal symbols for which a value is assigned to `yylval` during lexical analysis and to those nonterminal symbols for which $\$_i$ or `$$` are referenced. This is achieved by using yacc `%type` clauses as shown in the following example:

```
%type <Vattribute>      IDENTIFIER STRING INTEGER
%type <Vprogram>        program
%type <Vprocname>       procname
%type <Vparam_list>     param_list
%type <Vdeclaration>    declaration
%type <proc_body>       proc_body
```

- Actions are now generated in the yacc specification. Actions for most rules create a node, set the value of the result attribute from the values associated with the symbols on the RHS of the rule, and return a pointer to the node. In this way, the AST is created in a bottom-up fashion. For example,

```
program:"PROC" procname "("param_list")" declaration proc_body "."
                    { $$ = (PROGRAM_TYPE)mknode (5,1,$2,$4,$6,$7); }
          | "PROC" procname declaration proc_body "."
                    { $$ = (PROGRAM_TYPE)mknode (4,2,$2,$3,$4); };
```

where the first parameter to the function `mknode()` is the number of parameters, since it is useful to write a single `mknode()` function that accepts a variant parameter list; the second parameter indicates the alternative number that will be assigned to the type field. $\$_i$, `i = 1, n` are pointers to nodes in that alternative.

Lexeme attributes

In an earlier chapter, it was shown how the "==>" construct was used in the CORGI specification to indicate lexeme evaluation requirements. In such case values need to be passed from the lexical analyser generated by lex back to the parser, in addition to the token numbers of lexemes found during scanning. In a lex specification this is normally achieved using the yacc external variable `yylval`, which is declared as a union of all possible types of grammar symbols. CORGI generates actions in the lex specification which deal with lexeme evaluation ready for insertion of the resulting value in the AST. Such actions first evaluate the appropriate lexeme, and set `yylval` to the result, before returning the lexeme's token number. They are attached to rules for operators, lexemes, and attributed keywords and attributed delimiters, as required.

Given below is a CORGI specification which describes a language with lexemes `identifier`, `string` and `integer`, and in which keywords are a subset of a

lexeme which is found by CORGI to be the `identifier` lexeme. Also this language has an operator "+", and a delimiter "#".

```
KEYWORD_CASE        CASE_NONSIG.
LEXEME              identifier string integer "#".
OPERATORS
opr = \L {"+"}.
%%
...
identifier  = letter { letter | digit }   ==>evaluate_idr(0).
string      = """" { any_PR_char } """"   ==>evaluate_Cstr.
integer     = digit { digit }             ==>evalaute_denary.
...
```

From the above specification CORGI will generate the following lex specification.

```
...                   /* Lex's definition section  */
%%
"#"                   { /* It is an attributed delimiter */
                        yylval.Vattribute = (ATTRIBUTE_TYPE)
                            malloc (sizeof (ATTRIBUTE_TYPE));
                        yylval.Vattribute = DEL1;
                        yylval.Vattribute->type = 1;
                        return (DEL1);
                      }
"+"                   {  /* It is an operator  */
                        yylval.Vattribute = (ATTRIBUTE_TYPE)
                            malloc (sizeof (ATTRIBUTE_TYPE));
                        yylval.Vattribute->textvalue.d = OP1;
                        yylval.Vattribute->type = 1;
                        return (OP1);
                      }
#define               IDR_TOKEN    IDENTIFIER
int                   Toknum;
{identifier}          { /* It is an attributed lexeme */
                        Toknum = screen();
                        if(Toknum == IDR_TOKEN)
                        {
                          yylval.Vattribute = (ATTRIBUTE_TYPE)
                             malloc (sizeof (ATTRIBUTE_TYPE));
                          yylval.Vattribute-> textvalue.s =
                                               evaluate_idr();
                          yylval.Vattribute -> type = 3;
                        }
                        return (Toknum);
                      }
{string}              {  /* It is an attributed lexeme */
                          yylval.Vattribute = (ATTRIBUTE_TYPE)
                             malloc (sizeof (ATTRIBUTE_TYPE));
                          yylval.Vattribute -> textvalue.s =
                                               evaluate_Cstr();
                          yylval.Vattribute -> type = 3;
                          return (STRING);
                      }
{integer}             { /* It is an attributed lexeme  */
```

```
                    yylval.Vattribute = (ATTRIBUTE_TYPE)
                       malloc (sizeof (ATTRIBUTE_TYPE));
                    yylval.Vattribute -> textvalue.s =
                                        evaluate_denary();
                    yylval.Vattribute -> type = 1;
                    return (INTEGER);
                }
%%
screen() {}     /* definition of the screen functions */
```

6.3.5 Error-recovery

As described in section 2.2.1, the error-recovery strategy used by yacc is known as the "error production" scheme. In this method the user augments the grammar with `error` symbols where appropriate. When an error is detected the generated syntax analyser behaves as if it had just seen the special symbol "error" immediately before the token which caused the error. The syntax analyser then takes the nearest rule for which the `error` symbol is a valid token and resumes processing at this rule.

However the process of building a robust analyser by including `error` symbols in some rules is non-trivial. Adding `error` symbols in an arbitrary manner may cause the generation of shift/reduce and reduce/reduce conflicts. A naive user might add the `error` symbol as the last formulation of each rule; this approach has the potential to generate reduce/reduce conflicts as illustrated in the following example:

Given the simple rules:

```
A  : B                      B  : C
   | error;                    | error;
```

these are equivalent to:

```
A  : C
   | error
   | error;
```

The second form, given above, contains two legal alternatives that can be reduced in the case of error; hence a reduce/reduce conflict will arise. In this particular example

this has limited significance due to the simplicity of the example given, but more interesting (difficult) cases arise in practice.

In this section we shall present a technique which is based on the approach used by Schreiner & Friedman (1985) for the placement of the `error` symbol in grammar rules at appropriate points to avoid the difficulties discussed above. One of the main differences between our work and Schreiner & Friedman (1985), is that we automatically generate `error` symbols in their best place. Again the user is relieved of this arduous task. We believe that using this technique the parser is able to recover in most cases that arise in practice. The following section presents the algorithm used in placing the `error` symbol in grammar rules.

Algorithm for error symbol insertion

In most programming languages, the majority of errors seem to occur in repetitive constructs, as stated by Schreiner & Friedman (1985), which therefore need special attention when dealing with error-recovery, unlike optional constructs for which error-recovery is considerably simpler. In discussing this algorithm we shall, for convenience, omit the generation of actions which deal with building the abstract syntax tree.

In order to avoid a cascade of error messages, yacc insists that the parser must shift three terminal symbols beyond the point of error, before another apparent error results in a printed error message. But this way, a sequence of errors may result in only a single error message. With the yacc `yyerrok` action the parser can be persuaded to feel that it has accepted enough terminal symbols, and thus to report errors in close proximity to one another. The placement of `error` symbols and the `yyerrok` action is guided by the recommendations given below.

- The reason why optional constructs do not present a problem for error-recovery is illustrated by the following simple example:

  Given the rules `A = ["+"]` and `B = [params]`, CORGI generates the following:

```
A :  /* empty */        B  :  /* empty */
  |  '+';                  |  params;
```

  With the rule `A` if the parsed symbol were ',' instead of '+', the parser would always reduce by the empty alternative and take the ',' as the next lookahead symbol. With rule `B`, if there is an error in the input, it will be processed by the rule describing `params`. In every case the parser would never associate an error to an optional construct. Therefore an error production is never needed for such cases.

- For each repetitive construct which contains one single element, the following algorithm is used:

  given the rule A = {B}, CORGI generates the following:

```
A :  /* empty */
  |  A  B  { yyerrok; }
  |  A  error;
```

- For each repetitive construct which contains at least two elements, the following algorithm is used:

  given the rule A = {B C}, CORGI generates the following:

```
A : /* empty */
  |  A  B  C      { yyerrok; }
  |  A  error
  |  A  error  C  { yyerrok; }
  |  A  B  error;
```

- For each grouped construct, the following algorithm is used:

given the rule `A = ( B | C )`, CORGI generates the following:

```
A : B
  | C
  | error;
```

- For each rule from the original EBNF of the form `A = B c d` where `B` is a nonterminal symbol, and `c` and `d` are terminal symbols, CORGI generates the following:

```
A : B c d
  | B error d
  | B c error
```

- For each rule of the following form: `A = b C D` where the first symbol in the rule is a terminal symbol which is followed by at least one terminal or nonterminal symbols, CORGI generates the following set of rules:

```
A  : b C D
   | b error;
```

The technique used for the placement of `error` symbols does not guarantee that a useful input symbol is not ignored in some error situations; however, it does guarantee (Schreiner & Friedman, 1985) that no reduce/reduce or shift/reduce conflicts are introduced. We believe that this technique results in robust syntax analysers for common language constructs in a systematic manner. This contention will be demonstrated in the next chapter.

6.3.6 The manipulation routines

The actions inserted into the yacc specification as described in section 6.3.3.2, build an abstract syntax tree for the source program being parsed. CORGI also generates routines for input/output of this tree in flattened form, and for tree-walking during semantic analysis and code generation. The structure of these routines closely

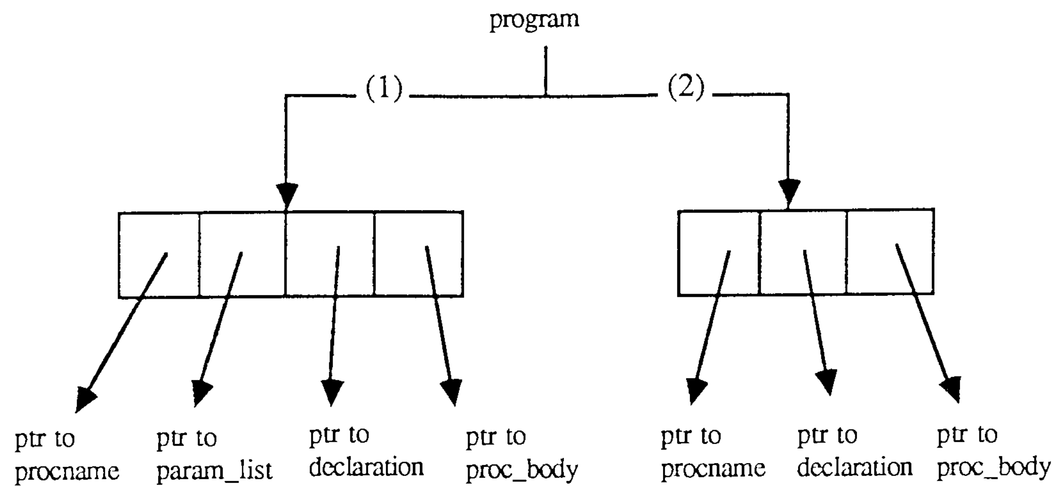
follows the rules in the generated yacc specification. For each generated data structure definition of an AST node, CORGI also generates writer, reader and walker functions.

The writer functions are designed so that when parsing is complete the entire tree can be flattened and output to permanent storage as described in section 5.4.1. Typically, the writer functions form a mutually-recursive hierarchy of calls which begin at the root of the tree, and proceed towards the leaves. If a node corresponding to a particular rule is not found in the tree, CORGI outputs a `zero` `(0)` followed by the name of the rule. If however, such a node is present a `one` `(1)` is output with the rule name, and the writer routine then deals with each "subtree" associated with that node in turn.

Thus for non-leaf nodes, further writer functions are invoked, and for leaf nodes their attribute values are output. Using this recursive traversal of the tree, a linear representation is generated, in terms of rule names and terminal values. A `writer` routine for the following grammar rule is given in Code 6.2 below:

```
program:"PROC" procname "("param_list")" declaration proc_body "."
       |"PROC" procname declaration proc_body ".";
```

The data structure associated with this grammar rule, which is given in Code 6.1, is shown diagrammatically below:



where path (1) represents the first alternative and path (2) represents the second alternative of the grammar rule given above.

Given input with the form

```
procname
declaration
proc_body
.
```

the writer function given in Code 6.2 would generate the following linear form of the abstract syntax tree into the `tree_dat` file.

```
1     <program>
2           <alternative number of program>
1     <procname>
1           <alternative number of procname>

...  /* deal with procname routine */

1     <declaration>
1           <alternative number of declaration>

...  /* deal with declaration routine */

1     <proc_body>
1           <alternative number of proc_body>

...  /deal with proc_body routine */
```

```
write_program(ptr)
PROGRAM_TYPE      ptr;
{

if(ptr == NULL)
     fprintf(fp_data, "0\t<program>\n");
else{
     fprintf(fp_data, "1\t<program>\n");
     fprintf(fp_data,"%d\t\t<alternative number of program>\n",
     ptr->type);
     switch(ptr->type){
     case 1 :  /* it is alternative 1 */
         write_procname(
         ptr->RIGHTSIDE.ALTER_program_1.procname_1);
         write_param_list(
         ptr->RIGHTSIDE.ALTER_program_1.param_list_2);
         write_declaration(
         ptr->RIGHTSIDE.ALTER_program_1.declaration_3);
         write_proc_body(
         ptr->RIGHTSIDE.ALTER_program_1.proc_body_4);
         break;
     case 2 :  /* it is alternative 2 */
         write_procname(
         ptr->RIGHTSIDE.ALTER_program_1.procname_1);
         write_declaration(
         ptr->RIGHTSIDE.ALTER_program_2.declaration_3);
         write_proc_body(
         ptr->RIGHTSIDE.ALTER_program_2.proc_body_4);
         break;
     default :
         printf(
         "ERROR - Wrong alternative number generated\n");

     }  /* end of switch */
}  /*  end of else  */
} /* end of write_program()   */
```

Code 6.2 the writer routine of the rule `program`

The generated reader routines are designed to take the linear representation of the tree, produced by the tree writer routines, and re-construct the original dynamically allocated data structures. The overall structure of the reader for the above rule is equivalent to that of the writer, but reads in the linear representation and re-constructs the tree rather than writing it.

walker routines which are also generated by CORGI can then be called to perform a recursive tree-traversal in a similar manner to the writer functions, except that no output is produced. It is intended that the user of the CORGI system will insert his

own routines into this generated "template", to perform semantic analysis and code generation at appropriate tree nodes. This has the advantage that the tree-walk routines are guaranteed to provide a bug-free tree traversal, and code which deals with a particular rule can be easily identified due to the naming conventions used (a routine which traverses a node A is called `walker_A()`). The generated walker routines for the above rule is given in Code 6.3. Using an editor, the user may insert code in the above walker routine to perform whatever semantic actions are required for the application; the user's code has access to information stored in the tree. Language-tailored walker routines may be obtained by simple changes to the generated walker routines. The user may insert code in the above walker routine to perform whatever semantic actions are required for the application; the user's code has access to information stored in the tree. Language-tailored walker routines may be obtained by simple changes to the generated walker routines. The highlighted code in the walker routine given in Code 6.3 is an example of user's inserted code.

```
walk_program(ptr)
PROGRAM_TYPE      ptr;
{
BOOLEAN    anyprocs = FALSE;
int        jmain;

if(ptr == NULL) ;
else{
     switch(ptr->type){
     case 1 :  /* it is alternative 1 */
            /*  deal with the lexeme identifier   */
            if(ptr->RIGHTSIDE.ALTER_program_1.procname_1)
            {
               anyprocs = TRUE;
               cg2(J,0);
               jmain = PSused;
            }
            walk_procname(
            ptr->RIGHTSIDE.ALTER_program_1.procname_1);
            walk_param_list(
            ptr->RIGHTSIDE.ALTER_program_1.param_list_2);
            walk_declaration(
            ptr->RIGHTSIDE.ALTER_program_1.declaration_3);
            walk_proc_body(
            ptr->RIGHTSIDE.ALTER_program_1.proc_body_4);
            break;
     case 2 :  /* it is alternative 2 */
            /*  deal with the lexeme identifier   */
```

```
            if(ptr->RIGHTSIDE.ALTER_program_1.procname_1)
            {
                anyprocs = TRUE;
                cg2(J,0);
                jmain = PSused;
            }
            walk_procname(
            ptr->RIGHTSIDE.ALTER_program_2.procname_1);
            walk_declaration(
            ptr->RIGHTSIDE.ALTER_program_2.declaration_3);
            walk_proc_body(
            ptr->RIGHTSIDE.ALTER_program_2.proc_body_4);
            break;
    default :
            printf(
            "ERROR - Wrong alternative number generated\n");
    }  /* end of switch */
     if(anyprocs)            /* fix up jump to code */
     PS[jmain] = PSused+1;   /* for main program    */
}  /*  end of else  */
  cg1(HALT);     /* end of the program  */
} /* end of walk_program()  */
```

Code 6.3 the walker routine of the rule `program`

6.4 Error handling in programs generated by lex and yacc

When using lex and yacc, there are two possible types of error which must be catered for. These are:

- errors detected at generation time
- errors detected at execution time.

The first kind of error occurs due to invalid lex or yacc specifications, caused for example by the duplicate definition of a nonterminal, an undefined nonterminal or by any violation of the syntactic rules for defining lex and yacc specifications. These errors are detected when running the lex and yacc processors to generate the required lexical and syntax analysers for the desired grammar: hence, the term generation-time errors.

The second kind of error occurs when executing the programs generated by lex and yacc, in which case the errors are due to the incompatibility of the source programs

with the grammar defining the language in which they are written. In this section, we address specifically the question of these compiler execution-time errors.

One of the disadvantages of building CORGI as a preprocessor to lex and yacc, is that we are obliged to accept the deficiencies of these two tools. Error reporting and error messages of lex and yacc are very poor and need to be improved. Methods of remedying this situation have been reported in the literature; Schreiner & Friedman (1985) presented an attractive solution to the problem, and we have adopted this approach in the CORGI system. Their suggestions were as follows:

- modification of the skeletal parser which is held in a file called /usr/lib/yaccpar and included into yacc's output
- creating new lex and yacc libraries
- including some additional features into the lex specification.

In the following section we shall show an example of the improvements, which have been incorporated into our system, by running some source input using the original and the new debugging facilities. All the other error messages which can arise during parsing are similarly clarified.

Given the following erroneous input where a list of numbers, separated by spaces, is expected:

10, 20 30.

The generated syntax analyser using the original debugging facilities would generate the following output:

```
state 0, char 037777777777
reduce 2
state 2, char 037777777777
state 4, char 037777777777
reduce 3
state 2, char 037777777777
```

```
state 5, char 054
reduce 4
state 2, char 054
error recovery discards char 44
state 4, char 037777777777
reduce 3
state 2, char 037777777777
state 4, char 037777777777
reduce 3
state 2, char 037777777777
state 3, char 037777777777
reduce 1
state 1, char 037777777777

Compilation error(s): 1
Compilation terminated
```

However using the new debugging facilities the syntax analyser would generate the following output, which provides far clearer information as to its operation. In a final operational compiler only lines such as the line marked with (*), which are more suitable for the typical user, are output.

```
[yydebug] push state 0
[yydebug] reduce by (2)
[yydebug] push state 2
[yydebug] reading CONSTANT
[yydebug] push state 4
[yydebug] reduce by (3)
[yydebug] push state 2
[yydebug] reading ','
[error 1] line 1 near ",":expecting: '.'CONSTANT      (*)
[yydebug] push state 5
[yydebug] reduce by (4)
[yydebug] push state 2
[yydebug] recovery discards ','
[yydebug] reading CONSTANT
[yydebug] push state 4
[yydebug] reduce by (3)
[yydebug] push state 2
[yydebug] reading CONSTANT
[yydebug] push state 4
[yydebug] reduce by (3)
[yydebug] push state 2
[yydebug] reading '.'
[yydebug] push state 3
[yydebug] reduce by (1)
[yydebug] push state 1

Compilation error(s): 1
Compilation terminated
```

Note that with the original debugging facility, not much information is given about the type of error, the cause of the error, nor even the line number where the error

occurred. However all these are given by the new debugging facility. Note also that the new version reports the input read at each stage unlike the original version. The numeric ASCII code of characters encountered in the input is used in the original version when discarding tokens after an error, unlike the new version where the text of the token is displayed. This facilitates reading of the output and makes error correction much faster.

6.5 Summary

In this chapter, we have shown how the design outline given in chapter 5 has been used to construct the CORGI system. We have described how CORGI generates a valid lex specification for producing a lexical analyser, by extracting information from the CORGI input specification relating to lexical analysis; we have also shown how we translate the original EBNF rules corresponding to lexical features into lex's regular expressions. The manner in which CORGI generates a yacc specification has also been described, including the translation process from EBNF to yacc's BNF notation.

We have shown how the generated lex and yacc specifications are further augmented by CORGI. The augmentation of the lex specification involved provision for the evaluation of certain lexemes, storing information in the generated tree, and processing nested comments. The augmentation of the yacc specification involved the automatic generation of the error productions and the tree manipulation actions. These actions deal with the construction and manipulation of the abstract syntax tree.

As far as the generation of the data structure for the tree and the routines to manipulate it are concerned, an interesting alternative method was presented, ie making use of the IDL software tool; however, for the reasons given earlier this

method was rejected, and a direct method was used to achieve the automatic construction of the abstract syntax tree and associated manipulation routines.

Another important point discussed in this chapter is the shortcomings of the default error handling of programs generated by lex and yacc; a way of remedying these deficiencies was presented.

In chapter 7, we shall present the results of a number of tests which we conducted using CORGI with grammars for typical modern programming languages.

Chapter 7

Demonstration and Evaluation of CORGI

7.1 Introduction

The evaluation and validation of a large system is an important step in the software lifecycle. In order to validate such a system, we should show that its software components correspond to the original specification. In principle, this may be performed using formal, mathematical methods; however, currently the cost and difficulty associated with such an approach makes it viable for use only on small, highly important parts of the system.

Additional complexity is introduced by the fact that we do not only wish to prove the correctness of a hand-written program, but also that of generated software. In fact, for the CORGI system, the following three levels of formal proof would be required:

Functional specification	——(1)——→	implementation
Implementation	——(2)——→	programs and specifications
Programs and Specifications	——(3)——→	generated program

Due to the impracticality of such formal proof, and given the time constraints of this research project, we have necessarily limited the validation process to testing using representative examples.

7.2 Testing

The CORGI system was developed and tested incrementally using both test data designed to test very specific parts of the system, and real-life test data such as grammars for certain programming language such as Modula-2, Pascal, a subset of C, and MSL. For each of these languages we continued testing until a correct parser was produced which was able to successfully parse a number of test programs written in that language.

The design of the system is modular, and hence each module was tested until we were satisfied as to its correct operation; this meant that as far as possible we were sure that any errors encountered were due to the current module under development, or at least to its interface with already tested modules. Thus each module was gradually integrated into a complete system.

The system is portable across Unix systems, and has been tested on a HLH ORION running a 4.3 BSD Unix operating system and a SUN 3/160 workstation. The system is written in the C programming language and it consists of approximately 9000 lines of code.

7.3 Application of CORGI

In this section we shall show how a full CORGI specification should be written, using the MSL language as discussed previously, and also briefly review extra steps required for other programming languages. We shall also demonstrate the error-recovery mechanism, as presented in Chapter 6, for major programming language constructs which involve repetition and grouping in grammar rules. This will involve running erroneous source programs through the parser generated by CORGI.

In order to evaluate the performance of CORGI, we shall present results of the test runs using artificial test specifications and also those for real programming languages. These results will show the relationship between the number of rules in the grammar and CPU time required to produce a parser for the grammar. We also compare the time required by the parser to process a number of example source programs, with that of a parser produced from hand-written lex and yacc specifications as discussed in chapter 4.

Finally we shall show how the output of CORGI may easily be augmented so as to perform semantic analysis and code generation for MSL.

7.3.1 Writing a CORGI specification for MSL

As mentioned in chapter 5 a CORGI input specification has the following format:

```
{ Annotation section }
%%
{ Rule section }
%%
{ User's routine section }
```

The best approach to producing a specification for a given language is to tackle each section in the order in which they appear in the above format.

Annotation section for MSL: As seen previously this section consists of four major parts, each of which is given below. The order in which these are specified is not important.

• Comment: Comments in MSL cannot be nested; they start with "--" and end with a carriage return, similar to Ada, and hence their CORGI specification would be:

```
STARTCOMMENT        "--".
ENDCOMMENT          NEWLINE.
```

• Case sensitivity of keywords: In MSL the case of keywords is not significant, and this is described as follows:

```
KEYWORD_CASE        CASE_NONSIG.
```

• Lexeme list: As previously described in section 4.4.1, lexemes in MSL are: identifier, integer literal, boolean literal, text literal, keyword, operator and delimiters. As operators are important for semantic analysis and code generation, they are treated separately in the next section.

In general, delimiters are only relevant to context free analysis, therefore they are normally not considered in this section. However there are exceptions which occur in MSL. The symbols "#" and "+" are in certain contexts (the MSL READ & WRITE statements) used with special semantic significance, which means that they are not only important for the context-free features but also for code generation. For example:

```
READ +  data1, data2
WRITE + #text1, "hello", #text2, numvalue
```

The above statements are equivalent to `read/write` in Pascal as opposed to `readln/writeln`, this is indicated by the presence of the symbol +. The # symbol on the other hand, indicates that the values of `text1` and `text2` are to be interpreted as strings rather than as numeric values such as is the case for the last parameter `numvalue`.

In such a case the LEXEME list for MSL may be given as follows:

```
LEXEME     identifier int text "TRUE" "FALSE" "#".
```

Note that the optional plus "+" does not appear in the LEXEME list, since it is also an operator and is dealt with as such, to avoid ambiguity (each symbol must have a

unique name in lex). The distinction between "+" used in conjunction with read/write or used as an arithmetic operator (see the example MSL programs in appendix B) is made during semantic analysis and code generation.

The definition of each of the lexemes from the LEXEME list will be given in the rule section .

• Operators: Operators in MSL all have the same precedence and the same associativity, left-to-right. They are therefore described as follows:

```
OPERATORS
opr =\L {"+" "-" "*" "/" "<" ">" "<=" ">=""<>" "=" "%" "&"}.
```

Rule section for MSL: The syntax of MSL in EBNF notation is given in figure 4.5, but we reproduce it here for convenience.

```
Program        =  ["RESERVE" int ] {ProcDec} Series ".".
ProcDec        =  "PROC" idr  ["(" Idrlist ")"] Series "END".
Idrlist        =  idr {"," idr}.
Series         =  Stmt {Stmt}.
Stmt           =  Assignst | Whilest | Ifst
                | Callst | Readst | Writest.
Assignst       =  StoreAccess ":=" Expr.
Whilest        =  "WHILE" Expr "DO" Series "OD".
Ifst           =  "IF" Expr "THEN" Series ["ELSE" Series] "FI".
Callst         =  "CALL" idr ["(" Exprlist ")"].
Exprlist       =  Expr {"," Expr}.
Readst         =  "READ" Optionalplus  Readinlist.
Writest        =  "WRITE" Optionalplus  Writeoutlist.
Readinlist     =  Optionalhash StoreAccess
                  { "," Optionalhash StoreAccess }.
Writeoutlist   =  Optionalhash Expr {"," Optionalhash Expr}.
```

```
Optionalhash   =  ["#"].

Optionalplus   =  ["+"].

Expr           =  Operand {opr Operand}.

Operand        =  int | text | "TRUE" | "FALSE" | "@" idr
                  | StoreAccess |"(" Expr ")".

StoreAccess    =  idr ["!" Operand].

idr            =  letter { letter | digit }.

int            =  digit { digit }.

text           =  """" {stringpic} """" .

stringpic      =  anybut_DQ_NL.
```

MSL syntax in EBNF

The rule section of the CORGI specification for MSL requires only a few minor additions to the above syntax, namely augmenting the grammar by associating actions for attribute evaluation with rules which define the lexemes given in the lexeme list above, as follows:

```
idr           = letter { letter | digit }   ==> evaluate_idr(0).
int           = digit { digit }             ==> evaluate_denary.
text          = """" {stringpic} """"       ==> evaluate_Pstr.
Optionalhash  = ["#"].
```

`evaluate_idr(0)` is a function that returns a pointer to the text of the identifier, taking one parameter which indicates the case sensitivity of this lexeme. For MSL, case is not significant for an identifier and this is indicated by 0.

`evaluate_denary` and `evaluate_Pstr` are two functions from the CORGI library to evaluate a denary number and Pascal-like string respectively. Note that evaluate_Pstr() also cater for strings with delimiters being (") as opposed to (') in Pascal.

User-routine section for MSL:

In the case of MSL, this section is omitted since all the evaluation routines needed for MSL lexemes are available in the CORGI library.

It should be noted from comparison of the example of generating an MSL compiler as discussed earlier that the use of CORGI requires considerably less effort than that needed when producing lex and yacc specifications by hand. CORGI also has the advantage of providing an attribute evaluation library, and automatic declaration and generation of the AST together with processing routines for the data structures formed.

Appendices A and B provide in full detail a comparison of the use of CORGI as opposed to the previous method (using lex and yacc) for a realistic example. From this comparison we see the evident advantages of using CORGI.

It should also be noted from the data given in appendix C that a compiler produced from a CORGI specification can emit identical object code to lex/yacc and hand-crafted version. Moreover, the CORGI generated compiler shows similar performance in terms of execution speed to the other alternative approaches; we shall discuss this issue in more depth in a later section.

7.3.2 Overview of other PLs

We also consider how various features of other programming languages such as Pascal, Modula-2 and Ada can be specified using the CORGI system, in order to show that such features can be accommodated.

Note that complete CORGI specifications for Modula-2 and Pascal were written and tested successfully; complete listings for this work are given in appendix E. Here we point out some of the more interesting issues which arise.

The CORGI mechanism for dealing with comments is capable of catering for generally-used styles of comments: most comments fall into one of the following categories:

- those found in Ada, MSL, Occam, where there is one opening symbol and one closing symbol:

```
STARTCOMMENT            "--".
ENDCOMMENT              NEWLINE.
```

- those found in Pascal where more than one symbol can be used to close a comment:

```
STARTCOMMENT        "{"   |  "(*".
ENDCOMMENT          "}"   "*)"   |   "*)"   "}".
```

- those found in Modula-2 where nested comments are allowed.

```
STARTCOMMENT        "(*"  NESTED.
ENDCOMMENT          "*)".
```

Specification of the case sensitivity of keywords is straightforward, since in most programming languages case is either significant or non-significant. CORGI provides facilities for both eventualities, together with the extra possibility where case is non-significant but cannot be mixed in a single lexeme. Thus the three possible cases are:

`KEYWORD-CASE CASE-NONSIG.`	as in Pascal, Ada etc...
`KEYWORD-CASE CASE-SIG.`	as in C, Miranda, Modula-2, Occam etc...
`KEYWORD-CASE CASE-NOTMIXED.`	

One aspect which must be borne in mind when writing the rule section is the relationship between the grammar rules which describe the lexemes and the other rules and whether the user calls the CORGI library routines for lexeme evaluation. A potential problem arises when the user may wish to combine the description of two different lexemes into one single rule. This case has been identified so far only in the syntax of Ada (Rogers, 1984) which describes a `decimal-literal` and `integer` as shown below:

```
decimal-literal   = integer [ "." integer ] [exponent].
integer           = digit { [underline] digit}.
exponent          = "E" ["+"] integer | "E" "-" integer.
based-literal     = base "#" based-int ["." based-int] "#"
                    [exponent].
base              = integer.
based-int         = extended-digit {[underline] extended-digit}.
extended-integer = digit | letter.
```

Example 7.1

With the LEXEME list as:

```
LEXEME            decimal-literal based-literal  ...
```

The evaluation of the `integer` and `decimal-literal` lexemes will require separate functions. However a user wishing to use only the CORGI library functions for lexeme evaluation may rearrange the grammar given in example 7.1 into the following grammar rewriting the rules marked (*) below:

```
decimal-literal  = decimal-integer | decimal-real.
decimal-integer  = integer                 ==> evaluate_denary.
decimal-real     = integer "." integer [exponent].   (*)
integer          = digit { [underline] digit}.
exponent         = "E" ["+"] integer | "E" "-" integer.
based-literal    = based-integer | based-real.
based-integer    = base "#" based-integer "#"
```

```
                                        ==> evaluate_Ada_int().
based-real        = base "#" based-int "." based-int "#"
                    [exponent].                        (*)
base              = integer.
based-int         = extended-digit {[underline] extended-digit}.
extended-integer  = digit | letter.
```

Example 7.2

or the following:

```
decimal-literal  = integer          ==> evaluate_denary
                 | integer "." integer [exponent].        (*)

based-literal    = base "#" based-integer "#"==> evaluate_Ada_int
                 | base "#" based-int "." based-int "#"
                   [exponent]                              (*)
```

Example 7.3

If the grammar describing `decimal-literal` and `based-literal` is given as in Example 7.2, the the LEXEME list may be given as:

```
LEXEME  decimal-integer decimal-real based-integer based-real ...
```

However, if the grammar is given in the form of Example 7.3 then the LEXEME list would be:

```
LEXEME  decimal-literal based-literal ...
```

In fact this only rarely presents a problem, since the description of lexemes in most reference manual grammars is given in the form shown in Example 1.2 or 7.3 and very rarely in the form shown in Example 7.1.

But since the idea of CORGI is to work so far as possible from the reference manual form of the grammar, it is more consistent and perhaps preferable to adhere to the

original form of the grammar and provide the necessary additional evaluation routines.

For Pascal, Modula-2 and C the LEXEME list is straightforward, and may be given as follows:

For Pascal (BS6192, 1982)

```
LEXEME  identifier signed-integer signed-real character-string.
```

where each lexeme is as follows:

```
identifier        = letter { letter | digit } ==> evaluate-idr(0).
signed-integer    = [sign] unsigned-integer   ==> evaluate_denary.
signed-real       = [sign] unsigned-real.                  (*)
character-string  = "'" string-element {string-element } "'"
                                               ==> evaluate_Pstr.
```

In Modula-2, the LEXEME list would be:

```
LEXEME   identifier integer real string.
```

where the grammar rules describing the lexemes are as follows:

```
identifier = letter { letter | digit }    ==> evaluate_idr(0).
integer    = digit { digit }              ==> evaluate_denary
           | octaldigit { octaldigit } ("B" | "C")
                                          ==> evaluate_Mod2_Octal.
           | digit { hexdigit } "H"       ==> evaluate_Mod2_Hex.
string     = "'"  { anybut_SQ_NL } "'"    ==> evaluate_Pstr
           | """" { anybut_DQ_NL } """"   ==> evaluate_Pstr.
real       = digit { digit } "." {digit} [ScaleFactor].
```

In C the LEXEME list may be given as follows:

```
LEXEME  identifier integer-constant floating-constant
        string-constant character-constant.
```

with the following grammar describing the lexemes as given in the C reference manual (Harbison & Steele, 1984):

```
identifier            = first-character { following-character }
                                                 ==> evaluate_idr(1).
first-character       = letter   | underscore.
following-character   = letter   | underscore | digit.
integer-constant      = decimal-constant         ==> evaluate_denary
                      | octal-constant           ==> evaluate_C_Octal
                      | hexadecimal-constant     ==> evaluate_C_Hex.
floating-constant     = digit-sequence exponent
                      | dotted-digits [exponent].          (*)
string-constant       = """" { character } """"  ==> evaluate_Cstr.
character-constant    = "'" character "'"        ==> evaluate_Cchar.
...
```

Note that an evaluation routine for real numbers is not provided by CORGI since this is inevitably dependent on the target architecture and therefore is left to the user's responsibility. The value associated with such lexemes is a string of characters as it is given in the input. This is in fact the default case for lexemes with no evaluation action attached to them, as previously discussed.

Operators in most programming languages may be categorized into classes, for example adding operators, multiplying operators, relational operators etc. Also they all have one of three types of associativity: left-to-right, right-to-left or non-associative which is indicated in CORGI by the use of \L, \R or \N respectively. For example, in Pascal or Modula-2 the OPERATORS section would be:

```
OPERATORS
relational-oprs      = \L { "<" ">" "<>" "=" ">=" "<=" "IN" }.
adding-oprs          = \L { "+" "-" "OR" }.
multiplying-oprs     = \L { "*" "/" "DIV" "MOD" "AND" }.
not-oprs             = \R { "NOT" }.
```

where the order in which these rules are given is important since it gives the precedence of the operators (in ascending order). Each of the above operator nonterminals are used in the grammar as follows:

```
expression    = simple-expr[ relational-oprs simple-expr ].
simple-expr   = [ sign ] term { adding-oprs term }.
term          = factor { multiplying-oprs factor }.
factor        = variable-access | ... | not-oprs factor.
```

An interesting example is Occam where all the operators have the same associativity ie. left-to-right and they all have the same precedence. The operators in Occam fall into two categories: dyadic operators and monadic operators with the same precedence. This is a very important facet dealing with operators. If for instance we describe Occam operators as:

```
OPERATORS
dyadic-operators     = \L { ... "+"  "/" "\/"   ... }.
monadic-operators    = \L { ... "~"  "-" "+" ... }.
```

then we are saying that monadic operators have a higher precedence than the dyadic operators, but this is not the case in Occam, since they all have the same precedence. Therefore this should be described as:

```
OPERATORS
Occam-operator     = \L { dyadic-operators
                          monadic- operators }.
dyadic-operators  = { ... "+"  "/" "\/"   ... }.
monadic-operators = { ... "~"  "-" "+" ... }.
```

where the grammar for an Occam expression as given by Hoare (1988) is:

```
expression = monadic-operators operand
           | operand dyadic-operators operand
           | conversion
           | operand
           ....
```

In such situation one may wish to associate the precedence of one class of operators, already defined, with another class. For instance in some languages the unary operators are given the precedence of `multiplying-operators` and the method illustrated above is the only way to specify such a feature.

7.3.3 Other limitations of the CORGI system

Certain features of programming languages cannot be specified using the CORGI system. CORGI does not deal with early programming languages such as Fortran and Cobol which have column-based layout rules. Also, it cannot cope with certain modern languages such as Occam and Miranda which use indentation to specify block structure. However, only limited modifications to CORGI would be required to allow for this latter feature. The CORGI system assumes that the keywords are reserved which is not the case with PL/I and Fortran. It appears now to be generally accepted that programming language design should not incorporate this feature since it severely restricts the clarity of the language. For example, in PL/I, one may have the following:

`IF THEN THEN ELSE := 1 ELSE IF := 2` which is equivalent to

`IF cond THEN stat1 ELSE stat2`

7.3.4 Testing error recovery

As mentioned in chapter 6, the algorithm used to place the `error` symbol together with the `yyerrok` action for the error recovery mechanism is based on that given by Schreiner & Friedman (1985). In this section we shall demonstrate the efficacy of an such algorithm using small examples and then give a full MSL test specification. The results given here are reproduced after running the examples on the HLH ORION 1/05 Unix machine.

- The first demonstration deals with an optional list of IDENTIFIERs with the following EBNF rule:

```
list = {IDENTIFIER} ".".
```

which is translated into the following yacc specification which we have augmented with tracing actions. These tracing actions will show which alternative is reduced by the parser:

```
%token    IDENTIFIER '.'  ','
%%
list         : single-item '.';
single-item  :   /*  empty */
             | single-item IDENTIFIER       {  yyerrok; }
             | single-item error   {printf("Error type 1\n"); };
%%
main()
{
#ifdef LEXDEBUG
  debug = 1;   /* debug = 1 for debugging and 0 otherwise   */
#endif
extern  int yynerrs;
int flag = 0;

    flag = yyparse();
    printf("Compilation error(s): %d\n", yynerrs);
    if(flag)
         printf("Compilation aborted\n");
    else
         printf("Compilation terminated\n");
}
```

Given the following input

```
1) a, ab.
```

the parser produces the following output:

```
[error 1] line 1 near ",": expecting:  '.' CONSTANT
Error type 1

Compilation error(s): 1
Compilation terminated
```

For the following incorrect input:

```
2) a, ab abc, abcd.
```

the parser produces the following output:

```
[error 1] line 1 near ",": expecting:'.' IDENTIFIER
Error type 1
[error 2] line 1 near ",": expecting:'.' IDENTIFIER
Error type 1

Compilation error(s): 2
Compilation terminated
```

The first example shows that the parser has recovered after the error caused by the ','. The value returned by the parser is 0 which indicates that the parser has successfully recovered from a syntax error.

The second example shows not only that the parser has recovered from the first error but also from the second error and the value returned is again 0.

- The second demonstration deals with a repeated list of at least two elements, in which case there are more potential errors.

```
idrlist    =    IDENTIFIER { ","  IDENTIFIER } ".".
```

which is translated into the following yacc specification:

```
%token    IDENTIFIER '.' ',' '+'
%%
idrlist      :  IDENTIFIER list '.'   { yyerrok; }
             |  error list '.'    { printf(" Error type 1\n");}
             | IDENTIFIER list error
                               { printf(" Error type 2\n");};

list         :  /* empty */
             | list   ','  IDENTIFIER   { yyerrok;}
             | list error      { printf("Error type 3\n");}
             | list error IDENTIFIER
                          { printf("Error type 4\n");   yyerrok;}
             | list ',' error  { printf("Error type 5\n");  };

%%
main()
{
#ifdef LEXDEBUG
debug = 1;  /* debug = 1 for debugging and 0 otherwise   */
#endif
extern  int yynerrs;
```

```
int  flag = 0;

 flag = yyparse();
 printf("Compilation error(s): %d\n", yynerrs);
 if(flag)
      printf("Compilation aborted\n");
 else
      printf("Compilation terminated\n");

}
```

For the following incorrect input:

```
1)    a, ab+.
```

the parser produces the following output:

```
[error 1] line 1 near "+": expecting:  ',' '.'
Error type 3

Compilation error(s): 1
Compilation terminated
```

For the second incorrect input:

```
2)   a ab.
```

the parser produces the following output:

```
[error 1] line 1 near "ab": expecting:  ',' '.'
Error type 4

Compilation error(s): 1
Compilation terminated
```

For the following incorrect input:

```
3)   a,.
```

the parser produces the following output:

```
[error 1] line 1 near ".": expecting:  IDENTIFIER
Error type 5

Compilation error(s): 1
Compilation terminated
```

For the following incorrect input:

```
4)   a, ab+ abc, abcd abcde.
```

the parser produces the following output:

```
[error 1] line 1 near "+": expecting:  ',' '.'
Error type 3                                        (*)
[error 2] line 1 near "abcde": expecting:  ',' '.'
Error type 4

Compilation error(s): 2
Compilation terminated
```

This demonstration shows that the parser is able to recover in all cases which can arise in the above example. However, we should note that the first error in input (4) is recovered through the rule `list error`, as can be seen from the trace facility (*), and not through the rule with `list error IDENTIFIER`, and so the third element of the list `abc` is discarded. Eliminating this alternative would result in unsuccessful termination of the parser in the case of a trailing error.

A larger demonstration, which shows the error reporting and recovery performance achievable in a realistic programming context, is given in appendix D.

7.3.5 The performance of the CORGI system

When considering the performance of a system, the most critical aspect is the time it requires to execute (since space is no longer at a premium on most modern computer systems). We have considered the time taken by CORGI to generate a parser using test data consisting of both artificial examples, and grammar rules for real programming languages such as Pascal, Modula-2, subset of C, MSL. The goal of these tests is to show how the time taken relates to the number of rules in the grammar.

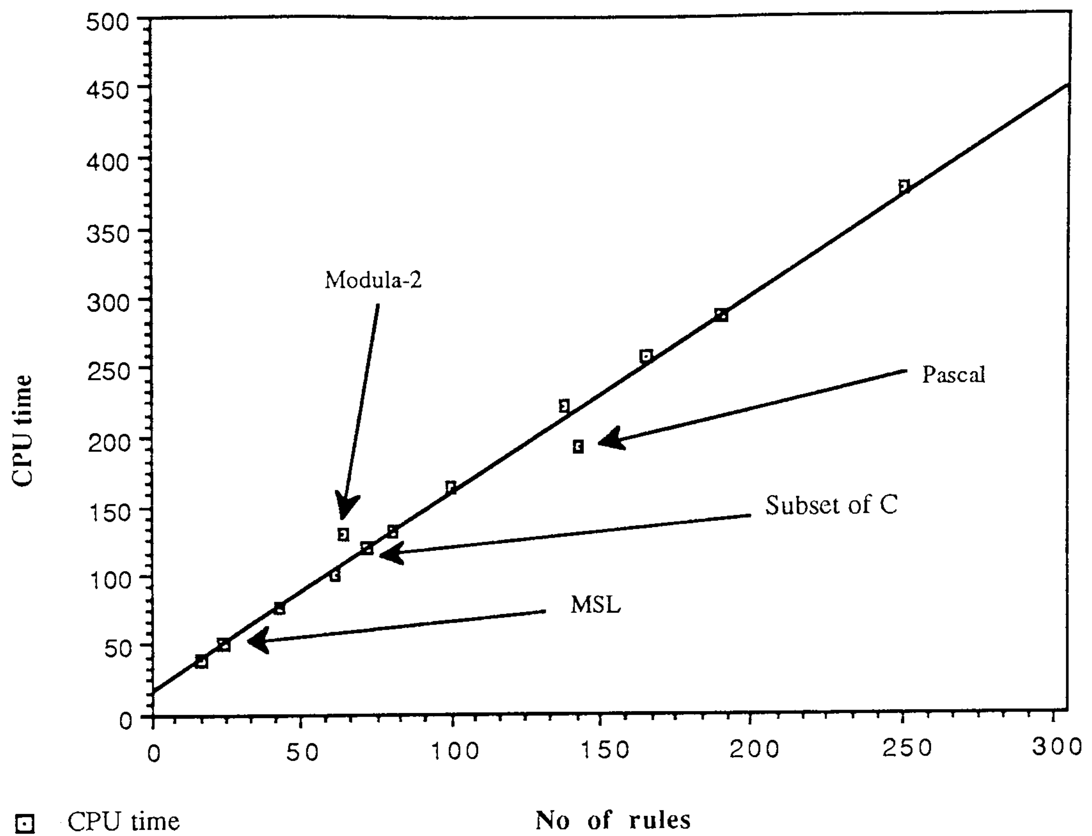


Figure 7.1 CORGI performance

Figure 7.1 shows the results of a number of test runs; points which are not annotated refer to artificial data. CPU time was measured using the Unix /lib/time command, and is the sum of the time used in both "user" and "system" mode.

It should be noted that where points lie off of the plotted regression line this may be explained by the level of complexity contained in each rule. A rule is considered more complex if it consists of a large number of alternatives, or has considerable repetition, optional and grouping constructs. Thus rules for Pascal were in fact less complex than those used as an artificial test case, whereas the rules for Modula-2 were more complex.

Of further interest is the time taken for a generated parser to process a number of test source programs. In Figure 7.2 we present a comparison of the performance of a CORGI-generated parser against one produced from hand-written lex and yacc specifications and one completely hand-coded. The hand-coded version is a top-down parser (since it not reasonable to hand-code bottom-up parsers) unlike the generated ones. For accuracy, the results were obtained by timing 10 runs and dividing.

Source Program	Hand-written version (seconds)	Lex and yacc generated version (seconds)	CORGI generated version (seconds)
msl.prog1	0.07	0.07	0.09
msl.prog2	0.07	0.07	0.09
msl.prog3	0.07	0.08	0.09
msl.prog4	0.11	0.16	0.16
msl.prog5	0.14	0.21	0.23
msl.prog6	0.21	0.33	0.35

Figure 7.2 Time comparisons for three versions of the parser

7.3.6 Compiler construction using the CORGI system

In this section we shall see how we would use the output generated by CORGI to write the rest of the compiler, namely the semantic analysis and code generation. For this demonstration we only used MSL as test data because it is reasonably small and simple but still sufficient in scope to give a realistic illustration.

As mentioned in chapter 5, the CORGI system may be used in two ways, which we termed deferred semantic reduction and direct semantic reduction. In both situations the CORGI system generates the following files:

- `lex.spec` : contains the lex specification for the given language
- `yacc.spec` : contains the yacc specification for the given language
- `syntree.h` : contains the C declaration of the types needed for the abstract syntax tree for the source program
- `writer.c` : contains a set of C routines which write a linear representation of the source program to a file
- `reader.c` : contains a set of C routines which read in the linear representation of the source program and reconstruct the tree in memory.
- `walker.c` : contains the C walker routines which visit each node of the tree to perform user's code.
- some further auxiliary routines needed by the above files.

These files are used in two ways depending on whether the compiler writer is using deferred semantic reduction or direct semantic reduction as described below.

Deferred semantic reduction

In this approach when the parse tree has been built it is written to a file called `tree_dat`, using a linear representation which is human-readable. This tree is later read back in and the tree is re-built in memory using the generated reader routines. Once the tree has been reconstructed, the walker routines are used to visit its nodes executing user-supplied code at every node. The usage of this approach involves the following steps:

- the execution of the CORGI system generates a parser called PARSER, which is achieved by using the following command:

```
CORGI      <filename>
```

  where `filename` is a file that contains the CORGI specification.

- the generated parser is then used to parse an input written in the given language:

```
PARSER  <   <input>
```

  This command will parse the input, produce a linear representation of the tree and will output one of the following set of messages:

```
•• Compilation error(s): <n° of errors found>
   Compilation terminated

•• Compilation error(s): <n° of errors found>
   Compilation aborted
```

- the walker routines generated by CORGI must have been augmented with user's code (eg. semantic actions and code generation).

- since deferred semantic reduction is used, the walker is not immediately called by the parser. The abstract syntax tree is written in linear form to a file, and so the generated reader routines should be used to reconstruct the tree from this file. When the tree has been reconstructed, a call should be made to the "top-level" walker routine, which causes calls to be made to further walker routines to traverse the tree, performing user-supplied semantic analysis and code generation functions.

  A model example of a skeletal main program for achieving this is supplied in a file `backend.c` and takes the following form:

```
#include  "syntree.h"      /* the generated syntax tree     */
#include  "reader.c"       /* the generated reader routines */
#include  "walker.c"       /* the generated walker routines */

/*  declaration of constants              */
/*  declaration of global variables   */

main()
{
PROGRAM_TYPE   ptr;

/* declaration and initialisation of variable */

  ptr = read_Program();      /* read the tree from a file */

  printf("\n*** Building the tree successfully completed ***\n");

  walk_Program(ptr);   /* visit the tree nodes */

/* print out results */

}/* end of main() */
```

A shell script is also provided, called `backend` which will compile and run the resulting C program. This performs the following:

```
cc -o run.x  backend.c
run.x
```

Direct semantic reduction

If direct semantic reduction is used, the parser builds an abstract syntax tree, but instead of writing this to a file, it calls the "top-level" walker routine as soon as parsing is complete. The set of generated walker routines must first have been augmented to perform whatever further processing is required. Use of the CORGI system in this manner takes the following form:

- use of -D option when invoking CORGI to generate the C files, thus:

```
CORGI  -D  <filename>
```

This will result in the generation of lex and yacc input specifications from the rules contained in the supplied file, together with walker routines for the associated abstract syntax tree.

- augment the walker routines with semantic analysis and code generation functions at appropriate nodes in the AST.

- use the provided shell script `create` which compiles and links the appropriate files and libraries as shown by the following commands:

```
cc -c y.tab.c
cc -c lex.yy.c
cc -o PARSER y.tab.o lex.yy.o -lyacc -leval -lmknode
```

which will compile and link all the necessary components into an executable parser. The parser will include a call to the function walker() which makes a call to the "top-level" walker routine, once parsing is complete. This is generated as:

```
/*  declaration of constants              */
/*  declaration of global variables   */

walker(ptr)
PROGRAM_TYPE   ptr;
{

/* Declaration and initialisation of variable */

  walk_Program(ptr); /* visit the tree nodes */

/* print out results */
}/* end of walker() */
```

- thus when using the parser to compile a source program the following command is used:

```
PARSER  <   <input>
```

which will parse the given input, build the AST, and walk round the tree performing semantic analysis and code generation.

Semantic processing for MSL

As an example of what is involved in augmenting CORGI-generated walker routines to perform semantic analysis and code generation, let us consider the case of MSL. In MSL there is no type checking to be done, since a value of any type can be assigned to any variable. All variables are global and visible throughout the entire program, and do not require declaration. The only restriction is that a name can only be used for one purpose in a program - either as a variable name, a procedure name, or a parameter name (the same name cannot be used as a parameter in more than one procedure). According to the above semantic constraints, the following semantic errors may occur:

- procedure name already used
- formal parameter identifier already used
- CALL identifier not declared as a procedure name
- wrong number of actual parameters
- improper use of procedure name
- improper use of formal parameter
- procedure not declared.

A symbol table needs to be maintained to support these semantic checks (with an associated look-up function) and also to allow correct code generation. The generated code is in the form of an intermediate representation, suitable for a hypothetical machine called TM. The full TM instruction set is given by Elsworth (1989).

A complete listing of the generated compiler for MSL can be found in appendix B; the test programs used to run this compiler together with the output from the compiler are given in appendix C.

Practical testing went further than MSL. CORGI specifications for Pascal, Modula-2 and a subset of C were supplied to CORGI which produced a parser for each of these languages. A set of example programs written in the specified language were submitted to the generated parser which parsed these programs successfully.

Full CORGI specifications for Modula-2 and Pascal are given in appendix E together with their example programs taken from Koffman (1988) and Findlay & Watt (1981) respectively.

7.4 Summary

In this chapter, we have shown how CORGI was used to generate a full compiler for a simple programming language, namely MSL. For more complex programming languages such as Pascal, Ada, Modula-2 we have shown how the features of such languages are catered for. Full specifications for Pascal and Modula-2 were written and correct parsers were produced.

We also presented details of the correct operation of our error-recovery mechanism. Results of a number of test runs were shown, and it was demonstrated that the performance of a CORGI-generated parser is comparable to that of a parser produced using hand-written lex and yacc specifications.

We gave an example of how CORGI can be used to construct an entire compiler for MSL, and noted the limitations of the system.

Chapter 8

Conclusions

In this thesis we have addressed the issue of automatic construction of the compiler front-end, in a manner which promotes ease of use as well as correct, efficient output. This thesis consists of two major sections: the first section reviews the fundamental principles of programming language definition and implementation and investigates various approaches taken by researchers to the automation of the process of compiler design and construction. The second section presents a new compiler writing system and discusses its philosophy.

Chapter 2 reviewed the essential aspects of programming language definition and implementation, and formal techniques for hand-crafting a language compiler. In chapter 3, we gave a detailed and critical review of approaches taken by other researchers for designing and implementing tools which automate the software construction process, with particular emphasis on so-called "compiler-compilers". Chapter 4 presented a more in-depth account of the use of the best-known existing compiler-writing tools, namely lex and yacc; in this chapter we recounted our experience gained when developing a complete compiler for a small but representative programming language known as MSL. In chapter 5, we identified the need for a compiler-writing system which uses a "reference manual" grammar as its input, and produces a parser for the given language together with additional routines and data structures to be used in subsequent phases of the compiler. Chapter 6 continued a detailed descriptions of the CORGI system, developed to achieve the goals referred to in earlier chapters. Finally, in chapter 7 we presented comprehensive examples of how CORGI can be used to generate parsers for typical programming languages like Modula-2 and Pascal, and demonstrated the

construction of an entire compiler by using the small programming language MSL. We also gave results concerning the efficiency of a CORGI-generated compiler front-end, relative to both that produced by lex and yacc and a hand-written version.

In appendices B and C, we demonstrate a complete application of CORGI to a realistic example. In appendix D, we demonstrate the improved error handling we have implemented, and in appendix E, we show how CORGI may be successfully applied in generating compiler front-end for Modula-2 and Pascal. Finally appendix F contains a CORGI specification of the syntax of the CORGI specification. The CORGI software and the user manual can be obtained from the Department of Computer Science and Applied Mathematics, Aston University, Birmingham B4 7ET.

In concluding this thesis, we summarise the project which we have presented, in relation to other work in the same field of research.

The distinctive features of the CORGI system are that it:

- accepts a specification based directly on a "reference manual" grammar;
- generates automatically a lexical analyser;
- generates automatically a parser together with the data structure declarations needed for the abstract syntax tree;
- produces a set of routines to manipulate the abstract syntax tree in memory and input/output;
- builds the tree and maintains it with lexical attributes required for later processing (eg. semantic analysis and code generation);
- provides a set of library routines for the evaluation of attributes of the lexemes of the language (this library may be extended by the user);
- provides automatically an error recovery mechanism.

CORGI possesses certain novel features which distinguish it from previous compiler-writing systems. To our knowledge, no system based on context-free grammars automatically generates code to process the abstract syntax tree. CORGI not only produces the necessary data structure declarations for creating such a tree during parsing, but it also generates functions to build it, perform input/output to and from permanent storage, and to walk around the tree during semantic analysis and code generation. The compiler-writer is then only required to insert code to perform the latter two stages of the compilation process into generated functions which are guaranteed to be well-formed, and which reflect the structure of the grammar used for parsing. In other systems, it is necessary to consider the detailed structure of the abstract syntax tree, and to write its associated manipulation functions.

There exist tools based on attribute grammars which do generate complete front-ends. However the reason for introducing attributions is to specify semantic aspects and this leads to great additional complication of the language description notation. But with CORGI we get the abstract syntax tree while still retaining the much more palatable simplicity of context-free notation.

CORGI can be used to merely generate a recognizer for a language, whilst the front-end is being tested, and can then be used to complete the first phase of compilation by building the abstract syntax tree. The error-recovery mechanism can be omitted during testing, and included simply by specifying a flag to CORGI at the command-line level.

Rather than re-building from scratch, involving vast effort, we have taken reasonably effective existing software, namely lex and yacc, and built a new shell around it to provide an improved facility. This of course involves some loss of

computational efficiency, but we have shown it to be a satisfactory approach with significant advantages.

Through a series of tests, we showed that the performance of the CORGI system compares favourably with hand-written examples, and that the time taken to process a CORGI specification rises only linearly as the number of rules increases. The additional time taken by a CORGI -generated parser for the test language MSL, over a parser generated from hand-written lex and yacc specifications, was shown to be quite tolerable. We also demonstrated that a CORGI-generated compiler for MSL was not significantly inferior to a carefully coded version written directly in C. It can be seen from the entire compiler produced for MSL, and parsers produced for Pascal and Modula-2, that CORGI is an easy-to-use, efficient tool for compiler construction.

Because we have chosen to implement CORGI "on top of" lex and yacc, it might be said that CORGI in effect provides an improved interface to lex and yacc. We therefore now look at recent related work concerning lex and yacc.

It has long been noted that lex and yacc have a number of major deficiencies, despite their obvious popularity; however, only a few of the critics of these tools have implemented possible improvements. One of these deficiencies has been identified in its error reports and messages. Schreiner & Friedman (1985) have suggested an improvement to the error reporting of the lex and yacc environment which was found to be adequate and hence adapted to our system. Although the error-recovery mechanism provided by yacc is technically adequate to deal with errors, its effective use requires a detailed understanding of its internal operation. For example, the user is responsible for the placement of actions which deal with the error-recovery. Schreiner & Friedman (1985) have also suggested a mechanism via which user can effectively deal with the placement of these actions without causing any shift/reduce

or reduce/reduce conflicts in the generated parser. In CORGI, we have integrated these ideas into the yacc specification which the system produces, rather than requiring the user to perform this by hand. Thus CORGI succeeds in improving error diagnosis and error-recovery in the resulting parser, without undue effort on the part of the user. Some other interesting work was carried out by Yang *et al.* (1988) in the error handling area. They developed SERCC which is based on yacc with the addition of systematic error recovery capability. It is a yacc-compatible experimental tool based on LALR(1) grammar. The philosophy of such systematic error-recovery is that users do not have to adjust their input grammars for error recovery purposes, unlike yacc. However, SERCC generates more states for error handling and also the size of the parsing tables generated is more than doubled due to the lack of structures for compacting these tables.

In a recent paper (Park, 1988), Joseph Park describes a system designed to be a preprocessor for yacc. The language y+ is used as an input notation allowing the user to specify actions, for building an AST, to be performed during parsing, and these actions are translated into C in a yacc specification. This system, however, still gives the user the responsibility to write his actions for the AST construction in a programming language (albeit a special-purpose one) and to code all the routines to manipulate this tree which is a burdensome task since it is repetitious, tedious and therefore highly error-prone. As this task is completely regular, it can in fact be fully automated, as has been provided in CORGI.

Although there have been the above attempts to improve yacc, we find that as far as lex is concerned, there appears to have been no other work designed to enhance its user interface. For example, when using lex in the normal way, the compiler-writer is obliged to deal with the evaluation of lexemes such as string literals and numeric constants himself. However, CORGI relieves him of this burden by automatically

inserting calls to library evaluation functions for this purpose. The library functions cover most modern programming language lexemes, but the user can add his own if they prove insufficient.

We thus find that other work on lex and yacc has concentrated on particular issues, rather than the overall objective of producing compiler front-ends as effectively as possible. With CORGI, we believe that we have met the main objectives of the research mentioned above, as well as addressing the overall problem. Moreover, CORGI has automated the application of improved error handling and AST construction, rather than leaving these tasks to the compiler writer.

Although the CORGI system as originally conceived has been completed, the work could profitably be taken further to incorporate certain additional features, as follows:

- At present the CORGI system supports only compiler front-end construction. It would naturally be desirable to allow CORGI specifications to be further augmented so as to include specification of semantic actions. The actions might be given in a suitable formal notation or in C code but in either case should lead to the automatic incorporation of semantic actions into the generated tree-walking routines. This enhancement would, however, involve substantial further research.

- As CORGI stands at present, a difficulty arises if the CORGI input is altered after the user has augmented the generated walker routines with his own code. When the revised CORGI input is processed, new walker routines will be generated, perhaps with very minor changes but the work of editing these to include the user's code will have to be repeated, which is clearly undesirable. This problem would be solved if the previous enhancement is

made, but otherwise, it would be useful to have some form of "intelligent" editor which would allow the edits involved in augmenting the walker routines to be recorded and re-used at a later stage.

- Cater for some modern languages such as occam and Miranda which use indentation to specify block structure. One suggestion to do this is to arrange to replace change-of-indentation by a symbol which is returned by the lexical analyser to the parser to be used as a token, with significance similar to `begin` or `end` in Pascal.

Overall, we feel that CORGI represents a significant advance in the field of compiler-compilers. It automatically incorporates the best-known techniques for using lex and yacc from a convenient reference manual grammar notation, and provides the compiler-writer with a firm framework on which to base the semantic analysis and code generation phases of the compiler.

References

Aho, A. V. (1980), 'Translator Writing Systems: Where Do They Now Stand?', *Computer* **13**, pp 9-14.

Aho, A. V., Sethi, R. & Ullman, J. D. (1986), *Compilers, Principles, Techniques, and Tools*, Wokingham, England: Addison-Wesley.

Aho, A. V. & Ullman, J. D. (1972), *The theory of Parsing, Translation, and Compiling1: Parsing*, Englewood Cliffs, N. J.: Prentice-Hall.

Aho, A. V. & Johnson, S. C. (1974), 'LR parsing', *Computing Survey* **6**(2), pp 99-124.

Asbrock, B., Kastens, U. & Zimmermann, E. (1981), 'Generating an efficient compiler front-end', Universitat Karlsruhe, Fakultat Fur Informatik, Bericht **17**.

Atteson, K., Lorenz, M. & Dowling, W. F. (1989), 'NAPL: A solution to the student compiler project problem', *SIGPLAN Notices* **24**(3), pp 57-66.

Backus, J. W. (1960), 'Report on the Algorithmic Language Algol60', *Communications of the ACM* **3** (5).

Berg, A., Bocking, D. A., Peachey, D. R., Sorenson, P. G., Tremblay, J. P. & Wald, J. A. (1984), 'VATS-The Visible Attributed Translation System', Technical Report 84-19, Dept. of Computer Science, University of Saskatchewan.

Bird, P. L. (1982), 'An Implementation of a Code Generator Specification Language for Table Driven Code Generators', *Proceedings of the SIGLAN Symposium on Compiler Construction*, Boston: Mass., pp 44-55.

Bjorner, D. & Jones, C. (1978), 'The Vienna Development Method: the metalanguage', *In Lecture Notes in Computer Science* **61**, New York: Springler-Verlag.

Bochmann, G. V. (1976), 'Semantic evaluation from left to right', *Communications of the ACM* **19**(2), pp 55-62.

Bochmann, G. V. & Ward, P. (1978), 'Compiler Writing System for Attribute Grammars', *The Computer Journal* **21**(2), pp 144-148.

Branquart, P., Cardinael, J. P., Lewi, J., Delescaille, J. P. & Van Begin, M. (1977), 'A simple translation automaton allowing the generation of optimized code', In Ershov, A. & Koster, C. H. A. (eds.), *Methods of Algorithmic Language Implementation - Proceedings of a workshop, Lecture Notes in Computer Science* **47**, Berlin: Springer-Verlag, pp 209-217.

British Standards Institution, (1982), 'The Pascal standard', BS6192.

Brooker, R. & Morris, D. (1962), 'A general translation program for phrase structure languages', *Journal ACM* **9**, pp 1-10.

Brooker, R. (1963), 'The Compiler Compiler', *Annual Review of Automatic Programming III*, pp 229-275.

Bruno, J. & Burkhard, W. A. (1970), 'A circularity test for interpreted grammars', Technical Report **88**, Computer Science Laboratory, Dept. of Electrical Engineering, Princeton University, Princeton, N. J.

Cattell, R. G. (1980), 'Automatic Derivation of Code Generators from Machine Descriptions', *ACM Transactions on Programming Languages and Systems* **2**(2), pp 173-190.

Cattell, R. G., Newcomer, J. M. & Leverett, B. W. (1979), 'Code generation in a machine-independent compiler', *Proceeding of the SIGPLAN Symposium on Compiler Construction,* Denver, Colorado, pp 65-75.

Chomsky, N. (1959), 'On Certain Formal Properties of Grammar', *Information and Control* **2**, pp 137-167.

Deschamp, Ph. (1980), 'Production de compilateurs à partir d'une description sémantique des languages de programmation: Le système PERLUETTE', Thesis I.N.P.L., Nancy.

DeRemer, F. L. (1969), 'Practical Translators for LR(k) languages', Ph. D. Thesis, M.I.T., Cambridge, Mass.

DeRemer, F. L. (1971), 'Simple LR(k) grammars', *Communications of the ACM* **14**(7), pp 453-460.

DeRemer, F. L. (1975), 'On compiler structure and translator writing systems', *Proceedings of the 8th International Conference on System Sciences*, University of Hawaii, pp 195-197.

Donegan, M. K., Noonan, R. E. & Feycock, S. (1979), 'A generator generator language', Proceeding of the SIGPLAN Symposium on Compiler Construction, Denver, Colorado, pp 58-64.

Dwyer, B. (1988),'Regular Right Part Programming Languages', *ACM SIGPLAN Notices* **23**(6), pp 140-144.

Earley, J. (1970), 'An efficient context-free parsing algorithm', *Communications of the ACM* **13**(2), pp 94-102.

Earley, J. (1975), 'Ambiguity and precedence in syntax description', *Acta Informatica* **4**(2), pp 183-192.

Elliott, I. B. (1988), 'The PRESTO system', *ACM SIGPLAN Notices* **23**(6), pp 39-48.

Elson, M. & Rake, S. T. (1970), 'Code-generation technique for large-language compilers', *IBM Systems Journal* **9**(3), pp 166-188.

Elsworth, E. F. (1989), unpublished PLI lecture notes, Computer Science Department, Aston University, Birmingham.

Fang, I. (1972) 'FOLDS, a declarative formal language definition system', STAN-CS-72-329, Computer Science Dept., Stanford University, Stanford, California.

Farrow, R. (1982), 'LINGUIST-86: Yet Another Translator Writing System Based On Attribute Grammars', *ACM SIGPLAN Notices* **17**(6), pp 160-171.

Feldman, S. I. (1979), 'Implementation of a portable Fortran 77 compiler using modern tools', *ACM SIGPLAN Notices* **14**(8), pp 98-106.

Findlay, W. & Watt, D. A. (1981), *Pascal: An introduction to Methodical Programming*, Bath, England: Pitman.

Fisher, C. N. & LeBlanc, R. J. (1988), *Crafting a compiler*, Menlo Park, California: Benjamin/Cummings, pp 765-789.

Franzen, H., Hoffman, B., Pohl, B. & Schmiedecke, I. (1977), 'The EAGLE Generator', *Proceedings of the 5th Informal Implementor's Interchange Conference*, Guidel, France, pp 397-420.

Fraser, C. W. (1977), 'A knowledge based code generator generator', *Proceedings of the Symposium on Artificial Intelligence and Programming Language*, pp 126-129.

Gammill, R. C. (1983), 'A portable lexical analyser writing system', PS117, the Rand corporation, Santa Monica, CA.

Ganzinger, H., Ripken, K. & Wilhelm, R. (1977), 'Automatic Generation of Optimizing Multipass Compilers', In Gilchrist, B. (ed.), *Information Processing 77, IFIP*, New York: North-Holland, pp 535-540.

Ganzinger, H., Giegerich, R., Möncke, U. & Wilhelm, R. (1982), 'A truly generative semantics-directed compiler generator', *ACM SIGPLAN Notices* **17**(6), pp 172-184.

Glanville, R. S. & Graham, S. L. (1978), 'A new method for compiler code genration', *Fifth ACM Symposium on Principles of Programming Languages, SIGLPAN-SIGACT*, pp 231-240.

Graham, S. L. (1980), 'Table-Driven Code Generation', *IEEE Computer* **13**(8), pp 25-34.

Gray, R. W. (1988), 'γ-GLA: A generator for lexical analyzers that programmers can use', *Summer USENIX'88*, San Francisco, pp 147-160.

Gries, D. (1971), *Compiler construction for digital computers*, New York: Wiley International.

Grune, D. & Jacobs, C. J. H. (1988), 'A Programmer-friendly LL(1) Parser Generator', *Software- Practice and Experience* **18**(1), pp 29-38.

Harbison, S. P. & Steele, G. L. (1984), *A C Reference Manual*, Englewood Cliffs, N.J: Prentice-Hall.

Hennessy, J. & Ganapathi, M. (1986), 'Advances in compiler technology', *Annual Reviews in Computer Science* **1**, pp 83-106.

Heuring, V. P. (1986), 'The Automatic Generation of Fast Lexical Analysers', *Software-Practice and Experience* **16**(9), pp 801-808.

Hindley, J. R. & Seldin, J. P. (1986), *Introduction to combinators and λ-calculus*, London: Cambridge University Press.

Hoare, C. A. R. & Wirth, N. (1973), 'An Axiomatic Definition of the Programming Language Pascal', *Acta Informatica* **2**, pp 335-355.

Hopcroft, J. E. & Ullman, J. D. (1969), *Formal Languages and Their Relation to Automata*, Reading, Mass: Addison-Wesley.

Hopgood, F. R. (1969), *Compiling techniques*, London: MacDonald.

Honeywell (1980), *Formal Definition of the Ada Programming Language, Honeywell Inc.*

Horspool, R. N. & Levy, M. R. (1987), 'Mkscan-An Interactive Scanner Generator', *Software-Practice and Experience* **17**(6), pp 369-378.

Irons, E. T. (1963), 'The structure and use of the syntax directed compiler', *Annual Review of Automatic Programming III.*

Jacobson, V. (1987), 'Tuning UNIX lex (abstract only)', Winter USENIX, Washington, D. C.

Jazayeri, M., Ogden, W. L. & Rounds, W. C. (1975), 'The intrinsicably exponential complexity of the circularity problem for attribute grammars', *Communications of the ACM* **18**(12), pp 697-706.

Jazayeri, M. & Walter, K. G. (1975), 'Alternating semantic evaluator', *Proceedings of the ACM 1975 Annual Conference*, pp 230-234.

Johnson, S. C. (1975), 'Yacc - Yet another compiler-compiler', Computer Science Technical Report **32**, Bell Laboratories, Murray Hill, N. J.

Johnson, S. C. & Lesk, M. E. (1978), 'UNIX Time-Sharing System: Language Development Tools', *The Bell System Technical Journal* **57**(6), pp 2155-2175.

Johnson, S. C. (1978), 'A portable compiler: Theory and Practice', *Fifth ACM Symposium on Principles of Programming Languages, SIGPLAN-SIGACT*, pp 97-104.

Johnson, S. C. (1979), 'A tour through the portable C compiler', (in the *Unix programmers manual*), Bell Laboratories, Murray Hill, N. J.

Johnson, W. L., Porter, J. H., Ackley, S. I. & Ross, D. T. (1968), 'Automatic generation of efficient lexical processors using finite state technique', *Communication of the ACM* **11**(12), pp 805-813.

Jones, N. D. & Schmidt, A. (1980), 'Compiler generation from denotational semantics', In Jones, N. D. (ed.), *Semantic-directed Compiler Generation: Lecture Notes in Computer Science* **94**, Aarhus, Denmark: Springer-Verlag, pp 71-93.

Kasami, T. (1965), 'An efficient recognition and syntax analysis algorithm for context-free languages', AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass.

Kastens, U. & Zimmermann, E. (1980), 'GAG - A generator based on attribute grammars', Fak. für Informatik, Universität Karlsruhe, Bericht **14**/80.

Kennedy, K. & Warren, S. K. (1976), 'Automatic generation of efficient evaluators for attribute grammars', *Conference Record of the 3th ACM Symposium on Principles of Programming Languages*, pp 32-49.

Kernighan, B. W & Ritchie, D. M . (1978), *The C Programming Language*, Englewood Cliffs, New Jersey: Prentice-hall

Kernighan, B. W & Ritchie, D. M . & Thompson, K. (1972), 'QED Text Editor', *Computer Science Technical Report* **5**, Bell Laboratories.

Koffman E. B. (1988), *Problem Solving and Structured Programming in Modula-2*, England: Addison-Wesley.

Koskimies, K., Räihä, K. & Sarjakoski, M. (1982), 'Compiler construction using Attribute Grammars', *ACM SIGPLAN Notices* **17**(6), pp 153-159.

Koskimies, K., Nurmi, O. & Paakki, J. (1988), 'The Design of a language Processor Generator', *Software- Practice and Experience* **18**(2), pp 107-135.

Koster, C. H. A. (1971a), 'Affix grammars', In Peck, J. E. L. (ed.), *Algol 68 Implementation*, Amsterdam: North-Holland, pp 95-109.

Koster, C. H. A. (1971b), 'CDL- A compiler-compiler', Report MR 127, Mathematisch Centrum, Amsterdam.

Koster, C. H. A. (1974a), 'Using the CDL compiler-compiler', in Bauer, F. L. & Eickel, J. (eds.), *Compiler Construction: Lecture Notes in Computer Science* **21**, New York: Springer-Verlag, pp 366-426.

Koster, C. H. A. (1974b), 'Two-level Grammars', In Goos, G. & Hartmanis, J. (eds), *Compiler Construction: An advanced cource*, Berlin: Springer-Verlag, pp 146-156.

Knuth, D. E. (1965), 'On the translation of language from left to right', *Information and Control* **8**(6), pp 607-639.

Knuth, D. E. (1968) 'Semantics of context-free languages', *Mathematical System Theory* **2**(2), pp 127-145.

Knuth, D. E. (1971), 'Top-down syntax analysis', *Acta Informatica* **1**(2), pp 79-110.

Krzemien, R. & Lukasiewicz, A. (1976), 'Automatic generation of lexical analyzers in a compiler-compiler', *Information Processing Letters* **4**(6), pp 165-168.

Lamb, D. A. (1987), 'IDL: sharing intermediate representations', *ACM Transactions on Programming Languages and Systems* **9**(3), pp 297-318.

Lecarme, O. & Bochmann, G. V. (1974), ' A truly usable and portable compiler writing system', *Proceeding IFIP congress*, pp 218-221.

Leinius, R. P. (1970), Error Detection and recovery for Syntax Directed Compiler Systems, Ph. D. Thesis, University of Wisconsin, Madison.

Lesk, M. E. (1975), 'Lex - a lexical analyzer generator', Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J.

Leverett, B. W., Cattell, R. G. G., Hobbs, S. O., Newcomer, J. M., Reiner, A. H., Schatz, B. R. & Wulf, W. A. (1980), 'An Overview of the Production-Quality Compiler-Compiler Projects', *IEEE Computer* **13**(8), pp 38-49.

Lewi, J., De Vlaminck, K., Huens, J. & Mertens, P. (1975), 'SLS/1: A translator writing system', *Proceeding of the 5th Annual Conference of the gessellschaft fur informatic: Lecture Notes in Computer Science* **34**, Berlin: Springer-Verlag, pp 627-641.

Lewis, P. M., Rosenkrantz, D. J. & Stearns, R. E. (1976), *Compiler Design Theory*, Reading, Mass: Addison-Wesley.

Lorho, B. (1977), 'Semantic attribute processing in the system DELTA', In Ershov, A. & Koster, C. H. A. (eds.), *Methods of Algorithmic Language Implementation: Lecture Notes in Computer Science* **47**, Berlin: Springer-Verlag, pp 21-40.

Lucas, P. & Walk, K. (1969), 'On the Formal Description of PL/I', *Annual Review in Automatic Programming*, **6**(3), pp 105-182.

McCarthy, J. & Painter, J. (1967), 'Correctness of a compiler for arithmetic expressions', *Proceedings Symposium in Applied Math.* **19**, pp 33-41.

McKeeman, W. M. (1965), 'Peephole optimization', *Communications of the ACM* **8**(7), pp 443-444.

McKeeman, W. M., Horning, J. J. & Wortman, D. B. (1970), *A Compiler Generator*, Englewood Cliffs, N. J.: Prentice-Hall.

McNaughton, R. & Yamada, H. (1960), 'Regular expressions and state graphs for automata', *IRE Transactions on Electronic Computers* **EC-9**(1), pp 38-47.

Meijer, H. & Nijholt, A. (1984), 'Translator Writing Tools since 1970: A selective bibliography', Faculty of Science, Dept. of Informatics, Nijmegen University, The Netherlands, pp 62-72.

Milton, D. R., Kirchhoff, L. W. & Rowland, B. R. (1979), 'An ALL(1) Compiler Generator', *ACM SIGPLAN Notices* **14**(8), pp 152-157.

Minsky, M. L., (1972), *Computation: finite and infinite machines*, London: Prentice-Hall.

Mitchell, J. G., Maybury, W. & Sweet, R. (1979), 'Meas Language Manual', CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, CA.

Mössenböck, H. (1986), 'Alex - A Simple and Efficient Scanner Generator', *ACM SIGPLAN Notices* **21**(5), pp 69-78.

Mosses, P. D. (1975), 'Mathematical semantics and compiler generation', Ph. D. Thesis, Oxford University.

Mosses, P. D. (1978), 'SIS a compiler generator system using denotational semantics', Reference manual, Dept. of Computer Science, University of Aarhus, Denmark.

Park, J. C. H. (1988), 'Y+: A Yacc Preprocessor for Certain Semantic Actions', *ACM SIGPLAN Notices* **23**(6), pp 97-106.

Paxson, V. (1988), FLEX(1) user's manual.

Pennello, T. J. & DeRemer, F. (1978), 'A forward move algorithm for LR error recovery', *Conference Rec. of the 5th Annual ACM Symposium on Principles of Programming Languages*.

Raskovsky, M. & Collier, Ph. (1980), 'From Standard to Implementation of Denotational Semantics', in Jones, N. D. (ed.), *Semantic-directed Compiler Generation: Lecture Notes in Computer Science* **94**, Aarhus, Denmark: Springer-Verlag, pp 95-139.

Räihä, K. -J. (1980), 'Bibliography on attribute grammars', *ACM SIGPLAN Notices* **15**(3), pp 35-44.

Räihä, K.-J., Saarinen, M. Soisalon-Soininen, E. & Tienari, M. (1978), 'The compiler writing system HLP', Dept. of Computer Science, Helsinki University, Finland, Report A-1978-2.

Reiss, S. P. (1987), 'Automatic Compiler Production: The Front End', *IEEE Transactions on Software Engineering*, Special Issue **13**(6), pp 609-627.

Reps, T., Teitelbaum, T. & Demers, A. (1983), 'Incremental Context-Dependent Analysis for Language-based editors', *Transaction on Programming Languages and Systems*, **5**(3), pp 449-477.

Reps, T. & Teitelbaum, T. (1987), 'Language Processing in Program Editors', *IEEE Computer* **20**(11), pp 29-40.

Roberts, G. H. (1988), 'OPG: An Optimizing Parser Generator', *ACM SIGPLAN Notices* **23**(6), pp 80-90.

Rogers, M. W. (1984), *Ada: Language, compilers and bibliography*, Cambridge, England: Cambridge University Press.

Rosen, S. (1964), 'A compiler-building system developed by Brooker and Morris', *Communication of the ACM* **7**(7), pp 403-414.

Saarinen, M. (1978), 'On constructing efficient evaluators for atrribute grammars', in Ausiello, G. & Bohm, C. (eds.), *Automata, Languages and Programming: Lecture Notes in Computer Science* 62, New York: Springer-Verlag, pp 382-397.

Schreiner, A. T. & Friedman, H. G. (1985), *Introduction to Compiler Construction with UNIX*, Englewood Cliffs, NJ: Prentice-Hall.

Schwanke, Lee. M. (1972), 'MACS - A programmable pre-processor with macrogeneration facilities', M.Sc Thesis, University of Colorado.

Scott, D. & Stratchey, C. (1971), 'Towards a mathematical semantics for computer languages', *Proceedings of the Symposium Computer and Automata*, New York: Polytechnic Press, pp 19-46.

Siewiorek, D. P., Bell, G. C. & Newell, A. (1982), *Computer Structures: Principles and Examples,* New York: McGraw-Hill.

Thatcher, J. W., Wagner, E. G. & Wright, J. B. (1980), 'More on advice on structuring compilers and proving them correct', in Jones, N. D. (ed.), *Semantic-directed Compiler Generation: Lecture Notes in Computer Science* **94**, Aarhus: Springer-Verlag, pp 165-188.

Tremblay, J. -P & Sorenson, P. G. (1985), *The theory and practice of compiler writing*, New York: McGraw-Hill.

Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. & Fisker, R. G. (1976), *Revised report on the algorithmic language ALGOL68,* New York: Springer-Verlag.

Waite, W. M. (1984), 'Some topics in lexical analysis', Arbeitspapier 112, Gesellschaft für Mathematik und Datenverarbeitung, Karlsruhe, Germany.

Waite, W. M. (1986), 'The cost of lexical analysis', *Software-Practice & Experience* **16**, pp 473-488.

Waite, W. M. & Carter, L. R. (1985), 'The cost of a generated parser', *Software-Practice and Experience* **15**, pp 221-239.

Waite, W. M., Heuring,V. P. & Gray, R. W. (1986), 'GLA - A Generator for Lexical Analyzers', Software Engineering Group Technical Report **86-1-1**, Dept. of Electrical & Computer Engineering, University of Colorado.

Warren, W. B., Kichenson, J. & Snodgrass, R. (1985), A tutorial introduction to using IDL, Softlab Document **1**, Computer Science Department, University of North Carolina.

Watt, D. A. (1974), 'LR parsing of affix grammars', Report **7**, Department of Computer Science, University of Glascow.

Watt, D. A. & Madsen, O. L. (1983), 'Extended Attribute Grammars', *The Computer Journal* **26**(2), pp 142-153.

Wilhelm, R., Ripken, K., Ciesinger, J., Ganzinger, H., Lahner, W. & Nollman, R. (1976), 'Design evaluation of the compiler generating system MUG1', Proceedings of the 2nd Conference on Software Engeneering, San Francisco, pp 571-576.

Wirth, N. (1968), 'PL 360 - a programming language for the 360 computers', *Journal of the ACM* **15**(1), pp 37-74.

Wirth, N. (1977), 'What can we do about the unecessary diversity of notation for syntactic definitions ?', *Communication of the ACM* **20**(11).

Yang, C., Yeh, H., Hu, M., Wang, C., Sheu, H. & Hwang, T. (1988), 'SERCC - Systematic Error Recovery Compiler Compiler', *Proceedings of IASTED International Symposium, Applied Informatics*, Grindelwald, Switzeland, pp 211-214.

Yeh, D. (1988), 'Automatic Construction of Incremental LR(1) - Parsers', *ACM SIGPLAN Notices* **22**(3), pp 33- 42.

Younger, D. H. (1967), 'Recognition and parsing of context-free languages in time n^3', *Information and Control* **10**(2), pp 189-208.

Appendix A

This appendix contains the hand-written version of lex and yacc specifications for MSL. It also contains the hand-written walker routines which perform the semantic analysis and code generation required for MSL. Thus the set of files given in this appendix are: `HW.lex.spec`, `HW.yacc.spec`, `HW.walker.c`, `HW.syntree.h`, `HW.define.h`. The prefix `HW` stands for Hand-Written versions.

```
        /***  MSL lex specification   - "HW.lex.spec"   ***/
        /***                 Hand-written                ***/

%{
#define      ENDTABLE(v)    (v-1 + sizeof v /sizeof v[0])
static       int       screen();
int          linenumber;
int          Toknum;
#include     "HW.syntree.h"
#include     "y.tab.h"
#include     <ctype.h>
char*        malloc();
%}

anybut_DQ_NL        [^\"\n]
digit               [0-9]
uplow_case          [A-Za-z]
anybut_NL           [^\n]
stringpic           {anybut_DQ_NL}
letter              {uplow_case}
text                "\""{stringpic}*"\""
int                 ({digit})({digit})*
idr                 ({letter})(({letter})|({digit}))*

%%

[ \t]          {      /* do nothing */              }
[\n]           {      linenumber++;                 }
"!"            {      return(EXCsym);               }
"("            {      return(OBsym);                }
")"            {      return(CBsym);                }
","            {      return(COMMAsym);             }
"."            {      return(DOTsym);               }
":="           {      return(ASSIGNsym);            }
"@"            {      return(INDIRsym);             }
"\""           {      return(DOUBLEQUsym);          }
"#"            {      return(HASHsym);              }
"%"            {
                      yylval.Vopr = PERCop;
                      return(PERCop);               }
"&"            {
                      yylval.Vopr = ANDop;
                      return(ANDop);                }
```

```
"*"             {
                        yylval.Vopr = MULTIop;
                        return(MULTIop);                }
"+"             {
                        yylval.Vopr = PLUSop;
                        return(PLUSop);                 }
"-"             {
                        yylval.Vopr = MINUSop;
                        return(MINUSop);                }

"/"             {
                        yylval.Vopr = DIVop;
                        return(DIVop);                  }

"<"             {
                        yylval.Vopr = LESSop;
                        return(LESSop);                 }
"<="            {
                        yylval.Vopr = LESSEQop;
                        return(LESSEQop);               }
"<>"            {
                        yylval.Vopr = NOTEQop;
                        return(NOTEQop);                }
"="             {
                        yylval.Vopr = EQUALop;
                        return(EQUALop);                }
">"             {
                        yylval.Vopr = GREATERop;
                        return(GREATERop);              }
">="            {
                        yylval.Vopr = GREATEREQop;
                        return(GREATEREQop);            }
{idr}           {
                        Toknum = screen();
                        if(Toknum == IDR)
                               yylval.Vidr = (char*)evaluate_idr();
                         else
                        {
                              upyytext();
                              if(!strcmp(yytext,"TRUE"))
                                    yylval.Vboolean = TRUESYM;
                              else
                              if(!strcmp(yytext,"FALSE"))
                                    yylval.Vboolean = FALSESYM;
                        }
                        return(Toknum);                 }

{int}           {       yylval.Vint = evaluate_dinary();
                        return(INT); }
{text}          {       yylval.Vtext = (char*)evaluate_Cstr();
                        return(TEXT);}

"--"{anynut_NL}*[\n]    { linenumber++; /* it is a comment */  }

.      { printf("\n!!! char <%c> is illegal here\n",yytext[0]);}

%%

      /**  reserved word table  **/

static  struct rwtable{             /* reserved word table */
```

```
        char  *rw_name;              /* representation      */
        int   rw_yylex;              /* yylex() value       */
        }rwtable[] = {
            "CALL",            CALLSYM,
            "DO",              DOSYM,
            "ELSE",            ELSESYM,
            "END",             ENDSYM,
            "FALSE",           FALSESYM,
            "FI",              FISYM,
            "IF",              IFSYM,
            "OD",              ODSYM,
            "PROC",            PROCSYM,
            "READ",            READSYM,
            "RESERVE",         RESERVESYM,
            "THEN",            THENSYM,
            "TRUE",            TRUESYM,
            "WHILE",           WHILESYM,
            "WRITE",           WRITESYM,
};

static  int  screen()
{
struct  rwtable  *low = rwtable,
                 *high = ENDTABLE(rwtable),
                 *place;
int   cond;
char  *pname;

   while(low <= high)
   {
      place = low + (high - low) / 2;
      pname = place -> rw_name;
      if((cond = streqv(pname)) < 0)
         low = place +1;
      else
      if(cond > 0)
         high = place -1;
      else  return(place -> rw_yylex);
   }
   return(IDR);
}

streqv(pname)
char  *pname;
{
extern  char  toupper_c();
char  *s, *t;

   s = pname;  t = yytext;
   for(;toupper_c(*s) == toupper_c(*t); s++,t++)
   if(*s == '\0')        return(0);
   return(toupper_c(*s) - toupper_c(*t));
}

upyytext()
{
extern  char  toupper_c();
char  *s, *t;

   t = yytext;
   while(*t != '\0')
```

```
   { *s = toupper_c(*t);  s++; t++;  }
   *s = '\0';
   yytext = s;
}

char  toupper_c(c)
char  c;
{
   if(islower(c))  return(toupper(c));
   return(c);
}
yywrap()  {    return;    }
yyerror()   { return(-1); }
```

```
       /***   MSL yacc specification  "HW.yacc.spec"   ***/
       /***                   Hand-written                ***/


%{
#include    "mknodes.h"
#include    "HW.walker.c"
%}
      /**  Types associated with grammar symbols  **/

%union{
int                             Vboolean;
int                             Vint;
char*                           Vidr;
char*                           Vtext;
int                             Vopr;
PROGRAM_TYPE                    VProgram;
PROCDECS_TYPE                   VProcDecs;
OP_RESERVE_TYPE                 VOp_Reserve;
PROCDECLIST_TYPE                VProcDeclist;
OP_PARAM_TYPE                   VOp_Param;
IDRLIST_TYPE                    VIdrlist;
IDRS_TYPE                       VIdrs;
SERIES_TYPE                     VSeries;
STMTS_TYPE                      VStmts;
OP_ELSE_TYPE                    VOp_Else;
OP_EXPRLIST_TYPE                VOp_Exprlist;
EXPRLIST_TYPE                   VExprlist;
EXPRS_TYPE                      VExprs;
STMT_TYPE                       VStmt;
ASSIGNST_TYPE                   VAssignst;
WHILEST_TYPE                    VWhilest;
IFST_TYPE                       VIfst;
CALLST_TYPE                     VCallst;
READST_TYPE                     VReadst;
int                             VOptionalplus;
int                             VOptionalhash;
WRITEST_TYPE                    VWritest;
READINLIST_TYPE                 VReadinlist;
WRITEOUTLIST_TYPE               VWriteoutlist;
STORELIST_TYPE                  VStorelist;
LISTEXPR_TYPE                   VListExpr;
EXPR_TYPE                       VExpr;
OPERAND_TYPE                    VOperand;
OPERANDS_TYPE                   VOperands;
```

```
STOREACCESS_TYPE              VStoreAccess;
OP_EXCL_TYPE                  VOp_Excl;
}

      /**  special character tokens  **/

%token     EXCsym             /*    EXCsym = "!"              */
%token     PERCop             /*    PERCop = "%"              */
%token     ANDop              /*    ANDop = "&"               */
%token     OBsym              /*    OBsym = "("               */
%token     CBsym              /*    CBsym = ")"               */
%token     MULTIop            /*    MULTIop = "*"             */
%token     PLUSop             /*    PLUSop = "+"              */
%token     COMMAsym           /*    COMMAsym = ","            */
%token     MINUSop            /*    MINUSop = "_"             */
%token     DOTsym             /*    DOTsym = "."              */
%token     DIVop              /*    DIVop = "/"               */
%token     ASSIGNsym          /*    ASSIGNsym = ":="          */
%token     LESSop             /*    LESSop = "<"              */
%token     LESSEQop           /*    LESSEQop = "<="           */
%token     NOTEQop            /*    NOTEQop = "<>"            */
%token     EQUALop            /*    EQUALop = "="             */
%token     GREATERop          /*    GREATERop = ">"           */
%token     GREATEREQop        /*    GREATEREQop = ">="        */
%token     INDIRsym           /*    INDIRsym =  "@"           */
%token     DOUBLEQUsym        /*    DOUBLEQUsym = "\""        */
%token     HASHsym            /*    HASHsym = "#"             */

      /**  lexeme tokens  **/

%token     IDR
%token     INT
%token     TEXT

      /**  The keyword tokens  **/

%token     CALLSYM
%token     DOSYM
%token     ELSESYM
%token     ENDSYM
%token     FALSESYM
%token     FISYM
%token     IFSYM
%token     ODSYM
%token     PROCSYM
%token     READSYM
%token     RESERVESYM
%token     THENSYM
%token     TRUESYM
%token     WHILESYM
%token     WRITESYM
%token     FALSESYM
%token     TRUESYM

      /** Operator precedence and associativity  **/

%left       PERCop  ANDop  MULTIop  PLUSop  MINUSop  DIVop  LESSop
            LESSEQop  NOTEQop EQUALop  GREATERop  GREATEREQop

      /**  Type declaration. **/
```

```
%type  <Vopr>               PERCop ANDop MULTIop PLUSop MINUSop
                            DIVop LESSop LESSEQop NOTEQop EQUALop
                            GREATERop GREATEREQop
%type  <Vboolean>           FALSESYM   TRUESYM
%type  <Vidr>               IDR
%type  <Vint>               INT
%type  <Vtext>              TEXT
%type  <VProgram>           Program
%type  <VProcDecs>          ProcDecs
%type  <VOp_Param>          Op_Param
%type  <VOp_Reserve>        Op_Reserve
%type  <VProcDeclist>       ProcDeclist
%type  <VIdrs>              Idrs
%type  <VSeries>            Series
%type  <VStmts>             Stmts
%type  <VOp_Else>           Op_Else
%type  <VOp_Exprlist>       Op_Exprlist
%type  <VExprs>             Exprs
%type  <VStmt>              Stmt
%type  <VAssignst>          Assignst
%type  <VWhilest>           Whilest
%type  <VIfst>              Ifst
%type  <VCallst>            Callst
%type  <VExprlist>          Exprlist
%type  <VIdrlist>           Idrlist
%type  <VReadst>            Readst
%type  <VOptionalplus>      Optionalplus
%type  <VOptionalhash>      Optionalhash
%type  <VWritest>           Writest
%type  <VReadinlist>        Readinlist
%type  <VWriteoutlist>      Writeoutlist
%type  <VStorelist>         Storelist
%type  <VListExpr>          ListExpr
%type  <VOperands>          Operands
%type  <VOp_Excl>           Op_Excl
%type  <VExpr>              Expr
%type  <VOperand>           Operand
%type  <VStoreAccess>       StoreAccess

%%

Start    :  Program           {  walker($1); };

Program : Op_Reserve ProcDecs Series DOTsym
          { $$ = (PROGRAM_TYPE)mknode(4,1,$1,$2,$3); };

Op_Reserve  : /*  empty  */  {  $$ = (OP_RESERVE_TYPE)NULL; }
            | RESERVESYM INT
            { $$ = (OP_RESERVE_TYPE)mknode(2,1,$2); };

ProcDecs    :  /* empty */      { $$ = (PROCDECS_TYPE)NULL; }
          |  ProcDecs ProcDeclist
          { $$ = (PROCDECS_TYPE)mknode(3,1,$1,$2); };

ProcDeclist : PROCSYM IDR Op_Param Series ENDSYM
            { $$ = (PROCDECLIST_TYPE)mknode(4,1,$2,$3,$4);};

Op_Param  : /* empty  */       { $$ = (OP_PARAM_TYPE)NULL; }
          | OBsym Idrlist CBsym
          { $$ = (OP_PARAM_TYPE)mknode(2,1,$2); }
```

```
Idrlist      : IDR Idrs { $$ = (IDRLIST_TYPE)mknode(3,1,$1,$2); };

Idrs  : /*  empty  */          { $$ = (IDRS_TYPE)NULL;   }
      | Idrs COMMAsym  IDR
      { $$ = (IDRS_TYPE)mknode(3,1,$1,$3); };

Series : Stmt Stmts     { $$ = (SERIES_TYPE)mknode(3,1,$1,$2); };

Stmt   : Assignst       { $$ = (STMT_TYPE)mknode(2,1,$1); }
           | Whilest    { $$ = (STMT_TYPE)mknode(2,2,$1); }
           | Ifst       { $$ = (STMT_TYPE)mknode(2,3,$1); }
           | Callst     { $$ = (STMT_TYPE)mknode(2,4,$1); }
           | Readst     { $$ = (STMT_TYPE)mknode(2,5,$1); }
           | Writest    { $$ = (STMT_TYPE)mknode(2,6,$1); };

Stmts : /*  empty  */   { $$ = (STMTS_TYPE)NULL;   }
        | Stmts Stmt    { $$ = (STMTS_TYPE)mknode(3,1,$1,$2); };

Assignst : StoreAccess ASSIGNsym Expr
           { $$ = (ASSIGNST_TYPE)mknode(3,1,$1,$3); };

Whilest : WHILESYM Expr DOSYM Series ODSYM
          { $$ = (WHILEST_TYPE)mknode(3,1,$2,$4); };

Ifst : IFSYM Expr THENSYM Series Op_Else  FISYM
       { $$ = (IFST_TYPE)mknode(1,$2,$4,$5); };

Op_Else  : /* empty */        { $$ = (OP_ELSE_TYPE)NULL; }
         | ELSESYM Series     { $$ = (OP_ELSE_TYPE)mknode(2,1,$2);};

Callst : CALLSYM IDR Op_Exprlist
         { $$ = (CALLST_TYPE)mknode(3,1,$2,$3); };

Op_Exprlist : /* empty */    { $$ = (OP_EXPRLIST_TYPE)NULL;   }
            | OBsym Exprlist CBsym
            { $$ = (OP_EXPRLIST_TYPE)mknode(2,1,$2); };

Exprlist : Expr Exprs
           { $$ = (EXPRLIST_TYPE)mknode(3,1,$1,$2); };

Exprs  :  /*  empty  */        { $$ = (EXPRS_TYPE)NULL;   }
       | Exprs COMMAsym Expr
       { $$ = (EXPRS_TYPE)mknode(3,1,$1,$3); };

Readst : READSYM Optionalplus Readinlist
         { $$ = (READST_TYPE)mknode(3,1,$2,$3); };

Writest : WRITESYM  Optionalplus Writeoutlist
          { $$ = (WRITEST_TYPE)mknode(3,1,$2,$3); };

Readinlist : Optionalhash StoreAccess Storelist
             { $$ = (READINLIST_TYPE)mknode(4,1,$1,$2,$3); };

Storelist  : /*  empty  */    { $$ = (STORELIST_TYPE)NULL; }
           | Storelist COMMAsym Optionalhash StoreAccess
           { $$ =  (STORELIST_TYPE)mknode(4,1,$1,$3,$4); };

Writeoutlist : Optionalhash Expr ListExpr
               { $$ = (WRITEOUTLIST_TYPE)mknode(4,1,$1,$2,$3); };

ListExpr : /*  empty  */      { $$ = (LISTEXPR_TYPE)NULL; }
```

```
          | ListExpr COMMAsym Optionalhash Expr
          { $$ = (LISTEXPR_TYPE)mknode(4,1,$1,$3,$4); };

Optionalplus : /*  empty  */        { $$ = 0; }
             |  PLUSop              { $$ = 1; };

Optionalhash : /*  empty  */        { $$ = 0; }
             |  HASHsym             { $$ = 1; };

Expr :  Operand Operands { $$ = (EXPR_TYPE)mknode(3,1,$1,$2); };

Operands    : /*  empty  */          { $$ = (OPERANDS_TYPE)NULL; }
      | Operands PERCop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,1,$1,$2,$3); }
      |  Operands ANDop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,2,$1,$2,$3); }
      |  Operands MULTIop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,3,$1,$2,$3); }
      |  Operands PLUSop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,4,$1,$2,$3); }
      |  Operands MINUSop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,5,$1,$2,$3); }
      |  Operands DIVop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,6,$1,$2,$3); }
      |  Operands LESSop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,7,$1,$2,$3); }
      |  Operands LESSEQop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,8,$1,$2,$3); }
      |  Operands NOTEQop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,9,$1,$2,$3); }
      |  Operands GREATERop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,10,$1,$2,$3);}
      | Operands GREATEREQop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,11,$1,$2,$3);}
      |  Operands EQUALop Operand
                { $$ = (OPERANDS_TYPE)mknode(4,12,$1,$2,$3);};

Operand : INT                  { $$ = (OPERAND_TYPE)mknode(2,1,$1); }
        | TEXT                 { $$ = (OPERAND_TYPE)mknode(2,2,$1); }
        | TRUESYM              { $$ = (OPERAND_TYPE)mknode(2,3,$1); }
        | FALSESYM             { $$ = (OPERAND_TYPE)mknode(2,4,$1); }
        | INDIRsym IDR         { $$ = (OPERAND_TYPE)mknode(2,5,$2); }
        | StoreAccess          { $$ = (OPERAND_TYPE)mknode(2,6,$1); }
        | OBsym Expr CBsym { $$ = (OPERAND_TYPE)mknode(2,7,$2); };

StoreAccess : IDR Op_Excl
            { $$ = (STOREACCESS_TYPE)mknode(3,1,$1,$2); };

Op_Excl   : /*  empty  */  { $$ = (OP_EXCL_TYPE)NULL; }
          | EXCsym Operand { $$ = (OP_EXCL_TYPE)mknode(2,1,$2); };

%%
main()
{
#ifdef LEXDEBUG
  debug = 1;          /* debug = 1 for debugging and 0 otherwise */
#endif
extern  int  yynerrs;
int flag = 0;

     flag = yyparse();
```

```
    printf("Compilation error(s): %d\n", yynerrs);
    if(flag)
       printf("Compilation aborted\n");
    else
       printf("Compilation terminated\n");
}
```

```
/***  The declaration of MSL AST "HW.syntree.h" ***/
/***                 Hand-written               ***/


#define         NIL             0
#define         STRUCT          struct
#define         UNION           union

typedef  struct  Program_type          *  PROGRAM_TYPE;
typedef  struct  ProcDecs_type         *  PROCDECS_TYPE;
typedef  struct  Op_Param_type         *  OP_PARAM_TYPE;
typedef  struct  Op_Reserve_type       *  OP_RESERVE_TYPE;
typedef  struct  ProcDeclist_type      *  PROCDECLIST_TYPE;
typedef  struct  Series_type           *  SERIES_TYPE;
typedef  struct  Stmts_type            *  STMTS_TYPE;
typedef  struct  Op_Else_type          *  OP_ELSE_TYPE;
typedef  struct  Op_Exprlist_type      *  OP_EXPRLIST_TYPE;
typedef  struct  Exprs_type            *  EXPRS_TYPE;
typedef  struct  Stmt_type             *  STMT_TYPE;
typedef  struct  Assignst_type         *  ASSIGNST_TYPE;
typedef  struct  Whilest_type          *  WHILEST_TYPE;
typedef  struct  Ifst_type             *  IFST_TYPE;
typedef  struct  Callst_type           *  CALLST_TYPE;
typedef  struct  Exprlist_type         *  EXPRLIST_TYPE;
typedef  struct  Idrlist_type          *  IDRLIST_TYPE;
typedef  struct  Idrs_type             *  IDRS_TYPE;
typedef  struct  Readst_type           *  READST_TYPE;
typedef  struct  Writest_type          *  WRITEST_TYPE;
typedef  struct  Readinlist_type       *  READINLIST_TYPE;
typedef  struct  Writeoutlist_type     *  WRITEOUTLIST_TYPE;
typedef  struct  Storelist_type        *  STORELIST_TYPE;
typedef  struct  ListExpr_type         *  LISTEXPR_TYPE;
typedef  struct  Expr_type             *  EXPR_TYPE;
typedef  struct  Operand_type          *  OPERAND_TYPE;
typedef  struct  Operands_type         *  OPERANDS_TYPE;
typedef  struct  Op_Excl_type          *  OP_EXCL_TYPE;
typedef  struct  StoreAccess_type      *  STOREACCESS_TYPE;

      /**   Declaration of data structure types   **/

STRUCT  Program_type{
      int               type;
      OP_RESERVE_TYPE   Op_Reserve;
      PROCDECS_TYPE     ProcDecs;
      SERIES_TYPE       Series;
};
STRUCT  Op_Reserve_type{
      int   type;
      int   intsym;
};
STRUCT  ProcDecs_type{
      int               type;
      PROCDECS_TYPE     ProcDecs;
      PROCDECLIST_TYPE  ProcDeclist;
};
STRUCT  ProcDeclist_type{
      int               type;
      char*             idr;
      OP_PARAM_TYPE     Op_Param;
      SERIES_TYPE       Series;
};
```

```
STRUCT  Op_Param_type{
        int           type;
        IDRLIST_TYPE Idrlist;
};

STRUCT  Idrlist_type{
       int          type;
       char*        idr;
       IDRS_TYPE    Idrs;
};
STRUCT  Idrs_type{
       int          type;
       IDRS_TYPE    Idrs;
       char*        idr;
};
STRUCT  Series_type{
       int          type;
       STMT_TYPE    Stmt;
       STMTS_TYPE   Stmts;
};
STRUCT  Stmts_type{
       int          type;
       STMTS_TYPE   Stmts;
       STMT_TYPE    Stmt;
};
STRUCT  Stmt_type{
       int      type;
       UNION    {
       ASSIGNST_TYPE           Assignst;
       WHILEST_TYPE            Whilest;
       IFST_TYPE               Ifst;
       CALLST_TYPE             Callst;
       READST_TYPE             Readst;
       WRITEST_TYPE            Writest;
       }RIGHTSIDE;
};
STRUCT  Assignst_type{
       int                type;
       STOREACCESS_TYPE   StoreAccess;
       EXPR_TYPE          Expr;
};
STRUCT  Whilest_type{
       int             type;
       EXPR_TYPE       Expr;
       SERIES_TYPE     Series;
};
STRUCT  Ifst_type{
        int            type;
        EXPR_TYPE      Expr;
        SERIES_TYPE    Series;
        OP_ELSE_TYPE   Op_Else;
};
STRUCT  Op_Else_type{
       int             type;
       SERIES_TYPE     Series;
};
STRUCT  Callst_type{
       int                type;
       char*              idr;
       OP_EXPRLIST_TYPE   Op_Exprlist;
};
```

```
STRUCT  Op_Exprlist_type{
      int                 type;
      EXPRLIST_TYPE       Exprlist;
};
STRUCT  Exprlist_type{
      int          type;
      EXPR_TYPE    Expr;
      EXPRS_TYPE   Exprs;
};
STRUCT  Exprs_type{
      int          type;
      EXPRS_TYPE   Exprs;
      EXPR_TYPE    Expr;
};
STRUCT  Readst_type{
      int                 type;
      int                 Optionalplus;
      READINLIST_TYPE     Readinlist;
};
STRUCT  Readinlist_type{
      int                 type;
      int                 Optionalhash;
      STOREACCESS_TYPE    StoreAccess;
      STORELIST_TYPE      Storelist;
};
STRUCT  Storelist_type{
      int                 type;
      STORELIST_TYPE      Storelist;
      int                 Optionalhash;
      STOREACCESS_TYPE    StoreAccess;
};
STRUCT  Writest_type{
      int                     type;
      int                     Optionalplus;
      WRITEOUTLIST_TYPE       Writeoutlist;
};
STRUCT  Writeoutlist_type{
      int                 type;
      int                 Optionalhash;
      EXPR_TYPE           Expr;
      LISTEXPR_TYPE       ListExpr;
};
STRUCT  ListExpr_type{
      int                 type;
      LISTEXPR_TYPE       ListExpr;
      int                 Optionalhash;
      EXPR_TYPE           Expr;
};
STRUCT  Expr_type{
      int                 type;
      OPERAND_TYPE        Operand;
      OPERANDS_TYPE       Operands;
};
STRUCT  Operands_type{
      int      type;
      OPERANDS_TYPE       Operands;
      int                 opr;
      OPERAND_TYPE        Operand;
};
STRUCT  Operand_type{
      int      type;
```

```
      UNION    {
      int                 intsym;
      char*               text;
      int                 truesym;
      int                 falsesym;
      char*               idr;
      STOREACCESS_TYPE    StoreAccess;
      EXPR_TYPE           Expr;
      }RIGHTSIDE;
};
STRUCT  StoreAccess_type{
       int             type;
       char*           idr;
       OP_EXCL_TYPE    Op_Excl;
};
STRUCT  Op_Excl_type{
       int             type;
       OPERAND_TYPE    Operand;
};
```

```
         /***   MSL walker routines   -   "HW.walker.c"   ***/
         /***                   Hand-written                ***/
```

```
#include     "HW.syntree.h"
#include     "HW.define.h"
#define      NULL   0

      /** global variables **/

int    PSused;      /* index to highest used program store loca. */
int    DSused;      /* index to highest used data store location */
int    numidrs;           /* number of idrs found     */
int    numfps;            /* number of formal para. */
int    numaps;            /* number of actual para. */
int    STindex;           /* symbol table index     */
int    currentprocref;    /* should be <= maxidrs   */
int    numerrs;           /* number of errors found */
char   *idrchars;         /* holds the identifier   */
char   *malloc();
BOOLEAN      textual;

      /** Start of the program     **/

walker(ptr)
PROGRAM_TYPE   ptr;
{
int   i;

#ifdef  WALKDEBUG
printf("In walker()\n");
#endif

  numidrs = numfps = numaps = STindex = 0;
  numerrs = currentprocref = 0;
  PSused = 0; DSused = -1;

  for(i=1; i<=Maxidrs; i++)
```

```
  {
      symtab[i].idrname = NULL;
      symtab[i].class = 0;
      symtab[i].idrusage.var_rtsl = -1;
      symtab[i].idrusage.PN.entryaddr = -1;
      symtab[i].idrusage.PN.numparam = 0;
      symtab[i].idrusage.FP.procref = -1;
      symtab[i].idrusage.FP.paramnum = 0;
  }

  for(i=0; i<PSsize; i++)
  {
      DS[i] = 0; PS[i] = 0;
  }

  walk_Program(ptr);

  printf("\n**** MSL compilation complete _ \
            %d errors reported ****\n",numerrs);
  ListSymTab();   /*  print out the symbol table content    */
  ListTMcode();   /*  print out the generated code          */
}/* end of walker() */

walk_Program(ptr)
PROGRAM_TYPE  ptr;
{
int    jmain;
       if(ptr -> ProcDecs)
       {
           cg2(J,0);    jmain = PSused;
       }
       walk_Op_Reserve( ptr -> Op_Reserve);
       walk_ProcDecs(ptr -> ProcDecs);
       if(ptr -> ProcDecs)
             PS[jmain] = PSused+1;
       walk_Series(ptr -> Series);
       cg1(HALT);         /* end of msl program  */
}

walk_Op_Reserve(ptr)
OP_RESERVE_TYPE  ptr;
{
      if(ptr)    DSused = ptr -> intsym;
}

walk_ProcDecs (ptr)
PROCDECS_TYPE  ptr;
{
       if(ptr)
       {
          walk_ProcDecs(ptr -> ProcDecs);
          walk_ProcDeclist(ptr -> ProcDeclist);
       }
}

walk_ProcDeclist (ptr)
PROCDECLIST_TYPE  ptr;
{
       idrchars = malloc(strlen(ptr -> idr)+1);
       strcpy(idrchars,ptr -> idr);
       checkprocidr();
```

```
        walk_Op_Param (ptr -> Op_Param);
        if(currentprocref != 0)
        if(symtab[currentprocref].class == Procname)
              symtab[currentprocref].idrusage.PN.numparam = numfps;
        numfps = 0;
        walk_Series(ptr -> Series);  cg1(RTN);
        currentprocref = 0;
}

walk_Op_Param (ptr)
OP_PARAM_TYPE  ptr;
{
        if(ptr) walk_Idrlist(ptr -> Idrlist);
}

walk_Idrlist (ptr)
IDRLIST_TYPE  ptr;
{
        idrchars = malloc(strlen(ptr -> idr)+1);
        strcpy(idrchars,ptr -> idr);
        numfps++;
        checkFPidr();
        walk_Idrs(ptr -> Idrs);
}

walk_Idrs (ptr)
IDRS_TYPE  ptr;
{
        if(ptr)
        {
            walk_Idrs(ptr -> Idrs);
            idrchars = malloc(strlen(ptr -> idr)+1);
            strcpy(idrchars,ptr -> idr);
            numfps++;
            checkFPidr();
        }
}

walk_Series (ptr)
SERIES_TYPE  ptr;
{
        if(ptr)
        {
            walk_Stmt(ptr -> Stmt);
            walk_Stmts(ptr -> Stmts);
        }
}

walk_Stmt (ptr)
STMT_TYPE  ptr;
{
switch (ptr -> type) {

case  1  :  walk_Assignst(ptr -> RIGHTSIDE.Assignst);     break;
case  2  :  walk_Whilest(ptr -> RIGHTSIDE.Whilest);       break;
case  3  :  walk_Ifst(ptr -> RIGHTSIDE.Ifst);             break;
case  4  :  walk_Callst(ptr -> RIGHTSIDE.Callst);         break;
case  5  :  walk_Readst(ptr -> RIGHTSIDE.Readst);         break;
case  6  :  walk_Writest(ptr -> RIGHTSIDE.Writest);       break;
default  :  printf("ERROR - wrong alternative number\n"); break;
}/* end of switch  */
```

```
}

walk_Stmts (ptr)
STMTS_TYPE  ptr;
{
      if(ptr)
      {
         walk_Stmts(ptr -> Stmts);
         walk_Stmt(ptr -> Stmt);
      }
}

walk_Assignst(ptr)
ASSIGNST_TYPE  ptr;
{
BOOLEAN     onstack;
int   DSloc;

      walk_StoreAccess(Lv, ptr -> StoreAccess, &onstack, &DSloc);
      walk_Expr(ptr -> Expr);
      if(onstack) cg1(SI);
      else  cg2(SD,DSloc);
}

walk_Whilest(ptr)
WHILEST_TYPE  ptr;
{
int   start, cj;

      start = PSused+1;
      walk_Expr(ptr -> Expr);  cg2(JF, 0);  cj = PSused;
      walk_Series(ptr -> Series);  cg2(J, start);
      PS[cj] = PSused+1;
}

walk_Ifst(ptr)
IFST_TYPE  ptr;
{
int        cj, uj;

      walk_Expr(ptr -> Expr);
      cg2(JF,0);  cj = PSused;
      walk_Series(ptr -> Series);
      if(ptr -> Op_Else)
      {
          cg2(J,0);
          uj = PSused;
      }
      PS[cj] = PSused+1;
      walk_Op_Else(ptr -> Op_Else);
      if(ptr -> Op_Else)  PS[uj] = PSused+1;
}

walk_Op_Else(ptr)
OP_ELSE_TYPE  ptr;
{
      if(ptr)  walk_Series(ptr -> Series);
}

walk_Callst(ptr)
CALLST_TYPE   ptr;
```

```
{
int     procSTref;
int     newidr;

        idrchars = malloc(strlen(ptr -> idr)+1);
        strcpy(idrchars,ptr -> idr);
        STlookup(&procSTref, &newidr);
        if(newidr)
              CSerror(ProcNotDec);
        else  if(symtab[procSTref].class != Procname)
              CSerror(NotaProcName);
        cg1(CALL);
        numaps = 0;
        walk_Op_Exprlist(ptr -> Op_Exprlist);
        if(procSTref)
        if(symtab[procSTref].class == Procname)
        if(symtab[procSTref].idrusage.PN.numparam != numaps)
              CSerror(WrongNumOfAPs);
        else
        cg3(JSR, symtab[procSTref].idrusage.PN.entryaddr,
        symtab[procSTref].idrusage.PN.numparam);
}

walk_Op_Exprlist(ptr)
OP_EXPRLIST_TYPE  ptr;
{
        if(ptr)   walk_Exprlist(ptr -> Exprlist);
}

walk_Exprlist(ptr)
EXPRLIST_TYPE  ptr;
{
        walk_Expr(ptr -> Expr);
        numaps++;
        walk_Exprs(ptr -> Exprs);
}

walk_Exprs(ptr)
EXPRS_TYPE  ptr;
{
        if(ptr)
        {
            walk_Exprs(ptr -> Exprs);
            walk_Expr(ptr -> Expr);
            numaps++;
        }
}

walk_Readst(ptr)
READST_TYPE  ptr;
{
        walk_Readinlist(ptr -> Readinlist);
        if(!ptr -> Optionalplus)   cg1(RNL);
}

walk_Writest(ptr)
WRITEST_TYPE  ptr;
{
        walk_Writeoutlist(ptr -> Writeoutlist);
        if(!ptr -> Optionalplus)   cg1(WNL);
}
```

```
walk_Readinlist(ptr)
READINLIST_TYPE   ptr;
{
BOOLEAN     onstack;
int         DSloc;

      if(ptr -> Optionalhash)
       walk_StoreAccess(Rv, ptr -> StoreAccess, &onstack, &DSloc);
      else
       walk_StoreAccess(Lv, ptr -> StoreAccess, &onstack, &DSloc);
      if(ptr -> Optionalhash)  cg1(RTXT);
      else
      {
            if(!onstack) cg2(LC,DSloc);
            cg1(RNUM);
      }
      walk_Storelist(ptr -> Storelist);
}

walk_Storelist(ptr)
STORELIST_TYPE  ptr;
{
BOOLEAN     onstack;
int         DSloc;
   if(ptr)
   {
       walk_Storelist(ptr -> Storelist);
       if(ptr -> Optionalhash)
        walk_StoreAccess(Rv, ptr->StoreAccess, &onstack, &DSloc);
       else
        walk_StoreAccess(Lv, ptr->StoreAccess, &onstack, &DSloc);
       if(ptr -> Optionalhash)  cg1(RTXT);
       else
       {
            if(!onstack) cg2(LC,DSloc);
            cg1(RNUM);
       }
   }
}

walk_Writeoutlist(ptr)
WRITEOUTLIST_TYPE  ptr;
{
      walk_Expr(ptr -> Expr);
      if(ptr -> Optionalhash || textual)   cg1(WTXT);
      else  cg1(WNUM);
      walk_ListExpr(ptr -> ListExpr);
}

walk_ListExpr(ptr)
LISTEXPR_TYPE  ptr;
{
  if(ptr)
  {
      walk_ListExpr(ptr -> ListExpr);
      walk_Expr(ptr -> Expr);
      if(ptr -> Optionalhash || textual)   cg1(WTXT);
      else  cg1(WNUM);
  }
}
```

```
walk_Expr(ptr)
EXPR_TYPE  ptr;
{
      walk_Operand(ptr -> Operand);
      walk_Operands(ptr -> Operands);
}
walk_Operands(ptr)
OPERANDS_TYPE  ptr;
{
int   opcode;
int   i;

  if(ptr)
  {
      walk_Operands(ptr -> Operands);
      for(i=0; i<12; i++)
      if(codes[i].tokennum == ptr  -> opr)
      {
             opcode = codes[i].tmcode;
             break;
      }
      walk_Operand(ptr -> Operand);
      cg1(opcode);
  }
}

walk_Operand(ptr)
OPERAND_TYPE   ptr;
{
BOOLEAN      onstack;
int          STpos;
int                DSloc;

switch(ptr -> type) {
case   1  :  cg2(LC,ptr->RIGHTSIDE.intsym);  break;
case   2  :  textual = TRUE;
             cg2(LC,textaddress(ptr->RIGHTSIDE.text));  break;
case   3  :        cg2(LC,TMtrue);  break;
case   4  :  cg2(LC,TMfalse);  break;
case   5  :  idrchars = malloc(strlen(ptr->RIGHTSIDE.idr)+1);
             strcpy(idrchars,ptr->RIGHTSIDE.idr);
             checkdeclared(&STpos);
             cg2(LC,getRTSL(STpos));  break;
case   6  : walk_StoreAccess(Rv, ptr->RIGHTSIDE.StoreAccess,
                                   &onstack, &DSloc);  break;
case   7  :  walk_Expr(ptr -> RIGHTSIDE.Expr);  break;
default   :  printf("ERROR -- wrong alternative number\n");  break;
}/* end of  switch  */
}

walk_StoreAccess(use,ptr,onstack,DSloc)
STOREACCESS_TYPE   ptr;
BOOLEAN            *onstack;
int                use;
int                *DSloc;
{
int STpos;

      idrchars = malloc(strlen(ptr -> idr)+1);
      strcpy(idrchars,ptr -> idr);
```

```
        checkdeclared(&STpos);
        if(ptr -> Op_Excl)
        {
              CGloadcontents(STpos);
              walk_Op_Excl(ptr -> Op_Excl);
              cg1(ADDop);
              *onstack = TRUE;
              if(use == Rv) cg1(LI);
        }
        else
        {
              if(use == Rv) CGloadcontents(STpos);
              else
              {
                  *DSloc = getRTSL(STpos);
                  *onstack = FALSE;
              }
        }

}

walk_Op_Excl(ptr)
OP_EXCL_TYPE  ptr;
{
        if(ptr)  walk_Operand(ptr -> Operand);
}

/************************** STlookup() *************************/

STlookup(pos, newidr)
int         *pos;
BOOLEAN     *newidr;
{

*newidr = TRUE;   /* initial assumption */
*pos = 1;

        while(*newidr && (*pos <= numidrs))
        if(!strcmp(idrchars,symtab[*pos].idrname))
              *newidr = FALSE;
        else  (*pos)++;
        if(*newidr)
        {
              numidrs++;
              *pos = numidrs;
              symtab[*pos].idrname = malloc(strlen(idrchars)+1);
              strcpy(symtab[*pos].idrname,idrchars);
              symtab[*pos].class = Undefined;
        }
}/* end of STlookup()  */


/************************ checkdeclared() **********************/

checkdeclared(STpos)
int         *STpos;
{
BOOLEAN     newidr;

  STlookup(STpos, &newidr);
  if(newidr)
```

```
  {
      symtab[*STpos].class = Variable;
      DSused++;
      symtab[*STpos].idrusage.var_rtsl = DSused;
  }
}/* end of checkdeclared() */


/********************** checkprocidr() ***********************/

checkprocidr()
{
BOOLEAN     newidr;

  STlookup(&currentprocref, &newidr);
  if(newidr)
  {
      symtab[currentprocref].class = Procname;
      symtab[currentprocref].idrusage.PN.entryaddr = PSused+1;
  }
  else   CSerror(PrNameNotNew);

}/* end of checkprocidr()  */

/************************ checkFPcidr() *********************/

checkFPidr()
{
BOOLEAN     newidr;
int         pos;

  STlookup(&pos, &newidr);
  if(newidr)
  {
      symtab[pos].class = Formalparam;
      symtab[pos].idrusage.FP.procref = currentprocref;
      symtab[pos].idrusage.FP.paramnum = numfps;
  }
  else  CSerror(FPnotNew);

}/* end of checkFPidr()  */

/********************** CGloadcontents() ********************/

CGloadcontents(STref)
int     STref;
{

  switch(symtab[STref].class){
  case  Procname : CSerror(BadProcNameUs);  break;

  case  Variable : cg2(LD,symtab[STref].idrusage.var_rtsl);
                  break;

  case  Formalparam :if(currentprocref ==
                          symtab[STref].idrusage.FP.procref)
                      cg2(LA,symtab[STref].idrusage.FP.paramnum);
                      else  CSerror(BadFPusage);
                      break;
  default      :  printf("It must be wrong case \n");
```

```
  }/* end of switch()  */

}/* end of CGloadcontents()  */

/************************** getRTSL() ***************************/

getRTSL(STref)
int      STref;
{
  if(symtab[STref].class == Variable)
      return(symtab[STref].idrusage.var_rtsl);
  else
  {
      if(symtab[STref].class == Procname)  CSerror(BadProcNameUs);
      else CSerror(BadFPusage);
      return(0);        /* dummy result for legality */
  }/* end else */

}/* end of getRTSL() */

/************************ textaddress() ************************/

textaddress(ptr)
char  *ptr;
{
int   i;
int   addr;
int   tokenlen;
char  *p;

  p = ptr;
  addr = DSused = DSused+1;
  tokenlen = strlen(ptr);
  DS[DSused] = tokenlen;           /* length of text */
  for(i=1; i<=tokenlen; i++)
      DS[DSused+i] = *p++;
  DSused = DSused+tokenlen;
  return(addr);

}/* end of textaddress() */


/******************** cg1(), cg2(), cg3() ********************/

cg1(PSitem)
int   PSitem;
{
      PSused++;  PS[PSused] = PSitem;
}

cg2(opcode, opnd)
int    opcode;
int   opnd;
{

      cg1(opcode);  cg1(opnd);
}

cg3(opcode, opnd1, opnd2)
int   opcode;
int   opnd1;
```

```
int   opnd2;
{

      cg1(opcode); cg1(opnd1); cg1(opnd2);
}

/************************ ListSymTab() ************************/

ListSymTab()
{
int    i;

  printf("\n****  Symbol Table    ****\n\n");
  printf("Entry no.    Identifier  Class    \t\t  Data\n");
  printf("----------   ------------     -----     \t\t  ----\n");
  for(i=1; i<= numidrs; i++)
  {
      printf("   %d\t\t  %s",i,symtab[i].idrname);
      switch(symtab[i].class){
      case Procname     :printf("\t\tProcName          \t%d\t%d\n",
                         symtab[i].idrusage.PN.entryaddr,
                         symtab[i].idrusage.PN.numparam); break;
      caseFormalparam :printf("\t\tFormalParam        \t%d\t%d\n",
                         symtab[i].idrusage.FP.procref,
                         symtab[i].idrusage.FP.paramnum);
                        break;
      case Variable     :printf("\t\tVariable          \t%d\t__\n",
                         symtab[i].idrusage.var_rtsl); break;
      case Undefined    :printf("\t\tUndefined          \t--\t--\n");
                        break;
      }/* end of switch */
  }
  printf("\n");

}/* end of ListSymTab() */

/************************* CSerror() *************************/

CSerror(errtype)
int   errtype;
{

   printf("+++ Context error detected - ");
   switch(errtype){
   case  PrNameNotNew:
   printf("procedure name already used\n"); break;
   case FPnotNew :
   printf("formal parameter identifier already used\n");break;
   case  NotaProcName :
   printf("CALL identifier not declared as a procedure name\n");
   break;
   case  WrongNumOfAPs:
   printf("wrong number of actual parameter\n"); break;
   case  BadProcNameUs     :
   printf("improper use of procedure name\n");  break;
   case  BadFPusage   :
   printf("improper use of formal parameter\n");  break;
   case  ProcNotDec   : printf("procedure not declared\n"); break;
   }/* end of switch */
   numerrs++;
}/* end of CSerror  */
```

```
/************************ ListTMcode() ************************/

ListTMcode()
{
int     PSindex;
int     opcode, j;

  PSindex = 1;  j = 0;
  printf("****  TM code  ****\n");
  while(PSindex <= PSused)
  {
      printf("%d)  ",PSindex);
      opcode = PS[PSindex];
      if(opcode < 1 && opcode > 31)
           printf("**  BAD OPCODE  **");
      else
      switch(opcode){
      case  J          :    printf("J  ");          break;
      case  LD         :    printf("LD  ");         break;
      case  SD         :    printf("SD  ");         break;
      case  LC         :    printf("LC  ");         break;
      case  LA         :    printf("LA  ");         break;
      case  JF         :    printf("JF  ");         break;
      case  JT         :    printf("JT  ");         break;
      case  JSR        :    printf("JSR  ");        break;
      case  LI         :    printf("LI  ");         break;
      case  SI         :    printf("SI  ");         break;
      case  WNUM       :    printf("WNUM  ");       break;
      case  RNUM       :    printf("RNUM  ");       break;
      case  WTXT       :    printf("WTXT  ");       break;
      case  RTXT       :    printf("RTXT  ");       break;
      case  WNL        :    printf("WNL  ");        break;
      case  RNL        :    printf("RNL  ");        break;
      case  CALL       :    printf("CALL  ");       break;
      case  RTN        :    printf("RTN  ");        break;
      case  HALT       :    printf("HALT  ");       break;
      case  ADDop      :    printf("ADD  ");        break;
      case  SUBop      :    printf("SUB  ");        break;
      case  MULTop     :    printf("MULT  ");       break;
      case  DVDop      :    printf("DVD  ");        break;
      case  LTop       :    printf("LT  ");         break;
      case  LEop       :    printf("LE  ");         break;
      case  GTop       :    printf("GT  ");         break;
      case  GEop       :    printf("GE  ");         break;
      case  NEop       :    printf("NE  ");         break;
      case  EQop       :    printf("EQ  ");         break;
      case  LOGORop    :    printf("LOGOR  ");      break;
      case  LOGANDop   :    printf("LOGAND  ");     break;
      default          :    printf("Wrong opcode \n");

      }/* end of switch */
      if(opcode >0 && opcode < 9)
      {
            PSindex++;
            printf("%d  ",PS[PSindex]);
            if(opcode == JSR)
            {
               PSindex++;
               printf("%d  ",PS[PSindex]);
            }
```

```
        }
        printf("\t");
        j++;
        if(j == 4)  /* printf four instructions in each line   */
        {  printf("\n"); j = 0; }
        PSindex++;
    }/* end of while */

}/* end of ListTMcode() */
```

```
/****  Definition of constants  HW.define.h   ****/

      /**  define constants for TM operation codes  **/

#define     LD              1
#define     SD              2
#define     LC              3
#define     LA              4
#define     J               5
#define     JF              6
#define     JT              7
#define     JSR             8
#define     LI              9
#define     SI              10
#define     WNUM            11
#define     RNUM            12
#define     WTXT            13
#define     RTXT            14
#define     WNL             15
#define     RNL             16
#define     CALL            17
#define     RTN             18
#define     HALT            19
#define     ADDop           20
#define     SUBop           21
#define     MULTop          22
#define     DVDop           23
#define     LTop            24
#define     LEop            25
#define     GTop            26
#define     GEop            27
#define     NEop            28
#define     EQop            29
#define     LOGORop         30
#define     LOGANDop        31
#define     TMfalse         0
#define     TMtrue          1
#define     Procname        100
#define     Formalparam     101
#define     Variable        102
#define     Undefined       103

      /** define constants for error messages  **/

#define     PrNameNotNew     104
#define     FPnotNew         105
#define     NotaProcName     106
#define     WrongNumOfAPs    107
#define     BadProcNameUs    108
#define     BadFPusage       109
#define     ProcNotDec       110
#define     Lv               111
#define     Rv               112

#define     BOOLEAN          int
#define     FALSE            0
#define     TRUE             1

#define     Maxidrs         100 /* maximum number of identifie      */
#define     PSsize          400 /* maximum size of the program store */
```

```
#define     DSsize          400 /* maximum size of the data store    */

typedef struct stentry{
      char  *idrname;           /* identifier name        */
      int   class;              /* the identifier class        */
      union {
            int   var_rtsl;     /* Run-time store location*/
            struct  procname{
                  int    entryaddr;  /* entry addresse        */
                  int    numparam;   /* number of parameters */
            }PN;
            struct      formalparam {
                  int   procref;  /* procedure reference      */
                  int   paramnum; /* parameter number (position) */
            }FP;
      }idrusage;  /* end of union   */
}STENTRY;

typedef     struct opcode{
      int   tokennum;
      int   tmcode;
}OPCODE;

STENTRY        symtab[Maxidrs];
int            PS[PSsize];
int            DS[DSsize];

static OPCODE codes[] ={{263,ADDop}, {265,SUBop}, {262,MULTop},
 {267,DVDop},{269,LTop}, {270,LEop}, {273,GTop},{274,GEop},
 {271,NEop},{272,EQop}, {258,LOGORop}, {259,LOGANDop}};
```

Appendix B

This appendix contains a CORGI specification for MSL (`msl.spec`), together with the generated files from CORGI. Thus the set of generated files given in this appendix are: `lex.spec`, `yacc.spec`, `writer.c`, `reader.c`, `walker.c` `syntree.h`, `define.h`. Due to the length of some files we preferred to cut down most of the body of the generated functions in `writer.c`, `reader.c` and `walker.c`. The walker given in this appendix is the generated version, ie. it has not been augmented with code to deal with semantic analysis and code generation yet.

```
                    /****   file = msl.spec   ****/

STARTCOMMENT          "--".
ENDCOMMENT            NEWLINE.
LEXEME                idr int text "TRUE" "FALSE" "#".
KEYWORD_CASE          KEY_NONSIG.
OPERATORS
opr  = \L { "%" "&" "+" "-" "*" "/" "<" "<=" "<>" ">=" ">" "=" }.

%%

Program     = ["RESERVE" int ] {ProcDec} Series ".".

ProcDec = "PROC" idr  ["(" Formalparams ")"] Series "END".

Formalparams = idr {"," idr}.

Series = Stmt {Stmt}.

Stmt = Assignst | Whilest | Ifst | Callst | Readst | Writest.

Assignst = StoreAccess ":=" Expr.

Whilest = "WHILE" Expr "DO" Series "OD".

Ifst = "IF" Expr "THEN" Series ["ELSE" Series] "FI".

Callst = "CALL" idr ["(" Exprlist ")"].

Exprlist = Expr {"," Expr}.

Readst = "READ" Optionaldolar Optionalhash Readinlist.

Writest = "WRITE" Optionaldolar Optionalhash Writeoutlist.

Readinlist = StoreAccess {"," Optionalhash StoreAccess }.

Writeoutlist = Expr {"," Optionalhash Expr}.
```

```
Optionalhash = ["#"]          ==> evaluate_token.

Optionaldolar = ["+"].

Expr = Operand {opr Operand}.

Operand  = int
         | text
         | "TRUE"             ==> evaluate_token
         | "FALSE"            ==> evaluate_token
         | "@" idr
         |StoreAccess | "(" Expr ")".

StoreAccess = idr ["!" Operand].

idr = letter { letter | digit }         ==> evaluate_idr (0).

letter = uplow_case.

int  = digit { digit }                  ==> evaluate_denary.

text = """" {stringpic} """"            ==> evaluate_Cstr.

stringpic = anybut_DQ_NL.
```

/****   <u>file = lex.spec</u>   ****/

```
%{
#include    "syntree.h"
#include    "/users/sirius/pg/nad/project/src/backend/Lex/yymark.c"
#ifdef      DEBUG          /* debugging version - if assert ok */
#include    <assert.h>
main()
{
char     *p;
     assert(sizeof(int) >= sizeof(char *));
     while(p = (char*)yylex())
     {
        if(yyleng <= 1)
           printf("%-10.10s is \"%s\"\n","yytext[0]",yytext);
        else
           printf("%-10.10s is \"%s\"\n",p,yytext);
        }
}
#else        !DEBUG
#include     "y.tab.h"
#endif       DEBUG
#define      ATTRIBUTE             ATTRIBUTE_TYPE
#define      ENDTABLE(v)          (v-1 + sizeof v /sizeof v[0])

#include      <ctype.h>
#include      "lexdefs.h"
#include"/users/sirius/pg/nad/project/src/backend/Lex/comment.c"

#define     KEYWORD_SIG       0
#define     IDENTIF_SIG       0
#define     NESTED            0
int gotlexeme = 0;
```

```
int linenumber = 0;
int yycode = 0;
char ch;
static int count = 0;
%}
digit                    [0-9]
uplow_case               [A-Za-z]
anybut_DQ_NL             [^\"\n]
stringpic                ({anybut_DQ_NL})
letter                   ({uplow_case})
text                     "\""(({stringpic}))*"\""
int                      ({digit})(({digit}))*
idr                      ({letter})(({letter})|({digit}))*
blank                    [ \t]

%%

^"#"{blank}*{digit}+({blank}+.*)?\n          {yymark();}
[ \t]         {/* do nothing */}
[\n]          { linenumber++; if(count != 0)   return(ENDCOMMENT);}

"!"           { return(DEL1); }
"%"           {
                yylval.Vattribute =
                     (ATTRIBUTE*) malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP1;
                yylval.Vattribute->type = 1;  return(OP1);   }
"&"           {
                yylval.Vattribute =
                     (ATTRIBUTE*) malloc(sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP2;
                yylval.Vattribute->type = 1;  return(OP2);   }
"("           { return(DEL2); }
")"           { return(DEL3); }
"*"           {
                yylval.Vattribute =
                     (ATTRIBUTE*) malloc(sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP5;
                yylval.Vattribute->type = 1;  return(OP5);   }
"+"           {
                yylval.Vattribute =
                     (ATTRIBUTE*) malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP3;
                yylval.Vattribute->type = 1;  return(OP3);   }
","           { return(DEL4); }
"-"           {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP4;
                yylval.Vattribute->type = 1;  return(OP4);   }
"."           {   return(DEL5); }
"/"           {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE_));
                yylval.Vattribute->textval.d = OP6;
                yylval.Vattribute->type = 1;  return(OP6);   }
":="          { return(DEL6); }
"<"           {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP7;
```

```
                yylval.Vattribute->type = 1;  return(OP7);  }
"<="          {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP8;
                yylval.Vattribute->type = 1;  return(OP8);  }
"<>"          {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP9;
                yylval.Vattribute->type = 1;  return(OP9);  }
"="           {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc(sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP12;
                yylval.Vattribute->type = 1;  return(OP12);  }
">"           {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP11;
                yylval.Vattribute->type = 1;  return(OP11);  }
">="          {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = OP10;
                yylval.Vattribute->type = 1;  return(OP10);  }
"@"           { return(DEL7); }
"\""          { return(DEL8); }
"#"           {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->textval.d = DEL9;
                yylval.Vattribute->type = 1;  return(DEL9);  }

{idr}               {
#define IDR_TOKEN   IDR
int    Toknum;
                Toknum = screen();
                if(Toknum == IDR_TOKEN)
                {
                  yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                  yylval.Vattribute->textval.s = evaluate_idr();
                  yylval.Vattribute->type = 3;
                }
                else
                {
                   if(gotlexeme)
                   {
                     yylval.Vattribute =
                          (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                     yylval.Vattribute->type = INT_PRINT;
                     yylval.Vattribute->textval.d = Toknum;
                   }
                }
                return(Toknum);  }

{int}         {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->type = INT_PRINT;
```

```
                yylval.Vattribute->textval.d = evaluate_decimal();
                return(token(INT));}
{text}        {
                yylval.Vattribute =
                     (ATTRIBUTE*)malloc (sizeof (ATTRIBUTE));
                yylval.Vattribute->type = TEXT_PRINT;
                yylval.Vattribute->textval.s = evaluate_Cstr();
                return(token(TEXT));}
"--"          {
                if(NESTED || count == 0)
                {
                   count++;
                   if(comment())
                   { printf("Premature EOF\n");  exit(-1); }
                   count--;
                }
              }
.             { if(count == 0)
                printf("\n!!! char <%c> is illegal here\n",
                      yytext[0]);}

%%
              /*  reserved word table  */

static  struct rwtable{  /* reserved word table */
      char   *rw_name;   /* representation  */
      int    rw_yylex;   /* yylex() value  */
      int    flag;  /* indicates whether it is a lexeme or not */
             }rwtable[] = {
             "CALL",          CALLSYM,       0,
             "DO",            DOSYM,         0,
             "ELSE",          ELSESYM,       0,
             "END",           ENDSYM,        0,
             "FALSE",         FALSESYM,      1,
             "FI",            FISYM,         0,
             "IF",            IFSYM,         0,
             "OD",            ODSYM,         0,
             "PROC",          PROCSYM,       0,
             "READ",          READSYM,       0,
             "RESERVE",       RESERVESYM,    0,
             "THEN",          THENSYM,       0,
             "TRUE",          TRUESYM,       1,
             "WHILE",         WHILESYM,      0,
             "WRITE",         WRITESYM,      0,
             };

static  int  screen()
{
struct  rwtable   *low = rwtable,
                  *high = ENDTABLE(rwtable),
                  *place;
int     cond;
char    *pname;

   while(low <= high)
   {
      place = low + (high - low) / 2;
      pname = place -> rw_name;
      if((cond = streqv(pname)) < 0)
           low = place +1;
      else
```

```
        if(cond > 0)
            high = place -1;
        else
        {
           if(place -> flag )
             gotlexeme = 1;
           return place -> rw_yylex;
        }
    }
    return(IDR_TOKEN);
}

streqv(pname)
char  *pname;
{
extern  char  toupper_c();
char    *s, *t;

    s = pname;  t = yytext;
    if(KEYWORD_SIG)
         return(strcmp(pname,yytext));
    for(;toupper_c(*s) == toupper_c(*t); s++,t++)
     if(*s == '\0')  return(0);
    return(toupper_c(*s) - toupper_c(*t));
}

char  toupper_c(c)
char  c;
{
    if(islower(c))    return(toupper(c));
    return(c);
}
yywrap() {    return;    }
```

/**** file = yacc.spec ****/

```
%{
#include    "syntree.h"
#include    "writer.c"
#include    "mknodes.h"
%}

/*  Types associated with grammar symbols  */

%union{
ATTRIBUTE_TYPE                *Vattribute;
PROGRAM_TYPE                  VProgram;
NEWRULE2_TYPE                 Vnewrule2;
NEWRULE1_TYPE                 Vnewrule1;
PROCDEC_TYPE                  VProcDec;
NEWRULE3_TYPE                 Vnewrule3;
FORMALPARAMS_TYPE             VFormalparams;
NEWRULE4_TYPE                 Vnewrule4;
SERIES_TYPE                   VSeries;
NEWRULE5_TYPE                 Vnewrule5;
STMT_TYPE                     VStmt;
ASSIGNST_TYPE                 VAssignst;
WHILEST_TYPE                  VWhilest;
IFST_TYPE                     VIfst;
NEWRULE6_TYPE                 Vnewrule6;
CALLST_TYPE                   VCallst;
NEWRULE7_TYPE                 Vnewrule7;
EXPRLIST_TYPE                 VExprlist;
NEWRULE8_TYPE                 Vnewrule8;
READST_TYPE                   VReadst;
WRITEST_TYPE                  VWritest;
READINLIST_TYPE               VReadinlist;
NEWRULE9_TYPE                 Vnewrule9;
WRITEOUTLIST_TYPE             VWriteoutlist;
NEWRULE10_TYPE                Vnewrule10;
OPTIONALHASH_TYPE             VOptionalhash;
OPTIONALPLUS_TYPE             VOptionalplus;
EXPR_TYPE                     VExpr;
NEWRULE11_TYPE                Vnewrule11;
OPERAND_TYPE                  VOperand;
STOREACCESS_TYPE              VStoreAccess;
NEWRULE12_TYPE                Vnewrule12;
}

/*  operator tokens  */

%token     OP1              /*     OP1 = "%"       */
%token     OP2              /*     OP2 = "&"       */
%token     OP5              /*     OP5 = "*"       */
%token     OP3              /*     OP3 = "+"       */
%token     OP4              /*     OP4 = "-"       */
%token     OP6              /*     OP6 = "/"       */
%token     OP7              /*     OP7 = "<"       */
%token     OP8              /*     OP8 = "<="      */
%token     OP9              /*     OP9 = "<>"      */
%token     OP12             /*     OP12 = "="      */
%token     OP11             /*     OP11 = ">"      */
%token     OP10             /*     OP10 = ">="     */
```

```
/*  delimiter tokens  */

%token    DEL1            /*     DEL1 = "!"     */
%token    DEL2            /*     DEL2 = "("     */
%token    DEL3            /*     DEL3 = ")"     */
%token    DEL4            /*     DEL4 = ","     */
%token    DEL5            /*     DEL5 = "."     */
%token    DEL6            /*     DEL6 = ":="    */
%token    DEL7            /*     DEL7 = "@"     */
%token    DEL8            /*     DEL8 = "\""    */
%token    DEL9            /*     DEL9 = "#"     */

/*  lexeme tokens  */

%token    IDR
%token    INT
%token    TEXT

/*  The keyword tokens  */

%token    CALLSYM
%token    DOSYM
%token    ELSESYM
%token    ENDSYM
%token    FISYM
%token    IFSYM
%token    ODSYM
%token    PROCSYM
%token    READSYM
%token    RESERVESYM
%token    THENSYM
%token    WHILESYM
%token    WRITESYM
%token    TRUESYM
%token    FALSESYM

/* Operator precedence and associativity  */

%left   OP1 OP2 OP5 OP3 OP4 OP6 OP7 OP8 OP9 OP12 OP11 OP10

/*  Type declaration. */

%type  <Vattribute>      OP1
%type  <Vattribute>      OP2
%type  <Vattribute>      OP3
%type  <Vattribute>      OP4
%type  <Vattribute>      OP5
%type  <Vattribute>      OP6
%type  <Vattribute>      OP7
%type  <Vattribute>      OP8
%type  <Vattribute>      OP9
%type  <Vattribute>      OP10
%type  <Vattribute>      OP11
%type  <Vattribute>      OP12

%type  <Vattribute>      DEL9

%type  <Vattribute>      IDR
%type  <Vattribute>      INT
%type  <Vattribute>      TEXT
%type  <Vattribute>      TRUESYM
```

```
%type  <Vattribute>       FALSESYM

%type  <VProgram>         Program
%type  <Vnewrule2>        newrule2
%type  <Vnewrule1>        newrule1
%type  <VProcDec>         ProcDec
%type  <Vnewrule3>        newrule3
%type  <VFormalparams>    Formalparams
%type  <Vnewrule4>        newrule4
%type  <VSeries>          Series
%type  <Vnewrule5>        newrule5
%type  <VStmt>            Stmt
%type  <VAssignst>        Assignst
%type  <VWhilest>         Whilest
%type  <VIfst>            Ifst
%type  <Vnewrule6>        newrule6
%type  <VCallst>          Callst
%type  <Vnewrule7>        newrule7
%type  <VExprlist>        Exprlist
%type  <Vnewrule8>        newrule8
%type  <VReadst>          Readst
%type  <VWritest>         Writest
%type  <VReadinlist>      Readinlist
%type  <Vnewrule9>        newrule9
%type  <VWriteoutlist>    Writeoutlist
%type  <Vnewrule10>       newrule10
%type  <VOptionalhash>    Optionalhash
%type  <VOptionalplus>    Optionalplus
%type  <VExpr>            Expr
%type  <Vnewrule11>       newrule11
%type  <VOperand>         Operand
%type  <VStoreAccess>     StoreAccess
%type  <Vnewrule12>       newrule12

%%

Start : Program         {  write_Program($1); };

Program : newrule1 newrule2 Series DEL5
          { $$ = (PROGRAM_TYPE)mknode(4,1,$1,$2,$3); } ;

newrule2 : /* empty */  { $$ = (NEWRULE2_TYPE)NULL; }
         |  newrule2 ProcDec
         { $$ = (NEWRULE2_TYPE)mknode(3,1,$1,$2);};

newrule1 : /* empty */  { $$ = (NEWRULE1_TYPE)NULL; }
         |  RESERVESYM INT
         { $$ = (NEWRULE1_TYPE)mknode(2,1,$2);};

ProcDec : PROCSYM IDR newrule3 Series ENDSYM
          { $$ = (PROCDEC_TYPE)mknode(4,1,$2,$3,$4);};

newrule3 : /* empty */        { $$ = (NEWRULE3_TYPE)NULL; }
         | DEL2 Formalparams DEL3
         { $$ = (NEWRULE3_TYPE)mknode(2,1,$2);};

Formalparams : IDR newrule4
               {$$= (FORMALPARAMS_TYPE)mknode(3,1,$1,$2);};

newrule4 : /* empty */        { $$ = (NEWRULE4_TYPE)NULL; }
         |  newrule4 DEL4 IDR
```

```
        { $$ = (NEWRULE4_TYPE)mknode(3,1,$1,$3);};

Series : Stmt newrule5
        { $$ = (SERIES_TYPE)mknode(3,1,$1,$2);};

newrule5 : /* empty */        { $$ = (NEWRULE5_TYPE)NULL; }
        | newrule5 Stmt
        { $$ = (NEWRULE5_TYPE)mknode(3,1,$1,$2);};

Stmt :  Assignst         {$$ = (STMT_TYPE)mknode(2,1,$1);}
     |  Whilest          { $$ = (STMT_TYPE)mknode(2,2,$1);}
     |  Ifst             { $$ = (STMT_TYPE)mknode(2,3,$1);}
     |  Callst           { $$ = (STMT_TYPE)mknode(2,4,$1);}
     |  Readst           { $$ = (STMT_TYPE)mknode(2,5,$1);}
     |  Writest          { $$ = (STMT_TYPE)mknode(2,6,$1);};

Assignst : StoreAccess DEL6 Expr
          { $$ = (ASSIGNST_TYPE)mknode(3,1,$1,$3);};

Whilest : WHILESYM Expr DOSYM Series ODSYM
         { $$ = (WHILEST_TYPE)mknode(3,1,$2,$4);};

Ifst : IFSYM Expr THENSYM Series newrule6 FISYM
      { $$ = (IFST_TYPE)mknode(4,1,$2,$4,$5);};

newrule6 : /* empty */ { $$ = (NEWRULE6_TYPE)NULL; }
        | ELSESYM Series
        { $$ = (NEWRULE6_TYPE)mknode(2,1,$2);};

Callst : CALLSYM IDR newrule7
        { $$ = (CALLST_TYPE)mknode(3,1,$2,$3);};

newrule7 : /* empty */ { $$ = (NEWRULE7_TYPE)NULL; }
        | DEL2 Exprlist DEL3
        { $$ = (NEWRULE7_TYPE)mknode(2,1,$2);};

Exprlist : Expr newrule8
          { $$ = (EXPRLIST_TYPE)mknode(3,1,$1,$2);};

newrule8 : /* empty */ { $$ = (NEWRULE8_TYPE)NULL; }
        | newrule8 DEL4 Expr
        { $$ = (NEWRULE8_TYPE)mknode(3,1,$1,$3);};

Readst : READSYM Optionalplus  Optionalhash Readinlist
        { $$ = (READST_TYPE)mknode(4,1,$2,$3,$4);};

Writest : WRITESYM Optionalplus Optionalhash Writeoutlist
         { $$ = (WRITEST_TYPE)mknode(4,1,$2,$3,$4);};

Readinlist : StoreAccess newrule9
            { $$ = (READINLIST_TYPE)mknode(3,1,$1,$2);};

newrule9 : /* empty */ { $$ = (NEWRULE9_TYPE)NULL; }
        | newrule9 DEL4 Optionalhash StoreAccess
        { $$ = (NEWRULE9_TYPE)mknode(4,1,$1,$3,$4);};

Writeoutlist : Expr newrule10
              { $$ = (WRITEOUTLIST_TYPE)mknode(3,1,$1,$2);};

newrule10 : /* empty */       { $$ = (NEWRULE10_TYPE)NULL; }
         | newrule10 DEL4 Optionalhash Expr
```

```
          { $$ = (NEWRULE10_TYPE)mknode(4,1,$1,$3,$4);};

Optionalplus : /* empty */    { $$ = (OPTIONALPLUS_TYPE)NULL; }
             | OP3
             { $$ = (OPTIONALPLUS_TYPE)mknode(2,1,$1);};

Optionalhash : /* empty */    { $$ = (OPTIONALHASH_TYPE)NULL; }
             | DEL9 { $$ = (OPTIONALHASH_TYPE)mknode(2,1,$1);};

Expr : Operand newrule11
       { $$ = (EXPR_TYPE)mknode(3,1,$1,$2);};

newrule11 : /* empty */        { $$ = (NEWRULE11_TYPE)NULL; }
          | newrule11 OP1 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,1,$1,$2,$3);}
          |  newrule11 OP2 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,2,$1,$2,$3);}
          | newrule11 OP3 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,3,$1,$2,$3);}
          | newrule11 OP4 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,4,$1,$2,$3);}
          | newrule11 OP5 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,5,$1,$2,$3);}
          | newrule11 OP6 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,6,$1,$2,$3);}
          | newrule11 OP7 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,7,$1,$2,$3);}
          | newrule11 OP8 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,8,$1,$2,$3);}
          | newrule11 OP9 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,9,$1,$2,$3);}
          | newrule11 OP10 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,10,$1,$2,$3);}
          | newrule11 OP11 Operand
                { $$ = (NEWRULE11_TYPE)mknode(4,11,$1,$2,$3);}
          | newrule11 OP12 Operand
                {$$ = (NEWRULE11_TYPE)mknode(4,12,$1,$2,$3);};

Operand   :  INT              { $$ = (OPERAND_TYPE)mknode(2,1,$1);}
          |  TEXT             { $$ = (OPERAND_TYPE)mknode(2,2,$1);}
          |  TRUESYM          { $$ = (OPERAND_TYPE)mknode(2,3,$1);}
          |  FALSESYM         { $$ = (OPERAND_TYPE)mknode(2,3,$1);}
          |  DEL7 IDR         { $$ = (OPERAND_TYPE)mknode(2,5,$2);}
          |  StoreAccess      { $$ = (OPERAND_TYPE)mknode(2,6,$1);}
          |  DEL2 Expr DEL3   { $$ = (OPERAND_TYPE)mknode(2,7,$2);};

StoreAccess : IDR newrule12
              { $$ = (STOREACCESS_TYPE)mknode(3,1,$1,$2);};

newrule12 : /* empty */    { $$ = (NEWRULE12_TYPE)NULL; }
          | DEL1 Operand { $$ = (NEWRULE12_TYPE)mknode(2,1,$2);};

%%
main()
{
#ifdef LEXDEBUG
  debug = 1;          /* debug = 1 for debugging and 0 otherwise */
#endif
extern  int  yynerrs;
int flag = 0;
```

```
     flag = yyparse();
     printf("Compilation error(s): %d\n", yynerrs);
     if(flag)
        printf("Compilation aborted\n");
     else
        printf("Compilation terminated\n");
}


                 /****    file = syntree.h    ****/

      /*  definition of constants   */

#define   NIL            0
#define   STRUCT         struct
#define   UNION          union
#define   INT_PRINT      1
#define   TEXT_PRINT     3
#define   OPR_PRINT      1

      /*  typedef node types */

typedef  struct  Program_type           *  PROGRAM_TYPE;
typedef  struct  newrule2_type          *  NEWRULE2_TYPE;
typedef  struct  newrule1_type          *  NEWRULE1_TYPE;
typedef  struct  ProcDec_type           *  PROCDEC_TYPE;
typedef  struct  newrule3_type          *  NEWRULE3_TYPE;
typedef  struct  Formalparams_type      *  FORMALPARAMS_TYPE;
typedef  struct  newrule4_type          *  NEWRULE4_TYPE;
typedef  struct  Series_type            *  SERIES_TYPE;
typedef  struct  newrule5_type          *  NEWRULE5_TYPE;
typedef  struct  Stmt_type              *  STMT_TYPE;
typedef  struct  Assignst_type          *  ASSIGNST_TYPE;
typedef  struct  Whilest_type           *  WHILEST_TYPE;
typedef  struct  Ifst_type              *  IFST_TYPE;
typedef  struct  newrule6_type          *  NEWRULE6_TYPE;
typedef  struct  Callst_type            *  CALLST_TYPE;
typedef  struct  newrule7_type          *  NEWRULE7_TYPE;
typedef  struct  Exprlist_type          *  EXPRLIST_TYPE;
typedef  struct  newrule8_type          *  NEWRULE8_TYPE;
typedef  struct  Readst_type            *  READST_TYPE;
typedef  struct  Writest_type           *  WRITEST_TYPE;
typedef  struct  Readinlist_type        *  READINLIST_TYPE;
typedef  struct  newrule9_type          *  NEWRULE9_TYPE;
typedef  struct  Writeoutlist_type      *  WRITEOUTLIST_TYPE;
typedef  struct  newrule10_type         *  NEWRULE10_TYPE;
typedef  struct  Optionalhash_type      *  OPTIONALHASH_TYPE;
typedef  struct  Optionalplus_type      *  OPTIONALPLUS_TYPE;
typedef  struct  Expr_type              *  EXPR_TYPE;
typedef  struct  newrule11_type         *  NEWRULE11_TYPE;
typedef  struct  Operand_type           *  OPERAND_TYPE;
typedef  struct  StoreAccess_type       *  STOREACCESS_TYPE;
typedef  struct  newrule12_type         *  NEWRULE12_TYPE;

      /*  declaration of data structure types  */

typedef  union{
int   d;
char  c;
char  *s;
}UVAL;
```

```
typedef struct  attribute_type{
UVAL      textval;
int       type
}ATTRIBUTE_TYPE;

STRUCT  Program_type{
int   type;
UNION   {
      STRUCT   alter_Program_1{
      NEWRULE1_TYPE        newrule1_1;
      NEWRULE2_TYPE        newrule2_2;
      SERIES_TYPE          Series_3;
      }ALTER_Program_1;
              }RIGHTSIDE;
};

STRUCT  newrule2_type{
int   type;
UNION   {
      STRUCT   alter_newrule2_1{
      STRUCT    newrule2_type  *newrule2_1;
      PROCDEC_TYPE              ProcDec_2;
      }ALTER_newrule2_1;
              }RIGHTSIDE;
};

STRUCT  newrule1_type{
int     type;
UNION   {
      STRUCT   alter_newrule1_1{
      ATTRIBUTE_TYPE      * INT_1;
      }ALTER_newrule1_1;
              }RIGHTSIDE;
};

STRUCT  ProcDec_type{
int     type;
UNION   {
      STRUCT   alter_ProcDec_1{
      ATTRIBUTE_TYPE        * IDR_1;
      NEWRULE3_TYPE         newrule3_2;
      SERIES_TYPE           Series_3;
      }ALTER_ProcDec_1;
              }RIGHTSIDE;
};

STRUCT  newrule3_type{
int     type;
UNION   {
      STRUCT   alter_newrule3_1{
      FORMALPARAMS_TYPE     Formalparams_1;
      }ALTER_newrule3_1;
              }RIGHTSIDE;
};

STRUCT  Formalparams_type{
int     type;
UNION   {
      STRUCT   alter_Formalparams_1{
      ATTRIBUTE_TYPE        * IDR_1;
```

```
      NEWRULE4_TYPE        newrule4_2;
      }ALTER_Formalparams_1;
              }RIGHTSIDE;
};

STRUCT  newrule4_type{
int     type;
UNION    {
      STRUCT   alter_newrule4_1{
      STRUCT    newrule4_type  *newrule4_1;
      ATTRIBUTE_TYPE           * IDR_2;
      }ALTER_newrule4_1;
              }RIGHTSIDE;
};

STRUCT  Series_type{
int     type;
UNION    {
      STRUCT   alter_Series_1{
      STMT_TYPE            Stmt_1;
      NEWRULE5_TYPE        newrule5_2;
      }ALTER_Series_1;
              }RIGHTSIDE;
};

STRUCT  newrule5_type{
int     type;
UNION    {
      STRUCT   alter_newrule5_1{
      STRUCT    newrule5_type  *newrule5_1;
      STMT_TYPE                Stmt_2;
      }ALTER_newrule5_1;
              }RIGHTSIDE;
};

STRUCT  Stmt_type{
int     type;
UNION    {
      STRUCT   alter_Stmt_1{
      ASSIGNST_TYPE          Assignst_1;
      }ALTER_Stmt_1;
      STRUCT   alter_Stmt_2{
      WHILEST_TYPE           Whilest_1;
      }ALTER_Stmt_2;
      STRUCT   alter_Stmt_3{
      IFST_TYPE              Ifst_1;
      }ALTER_Stmt_3;
      STRUCT   alter_Stmt_4{
      CALLST_TYPE            Callst_1;
      }ALTER_Stmt_4;
      STRUCT   alter_Stmt_5{
      READST_TYPE            Readst_1;
      }ALTER_Stmt_5;
      STRUCT   alter_Stmt_6{
      WRITEST_TYPE           Writest_1;
      }ALTER_Stmt_6;
              }RIGHTSIDE;
};

STRUCT  Assignst_type{
int     type;
```

```
UNION    {
      STRUCT   alter_Assignst_1{
      STOREACCESS_TYPE        StoreAccess_1;
      EXPR_TYPE               Expr_2;
      }ALTER_Assignst_1;
               }RIGHTSIDE;
};

STRUCT  Whilest_type{
int     type;
UNION    {
      STRUCT   alter_Whilest_1{
      EXPR_TYPE            Expr_1;
      SERIES_TYPE          Series_2;
      }ALTER_Whilest_1;
               }RIGHTSIDE;
};

STRUCT  Ifst_type{
int     type;
UNION    {
      STRUCT   alter_Ifst_1{
      EXPR_TYPE             Expr_1;
      SERIES_TYPE           Series_2;
      NEWRULE6_TYPE         newrule6_3;
      }ALTER_Ifst_1;
               }RIGHTSIDE;
};

STRUCT  newrule6_type{
int     type;
UNION    {
      STRUCT   alter_newrule6_1{
      SERIES_TYPE       Series_1;
      }ALTER_newrule6_1;
               }RIGHTSIDE;
};

STRUCT  Callst_type{
int     type;
UNION    {
      STRUCT   alter_Callst_1{
      ATTRIBUTE_TYPE       * IDR_1;
      NEWRULE7_TYPE        newrule7_2;
      }ALTER_Callst_1;
               }RIGHTSIDE;
};

STRUCT  newrule7_type{
int     type;
UNION    {
      STRUCT   alter_newrule7_1{
      EXPRLIST_TYPE           Exprlist_1;
      }ALTER_newrule7_1;
               }RIGHTSIDE;
};

STRUCT  Exprlist_type{
int     type;
UNION    {
      STRUCT   alter_Exprlist_1{
```

```
        EXPR_TYPE               Expr_1;
        NEWRULE8_TYPE           newrule8_2;
        }ALTER_Exprlist_1;
                }RIGHTSIDE;
};

STRUCT  newrule8_type{
int     type;
UNION   {
        STRUCT   alter_newrule8_1{
        STRUCT    newrule8_type *newrule8_1;
        EXPR_TYPE                Expr_2;
        }ALTER_newrule8_1;
                }RIGHTSIDE;
};

STRUCT  Readst_type{
int     type;
UNION   {
        STRUCT   alter_Readst_1{
        OPTIONALPLUS_TYPE        Optionalplus_1;
        OPTIONALHASH_TYPE        Optionalhash_2;
        READINLIST_TYPE          Readinlist_3;
        }ALTER_Readst_1;
                }RIGHTSIDE;
};

STRUCT  Writest_type{
int     type;
UNION   {
        STRUCT   alter_Writest_1{
        OPTIONALPLUS_TYPE        Optionalplus_1;
        OPTIONALHASH_TYPE        Optionalhash_2;
        WRITEOUTLIST_TYPE        Writeoutlist_3;
        }ALTER_Writest_1;
                }RIGHTSIDE;
};

STRUCT  Readinlist_type{
int     type;
UNION   {
        STRUCT   alter_Readinlist_1{
        STOREACCESS_TYPE         StoreAccess_1;
        NEWRULE9_TYPE            newrule9_2;
        }ALTER_Readinlist_1;
                }RIGHTSIDE;
};

STRUCT  newrule9_type{
int     type;
UNION   {
        STRUCT   alter_newrule9_1{
        STRUCT    newrule9_type   *newrule9_1;
        OPTIONALHASH_TYPE         Optionalhash_2;
        STOREACCESS_TYPE          StoreAccess_3;
        }ALTER_newrule9_1;
                }RIGHTSIDE;
};

STRUCT  Writeoutlist_type{
int     type;
```

```
UNION    {
      STRUCT   alter_Writeoutlist_1{
      EXPR_TYPE                  Expr_1;
      NEWRULE10_TYPE             newrule10_2;
      }ALTER_Writeoutlist_1;
              }RIGHTSIDE;
};

STRUCT  newrule10_type{
int     type;
UNION   {
      STRUCT   alter_newrule10_1{
      STRUCT    newrule10_type  *newrule10_1;
      OPTIONALHASH_TYPE          Optionalhash_2;
      EXPR_TYPE       Expr_3;
      }ALTER_newrule10_1;
              }RIGHTSIDE;
};

STRUCT  Optionalhash_type{
int     type;
UNION   {
      STRUCT   alter_Optionalhash_1{
      ATTRIBUTE_TYPE            * DEL9_1;
      }ALTER_Optionalhash_1;
              }RIGHTSIDE;
};

STRUCT  Optionalplus_type{
int     type;
UNION   {
      STRUCT   alter_Optionalplus_1{
      ATTRIBUTE_TYPE        * OP3_1;
      }ALTER_Optionalplus_1;
              }RIGHTSIDE;
};

STRUCT  Expr_type{
int     type;
UNION   {
      STRUCT   alter_Expr_1{
      OPERAND_TYPE               Operand_1;
      NEWRULE11_TYPE             newrule11_2;
      }ALTER_Expr_1;
              }RIGHTSIDE;
};

STRUCT  newrule11_type{
int     type;
UNION   {
      STRUCT   alter_newrule11_1{
      STRUCT    newrule11_type  *newrule11_1;
      ATTRIBUTE_TYPE            * opr_2;
      OPERAND_TYPE               Operand_3;
      }ALTER_newrule11_1;
              }RIGHTSIDE;
};

STRUCT  Operand_type{
int     type;
UNION   {
```

```
      STRUCT   alter_Operand_1{
      ATTRIBUTE_TYPE           * INT_1;
      }ALTER_Operand_1;
      STRUCT   alter_Operand_2{
      ATTRIBUTE_TYPE           * TEXT_1;
      }ALTER_Operand_2;
      STRUCT   alter_Operand_3{
      ATTRIBUTE_TYPE           * TRUESYM_1;
      }ALTER_Operand_3;
      STRUCT   alter_Operand_4{
      ATTRIBUTE_TYPE           * FALSESYM_1;
      }ALTER_Operand_4;
      STRUCT   alter_Operand_5{
      ATTRIBUTE_TYPE           * IDR_1;
      }ALTER_Operand_5;
      STRUCT   alter_Operand_6{
      STOREACCESS_TYPE         StoreAccess_1;
      }ALTER_Operand_6;
      STRUCT   alter_Operand_7{
      EXPR_TYPE                Expr_1;
      }ALTER_Operand_7;
            }RIGHTSIDE;
};

STRUCT  StoreAccess_type{
int     type;
UNION    {
      STRUCT   alter_StoreAccess_1{
      ATTRIBUTE_TYPE         * IDR_1;
      NEWRULE12_TYPE         newrule12_2;
      }ALTER_StoreAccess_1;
              }RIGHTSIDE;
};

STRUCT  newrule12_type{
int     type;
UNION    {
      STRUCT   alter_newrule12_1{
      OPERAND_TYPE       Operand_1;
      }ALTER_newrule12_1;
              }RIGHTSIDE;
};
```

```
/****   file = writer.c   ****/


#include  <stdio.h>
FILE   *fopen(), *fp_data;

write_Program(ptr)
PROGRAM_TYPE   ptr;
{
fp_data = fopen("flat_tree","w");
if(ptr == NULL)
  fprintf(fp_data,"0\t<Program>\n");
else
{
  fprintf(fp_data,"1\t<Program>\n");
  fprintf(fp_data,"%d\t\t<type_of_Program>\n",ptr->type);
  switch(ptr->type){
  case 1:/* it is alternative no 1 */
      write_newrule1(ptr->RIGHTSIDE.ALTER_Program_1.newrule1_1);
      write_newrule2(ptr->RIGHTSIDE.ALTER_Program_1.newrule2_2);
      write_Series(ptr->RIGHTSIDE.ALTER_Program_1.Series_3);
      break;
  default:printf("ERROR : Wrong alternative number generated\n");
  }/* end switch  */
}/* end else */
}/* end write_Program */

write_newrule2(ptr)
NEWRULE2_TYPE   ptr;
{
if(ptr == NULL)
  fprintf(fp_data,"0\t<newrule2>\n");
else
{
  fprintf(fp_data,"1\t<newrule2>\n");
  fprintf(fp_data,"%d\t\t<type_of_newrule2>\n",ptr->type);
  switch(ptr->type){
  case 1:/* it is alternative no 1 */
      write_newrule2(ptr->RIGHTSIDE.ALTER_newrule2_1.newrule2_1);
      write_ProcDec(ptr->RIGHTSIDE.ALTER_newrule2_1.ProcDec_2);
      break;
  default:printf("ERROR : Wrong alternative number generated\n");
  }/* end switch  */
}/* end else */
}/* end write_newrule2 */

write_newrule1(ptr)
NEWRULE1_TYPE   ptr;
{
if(ptr == NULL)
  fprintf(fp_data,"0\t<newrule1>\n");
else
{
  fprintf(fp_data,"1\t<newrule1>\n");
  fprintf(fp_data,"%d\t\t<type_of_newrule1>\n",ptr->type);
  switch(ptr->type){
  case 1:/* it is alternative no 1 */
      fprintf(fp_data,"%d\n",
      ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->type);
      switch(ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->type)  {
```

```
        case 1:/* it is a number */
              fprintf(fp_data,"%d\n",
              ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->textval.d);
              break;
        case 2:/* it is a character */
              fprintf(fp_data,"%c\n",
              ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->textval.c);
              break;
        case 3:/* it is a string */
              fprintf(fp_data,"%d\n",strlen(
              ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->textval.s));
              if(strlen(
              ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->textval.s))
              fprintf(fp_data,"%s\n",
              ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->textval.s);
              break;
        default:printf(" ERROR : Wrong lexeme type generated\n");
        }/* end switch */
        break;

    default:printf("ERROR : Wrong alternative number generated\n");
    }/* end switch  */
  }/* end else */
  }/* end write_newrule1 */

  write_ProcDec(ptr)
  PROCDEC_TYPE   ptr;
  {
  if(ptr == NULL)
    fprintf(fp_data,"0\t<ProcDec>\n");
  else
  {
    fprintf(fp_data,"1\t<ProcDec>\n");
    fprintf(fp_data,"%d\t\t<type_of_ProcDec>\n",ptr->type);
    switch(ptr->type)   {
    case 1:/* it is alternative no 1 */
        fprintf(fp_data,"%d\n",
        ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->type);
        switch(ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->type)   {
           case 1:/* it is a number */
                 fprintf(fp_data,"%d\n",
                 ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->textval.d);
                 break;
            case 2:/* it is a character */
                 fprintf(fp_data,"%c\n",
                 ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->textval.c);
                 break;
            case 3:/* it is a string */
                 fprintf(fp_data,"%d\n",strlen(
                 ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1-> textval.s));
                 if(strlen(
                 ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1-> textval.s))
                 fprintf(fp_data,"%s\n",
                 ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->textval.s);
                 break;
            default:printf("ERROR : Wrong lexeme type generated\n");
            }/* end switch */
          write_newrule3(ptr->RIGHTSIDE.ALTER_ProcDec_1.newrule3_2);
          write_Series(ptr->RIGHTSIDE.ALTER_ProcDec_1.Series_3);
          break;
```

```
  default:printf("ERROR : Wrong alternative number generated\n");
  }/* end switch  */
}/* end else */
}/* end write_ProcDec */

/***  the rest of the writer routines are cut    ***/
/***  douwn due to the length of these routines  ***/
```

/**** **file = reader.c** ****/

```
#include  <stdio.h>
#include  "extern.h"
#define  realloc_char(s,pt) {\
            s = (char*)realloc(s,strlen(s)+2);\
            if(s==NULL)\
                  printf("cannot realloc char\n");\
            *(s+strlen(s)+1)='\0';\
            *(s+strlen(s)) = (char)pt;\
            }
FILE   *fopen(), *fp_data;
char    *read_Program()
{
PROGRAM_TYPE    ptr;
char   ch;
int    numchars, i, exists;

#ifdef  READDEBUG
printf("In read_Program()\n");
#endif

fp_data = fopen("flat_tree","r");
fscanf(fp_data,"%d\t%*s\n",&exists);
if(!exists)
  return(NULL);
else
{
  ptr = (PROGRAM_TYPE)malloc(sizeof(STRUCT Program_type));
  fscanf(fp_data,"%d\t\t%*s\n",&(ptr->type));
  switch(ptr->type)
  {
  case 1:/* it is alternative no 1 */
      ptr->RIGHTSIDE.ALTER_Program_1.newrule1_1 =
            (NEWRULE1_TYPE)read_newrule1();
      ptr->RIGHTSIDE.ALTER_Program_1.newrule2_2 =
            (NEWRULE2_TYPE)read_newrule2();
      ptr->RIGHTSIDE.ALTER_Program_1.Series_3 =
                  (SERIES_TYPE)read_Series();
      break;
  default:printf("ERROR : wrong alternative number read\n");
  }/* end switch  */
}/* end else  */
#ifdef  READDEBUG
printf("Out read_Program()\n");
#endif

return((char*)ptr);

}/* end read_Program  */
```

```
char     *read_newrule2()
{
NEWRULE2_TYPE    ptr;
char     ch;
int      numchars, i, exists;

#ifdef  READDEBUG
printf("In read_newrule2()\n");
#endif

fscanf(fp_data,"%d\t%*s\n",&exists);
if(!exists)
  return(NULL);
else
{
  ptr = (NEWRULE2_TYPE)malloc(sizeof(STRUCT newrule2_type));
  fscanf(fp_data,"%d\t\t%*s\n",&(ptr->type));
  switch(ptr->type)
  {
  case 1:/* it is alternative no 1 */
           ptr->RIGHTSIDE.ALTER_newrule2_1.newrule2_1 =
                 (NEWRULE2_TYPE)read_newrule2();
           ptr->RIGHTSIDE.ALTER_newrule2_1.ProcDec_2 =
                 (PROCDEC_TYPE)read_ProcDec();
           break;
  default:printf("ERROR : wrong alternative number read\n");
  }/* end switch  */
}/* end else  */
#ifdef   READDEBUG
printf("Out read_newrule2()\n");
#endif

return((char*)ptr);

}/* end read_newrule2  */

char     *read_newrule1()
{
NEWRULE1_TYPE    ptr;
char     ch;
int     numchars, i, exists;

#ifdef  READDEBUG
printf("In read_newrule1()\n");
#endif

fscanf(fp_data,"%d\t%*s\n",&exists);
if(!exists)
  return(NULL);
else
{
  ptr = (NEWRULE1_TYPE)malloc(sizeof(STRUCT newrule1_type));
  fscanf(fp_data,"%d\t\t%*s\n",&(ptr->type));
  switch(ptr->type)
  {
  case 1:/* it is alternative no 1 */
      ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1 =
            (ATTRIBUTE_TYPE*)malloc(sizeof(ATTRIBUTE_TYPE));
      fscanf(fp_data,"%d\n",
      &(ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->      type));
```

```
        switch(ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->type)
        {
        case 1:
        fscanf(fp_data,"%d\n",&(
              ptr -> RIGHTSIDE.ALTER_newrule1_1.INT_1->textval.d));
        break;
        case 2:
        fscanf(fp_data,"%c\n",&(
              ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->textval.c));
        break;
        case 3:
        ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->textval.s =
              malloc(2*sizeof(char));
        fscanf(fp_data,"%d\n",&numchars);
        if(numchars)
        {
        fscanf(fp_data,"%c",&(
        ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1-> textval.s[0]));
        ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->textval.s[1] = '\0';
        for(i=1;i<numchars;i++)
        {
            fscanf(fp_data,"%c",&ch);
            realloc_char(
            ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->t   extval.s,ch);
        }
        }
        break;
        default:printf("ERROR : wrong lexeme type read\n");

        }/* end switch  */
        break;
    default:printf("ERROR : wrong alternative number read\n");
    }/* end switch  */
}/* end else  */
#ifdef  READDEBUG
printf("Out read_newrule1()\n");
#endif

return((char*)ptr);

}/* end read_newrule1  */

char    *read_ProcDec()
{
PROCDEC_TYPE    ptr;
char   ch;
int    numchars, i, exists;

#ifdef  READDEBUG
printf("In read_ProcDec()\n");
#endif

fscanf(fp_data,"%d\t%*s\n",&exists);
if(!exists)
  return(NULL);
else
{
  ptr = (PROCDEC_TYPE)malloc(sizeof(STRUCT ProcDec_type));
  fscanf(fp_data,"%d\t\t%*s\n",&(ptr->type));
  switch(ptr->type)
  {
```

```
  case 1:/* it is alternative no 1 */
      ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1 = (ATTRIBUTE_TYPE*)
      malloc (sizeof (ATTRIBUTE_TYPE));
      fscanf(fp_data,"%d\n",&(
      ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->type));
      switch(ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->type)
      {
       case 1:
         fscanf(fp_data,"%d\n",&(
         ptr-> RIGHTSIDE.ALTER_ProcDec_1.IDR_1->textval.d));
         break;
          case 2:
         fscanf(fp_data,"%c\n",&(
         ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->textval.c));
         break;
        case 3:
         ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->textval.s =
                   malloc(2*sizeof(char));
         fscanf(fp_data,"%d\n",&numchars);
         if(numchars)
         {
             fscanf(fp_data,"%c",&(
             ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->textval.s[0]));
             ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->textval.s[1]
                                                             = '\0';
             for(i=1;i<numchars;i++)
             {
               fscanf(fp_data,"%c",&ch);
               realloc_char(ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->
                         textval.s,ch);
             }
         }
         break;
        default:printf("ERROR : wrong lexeme type read\n");

      }/* end switch  */
      ptr->RIGHTSIDE.ALTER_ProcDec_1.newrule3_2 =
                                   (NEWRULE3_TYPE)read_newrule3();
      ptr->RIGHTSIDE.ALTER_ProcDec_1.Series_3 =
                                   (SERIES_TYPE)read_Series();
      break;
  default:printf("ERROR : wrong alternative number read\n");
  }/* end switch  */
}/* end else  */
#ifdef  READDEBUG
printf("Out read_ProcDec()\n");
#endif

return((char*)ptr);
}

/***  the rest of the reader routines are cut      ***/
/***  douwn due to the length of these routines   ***/
```

```
                    /****   file = walker.c   ****/

#include    <stdio.h>
#include    "syntree.h"
#include    "define.h"
#include    "reader.c"

      /*  global variables    */

int         PSused; /* index to the highest used program store location  */
int         DSused; /*   index to highest used data store location     */
int         numidrs;         /*  number of idrs found   */
int         numfps;          /*  number of formal para. */
int         numaps;          /*  number of actual para. */
int         STindex;         /*  symbol table index     */
int         currentprocref;/*  should be <= maxidrs   */
int         numerrs;         /*  number of errors found */
char        *idrchars;       /*  holds the identifier   */
char        *malloc();
BOOLEAN     textual;
BOOLEAN     hashflag;

      /*   Start of the program     */
main()
{
PROGRAM_TYPE   ptr;
int    i;

  numidrs = 0;
  numfps = 0;
  numaps = 0;
  STindex = 0;
  numerrs = 0;
  currentprocref = 0;
  PSused = 0;
  DSused = -1;
  textual = FALSE;
  hashflag = FALSE;

  printf("\nStart reading in the tree:-- \n");
  ptr = read_Program();
  printf("Finished reading in the tree. \n\n");

  for(i=1;i<=Maxidrs;i++)
  {
      symtab[i].idrname = NULL;
      symtab[i].class = 0;
      symtab[i].idrusage.var_rtsl = -1;
      symtab[i].idrusage.PN.entryaddr = -1;
      symtab[i].idrusage.PN.numparam = 0;
      symtab[i].idrusage.FP.procref = -1;
      symtab[i].idrusage.FP.paramnum = 0;
  }

  for(i=0; i<PSsize; i++)
  {
      DS[i] = -9;
      PS[i] = -9;
  }
```

```
  walk_Program(ptr);
  printf("\n**** MSL compilation complete - %d errors reported
****\n",numerrs);
  ListSymTab();
  ListTMcode();

}/* end of main() */

walk_Program(ptr)
PROGRAM_TYPE   ptr;
{
BOOLEAN   anyprocs = FALSE;
int             jmain;

if(ptr != NULL)
{
  switch(ptr->type){
  case 1:/* it is alternative no 1 */
      walk_newrule1(ptr->RIGHTSIDE.ALTER_Program_1.newrule1_1);
      if(ptr->RIGHTSIDE.ALTER_Program_1.newrule2_2)
      {
            anyprocs = TRUE;
            cg2(J,0);
            jmain = PSused;
      }
      walk_newrule2(ptr->RIGHTSIDE.ALTER_Program_1.newrule2_2);
      if(anyprocs)
      PS[jmain] = PSused+1;     /* fix up jump to code for main program */
      walk_Series(ptr->RIGHTSIDE.ALTER_Program_1.Series_3);
      break;
  default:printf("ERROR : Wrong alternative number generated\n");

  }/* end switch  */
  cg1(HALT);/* end of msl program  */

}/* end if */
}/* end walk_Program */

walk_newrule2(ptr)
NEWRULE2_TYPE   ptr;
{

if(ptr != NULL)
{
  switch(ptr->type){
  case 1:/* it is alternative no 1 */
      walk_newrule2(ptr->RIGHTSIDE.ALTER_newrule2_1.newrule2_1);
      walk_ProcDec(ptr->RIGHTSIDE.ALTER_newrule2_1.ProcDec_2);
      break;
  default:printf("ERROR : Wrong alternative number generated\n");
  }/* end switch  */
}/* end if */
}/* end walk_newrule2 */

walk_newrule1(ptr)
NEWRULE1_TYPE   ptr;
{

if(ptr != NULL)
{
  switch(ptr->type){
```

```
  case 1:/* it is alternative no 1 */
    switch(ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->type){
    case 1:/* it is a number */
      DSused = ptr->RIGHTSIDE.ALTER_newrule1_1.INT_1->textval.d;
      break;
    case 2:/* it is a character */ break;
    case 3:/* it is a string */ break;
    default:printf(" ERROR : Wrong lexeme type generated\n");
       }/* end switch */
      break;
  default:printf("ERROR : Wrong alternative number generated\n");
  }/* end switch  */
}/* end if */
}/* end walk_newrule1 */


walk_ProcDec(ptr)
PROCDEC_TYPE   ptr;
{
if(ptr != NULL)
{
  switch(ptr->type){
  case 1:/* it is alternative no 1 */
      switch(ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->type){
      case 1:/* it is a number */ break;
      case 2:/* it is a character */ break;
      case 3:/* it is a string */
            idrchars = malloc(strlen(ptr->
            RIGHTSIDE.ALTER_ProcDec_1.IDR_1-> textval.s)+1);
            strcpy(idrchars,
            ptr->RIGHTSIDE.ALTER_ProcDec_1.IDR_1->textval.s);
            checkprocidr();
            break;
      default:printf(" ERROR : Wrong lexeme type generated\n");
      }/* end switch */
      walk_newrule3(ptr->RIGHTSIDE.ALTER_ProcDec_1.newrule3_2);
      if(currentprocref != 0)
      if(symtab[currentprocref].class == Procname)
            symtab[currentprocref].idrusage.PN.numparam = numfps;
      numfps = 0;
      walk_Series(ptr->RIGHTSIDE.ALTER_ProcDec_1.Series_3);
      cg1(RTN);
      currentprocref = 0;
      break;
  default:printf("ERROR : Wrong alternative number generated\n");
  }/* end switch  */
}/* end if */
}/* end walk_ProcDec */

/***  the rest of the walker routines are cut     ***/
/***  douwn due to the length of these routines   ***/

/************************ STlookup() ************************/

STlookup(pos, newidr)
int           *pos;
BOOLEAN       *newidr;
{
      /* As given in Appendix A

}/* end of STlookup()  */
```

```
/********************* checkdeclared() *******************/

checkdeclared(STpos)
int   *STpos;
{
      /* As given in Appendix A

}/* end of checkdeclared() */

/********************* checkprocidr() *******************/

checkprocidr()
{
      /* As given in Appendix A

}/* end of checkprocidr()  */

/********************* checkFPcidr() ********************/

checkFPidr()
{
      /* As given in Appendix A

}/* end of checkFPidr()  */

/******************* CGloadcontents() ******************/

CGloadcontents(STref)
int     STref;
{
      /* As given in Appendix A

}/* end of CGloadcontents()  */

/********************* getRTSL() ********************/

getRTSL(STref)
int     STref;
{
      /* As given in Appendix A

}/* end of getRTSL() */

/******************** textaddress() ********************/

textaddress(ptr)
char  *ptr;
{
      /* As given in Appendix A

}/* end of textaddress() */


/********************* cg1(), cg2(), cg3() *******************/

cg1(PSitem)
int     PSitem;
{
      /* As given in Appendix A
```

```
}

cg2(opcode, opnd)
int      opcode;
int      opnd;
{
      /* As given in Appendix A

}

cg3(opcode, opnd1, opnd2)
int      opcode;
int      opnd1;
int      opnd2;
{
      /* As given in Appendix A

}

/*********************  ListSymTab()  ********************/

ListSymTab()
{
      /* As given in Appendix A

}/* end of ListSymTab() */

/*********************  CSerror()  *********************/

CSerror(errtype)
int      errtype;
{
      /* As given in Appendix A

}/* end of CSerror  */


/*********************  ListTMcode()  *********************/

ListTMcode()
{
      /* As given in Appendix A

}/* end of ListTMcode() */
```

/**   file = define.h   ****/**

```
          /* define constants for TM operation codes */

#define LD                1
#define SD                2
#define LC                3
#define LA                4
#define J                 5
#define JF                6
#define JT                7
#define JSR               8
```

```
#define LI                  9
#define SI                  10
#define WNUM                11
#define RNUM                12
#define WTXT                13
#define RTXT                14
#define WNL                 15
#define RNL                 16
#define CALL                17
#define RTN                 18
#define HALT                19
#define ADDop               20
#define SUBop               21
#define MULTop              22
#define DVDop               23
#define LTop                24
#define LEop                25
#define GTop                26
#define GEop                27
#define NEop                28
#define EQop                29
#define LOGORop             30
#define LOGANDop            31

#define TMfalse             0
#define TMtrue              1

#define  Procname           100
#define  Formalparam        101
#define Variable            102
#define Undefined           103

              /* define constants for error messages  */

#define  PrNameNotNew       104
#define FPnotNew            105
#define NotaProcName        106
#define WrongNumOfAPs       107
#define BadProcNameUs       108
#define BadFPusage          109
#define ProcNotDec          110
#define Lv                  111
#define Rv                  112

#define BOOLEAN             int
#define FALSE               0
#define TRUE                1

#define Maxidrs             100 /* maximum number of identifie      */
#define PSsize              400 /* maximum size of the prog. store  */
#define DSsize              400 /* maximum size of the data store   */

typedef struct stentry{
   char   *idrname;    /* identifier name      */
   int    class;       /* the identifier class */
   union   {
        int   var_rtsl; /* Run-time store location */
        struct  procname{
           int    entryaddr;  /* entry addresse          */
           int numparam;      /* number of parameters    */
        }PN;
```

```
        struct    formalparam {
           int    procref;     /* procedure reference         */
           int paramnum;       /* parameter number (position) */
        }FP;
   }idrusage; /* end of union  */
}STENTRY;

typedef    struct opcode{
   int     tokennum;
   int     tmcode;
}OPCODE;

STENTRY     symtab[Maxidrs];
int         PS[PSsize];
int         DS[DSsize];
static OPCODE codes[] = {{260,ADDop}, {261,SUBop}, {259,MULTop},
{262,DVDop}, {263,LTop}, {264,LEop}, {267,GTop}, {268,GEop},
{265,NEop}, {266,EQop}, {257,LOGORop}, {258,LOGANDop}};
```

Appendix C

This appendix contains four MSL source programs together with the results produced by the hand-written version, the lex and yacc version and the CORGI version of the compiler. The results given here are produced by CORGI, and they have been shown to be exactly the same to those produced by the hand-written and the lex and yacc versions. Thus the files given in this appendix are set of program/result pairs, these are: `msl.prog1/res.prog1`, `msl.prog2/res.prog2`, `msl.prog3/res.prog3`, `msl.prog4/res.prog4`. msl.prog4 is the most representative example since it contains all the features of MSL.

/**** file = msl.prog1 ****/

```
RESERVE   10
PROC start
      s := 0
      WRITE "START"
END
PROC p(a,b,ptr)
      ptr!x := a + b
END
PROC q(j)
      CALL p(j,k+1,@z)
      x := x +1
END
CALL start
i := 1
CALL r(x+1)
j := 1
.
```

/**** file = result.prog1 ****/

```
+++ Context error detected - procedure not declared
+++ Context error detected - improper use of formal parameter

****  MSL compilation complete - 2 errors reported ****
****  Symbol Table    ****

Entry no.   Identifier      Class                 Data
---------   ----------      ------                -----
   1          start         ProcName              3      0
   2          s             Variable              11     --
```

```
   3          p              ProcName              12     3
   4          a              FormalParam           3      1
   5          b              FormalParam           3      2
   6          ptr            FormalParam           3      3
   7          x              Variable              18     --
   8          q              ProcName              24     1
   9          j              FormalParam           8      1
   10         k              Variable              19     --
   11         z              Variable              20     --
   12         i              Variable              21     --
   13         r              Undefined             --     --
```

```
****    TM code   ****

1)   J   45         3)   LC   0       5)   SD   11      7)   LC   12
9)   WTXT           10)   WNL         11)   RTN         12)   LA   3
14)   LD   18       16)   ADD         17)   LA   1      19)   LA   2
21)   ADD           22)   SI          23)   RTN         24)   CALL
25)   LA   1        27)   LD   19     29)   LC   1      31)   ADD
32)   LC   20       34)   JSR 12 3   37)   LD   18      39)   LC   1
41)   ADD           42)   SD   18     44)   RTN         45)   CALL
46)   JSR 3 0       49)   LC   1      51)   SD   21     53)   CALL
54)   LD   18       56)   LC   1      58)   ADD         59)   LC   1
61)   SD   0        63)   HALT

Compilation error(s): 0
Compilation terminated
```

/**** **file = msl.prog2** ****/

```
PROC newmax(m)
     IF m > max THEN
        max := m
     FI
END
max := 0
s := 0
i := 1
WHILE   i <= 20  DO
      READ + s!i
      CALL  newmax(s!i)
      i := i + 1
OD
.
```

/**** **file = result.prog2** ****/

```
**** MSL compilation complete - 0 errors reported ****
****  Symbol Table   ****

Entry no.   Identifier       Class           Data
--------    ----------       ---------       ------
   1          newmax         ProcName        3     1
```

```
   2          m              FormalParam      1      1
   3          max            Variable         0      --
   4          s              Variable         1      --
   5          i              Variable         2      --


****  TM code  ****

1)  J  15          3)  LA  1       5)  LD  0       7)  GT
8)  JF  14        10)  LA  1      12)  SD  0      14)  RTN
15)  LC  0        17)  SD  0      19)  LC  0      21)  SD  1
23)  LC  1        25)  SD  2      27)  LD  2      29)  LC  20
31)  LE           32)  JF  59     34)  LD  1      36)  LD  2
38)  ADD          39)  RNUM       40)  CALL       41)  LD  1
43)  LD  2        45)  ADD        46)  LI         47)  JSR  3  1
50)  LD  2        52)  LC  1      54)  ADD        55)  SD  2
57)  J  27        59)  HALT

Compilation error(s): 0
Compilation terminated
```

/**   file = msl.prog3   ****/**

```
PROC subt(a,b,c)
     c!0 := a - b
END

n := 0
READ  p, q

WHILE p >= q  DO
      CALL subt(p,q,@p)
      n := n + 1
OD

WRITE "Result = ", n
.
```

/**   file = result.prog3   ****/**

```
**** MSL compilation complete - 0 errors reported ****
****  Symbol Table   ****

Entry no.   Identifier      Class           Data
--------    ----------      ---------       ---------
   1          subt          ProcName        3      3
   2          a             FormalParam     1      1
   3          b             FormalParam     1      2
   4          c             FormalParam     1      3
   5          n             Variable        0      --
   6          p             Variable        1      --
   7          q             Variable        2      --
```

```
****  TM code  ****

1)  J  15        3)  LA  3      5)  LC  0      7)  ADD
8)  LA  1        10)  LA  2     12)  SUB       13)  SI
14)  RTN         15)  LC  0     17)  SD  0     19)  LC  1
21)  RNUM        22)  LC  2     24)  RNUM      25)  RNL
26)  LD  1       28)  LD  2     30)  GE        31)  JF  52
33)  CALL        34)  LD  1     36)  LD  2     38)  LC  1
40)  JSR 3 3     43)  LD  0     45)  LC  1     47)  ADD
48)  SD  0       50)  J  26     52)  LC  3     54)  WTXT
55)  LD  0       57)  WNUM      58)  WNL       59)  HALT

Compilation error(s): 0
Compilation terminated
```

/**** <u>file = msl.prog4</u> ****/

```
RESERVE 110
PROC Initialise
   title := 0
   NIL := 0-1
   freespaceptr := 21
   treehead := NIL
   seq := 0
END
PROC putin(ptrtoNILnode, inval)
   ptrtoNILnode!0 := freespaceptr
   freespaceptr := freespaceptr+3
   newnodeptr := ptrtoNILnode!0
   newnodeptr!0 := inval
   newnodeptr!1 := NIL
   newnodeptr!2 := NIL
END
PROC insert(newval)
   placeptr := @treehead
   seeking := 1
   WHILE seeking DO
      nodeptr := placeptr!0
      IF nodeptr = NIL
      THEN CALL putin(placeptr, newval)
           seeking := 0
      ELSE IF newval < nodeptr!0 THEN placeptr := nodeptr+1
      ELSE IF newval > nodeptr!0 THEN placeptr := nodeptr+2
           ELSE seeking := 0
           FI
      FI
      FI
   OD
END
PROC treeprint(treeptr)
   IF treeptr <> NIL
   THEN CALL treeprint(treeptr!1)
        seq := seq+1
        WRITE seq, treeptr!0
        CALL treeprint(treeptr!2)
```

```
   FI
END
WRITE "MSL Tree Sort Program started"
WRITE ""
CALL Initialise
READ #title
WRITE #title, " - unsorted"
WRITE ""
READ + num
WHILE num <> 0 DO
   WRITE num
   CALL insert(num)
   READ + num
OD
WRITE ""
WRITE + #title
WRITE " - sorted"
WRITE ""
CALL treeprint(treehead)
WRITE ""
WRITE "Print done -", seq, " distinct values found"
.
```

/**** **file = result.prog4** ****/

```
**** MSL compilation complete - 0 errors reported ****
****  Symbol Table   ****
```

Entry no.	Identifier	Class	Data	
1	Initialise	ProcName	3	0
2	title	Variable	111	--
3	NIL	Variable	112	--
4	freespaceptr	Variable	113	--
5	treehead	Variable	114	--
6	seq	Variable	115	--
7	putin	ProcName	27	2
8	ptrtoNILnode	FormalParam	7	1
9	inval	FormalParam	7	2
10	newnodeptr	Variable	116	--
11	insert	ProcName	75	1
12	newval	FormalParam	11	1
13	placeptr	Variable	117	--
14	seeking	Variable	118	--
15	nodeptr	Variable	119	--
16	treeprint	ProcName	163	1
17	treeptr	FormalParam	16	1
18	num	Variable	164	--

```
****  TM code  ****

1)  J  209        3)  LC  0         5)  SD  111      7)  LC  0
9)  LC  1         11)  SUB          12)  SD  112     14)  LC  21
16)  SD  113      18)  LD  112      20)  SD  114     22)  LC  0
24)  SD  115      26)  RTN          27)  LA  1       29)  LC  0
31)  ADD          32)  LD  113      34)  SI          35)  LD  113
37)  LC  3        39)  ADD          40)  SD  113     42)  LA  1
44)  LC  0        46)  ADD          47)  LI          48)  SD  116
```

```
50)  LD  116     52)  LC  0       54)  ADD         55)  LA  2
57)  SI          58)  LD  116     60)  LC  1       62)  ADD
63)  LD  112     65)  SI          66)  LD  116     68)  LC  2
70)  ADD         71)  LD  112     73)  SI          74)  RTN
75)  LC  114     77)  SD  117     79)  LC  1       81)  SD  118
83)  LD  118     85)  JF  162     87)  LD  117     89)  LC  0
91)  ADD         92)  LI          93)  SD  119     95)  LD  119
97)  LD  112     99)  EQ          100)  JF  116    102)  CALL
103)  LD  117    105)  LA  1      107)  JSR 27 2   110)  LC  0
112)  SD  118    114)  J  160     116)  LA  1      118)  LD  119
120)  LC  0      122)  ADD        123)  LI         124)  LT
125)  JF  136    127)  LD  119    129)  LC  1      131)  ADD
132)  SD  117    134)  J  160     136)  LA  1      138)  LD  119
140)  LC  0      142)  ADD        143)  LI         144)  GT
145)  JF  156    147)  LD  119    149)  LC  2      151)  ADD
152)  SD  117    154)  J  160     156)  LC  0      158)  SD  118
160)  J  83      162)  RTN        163)  LA  1      165)  LD  112
167)  NE         168)  JF  208    170)  CALL       171)  LA  1
173)  LC  1      175)  ADD        176)  LI         177)  JSR 163 1
180)  LD  115    182)  LC  1      184)  ADD        185)  SD  115
187)  LD  115    189)  WNUM       190)  LA  1      192)  LC  0
194)  ADD        195)  LI         196)  WNUM       197)  WNL
198)  CALL       199)  LA  1      201)  LC  2      203)  ADD
204)  LI         205)  JSR 163 1  208)  RTN        209)  LC  120
211)  WTXT       212)  WNL        213)  LC  150    215)  WTXT
216)  WNL        217)  CALL       218)  JSR 3 0    221)  LD  111
223)  RTXT       224)  RNL        225)  LD  111    227)  WTXT
228)  LC  151    230)  WTXT       231)  WNL        232)  LC  163
234)  WTXT       235)  WNL        236)  LC  164    238)  RNUM
239)  LD  164    241)  LC  0      243)  NE         244)  JF  261
246)  LD  164    248)  WNUM       249)  WNL        250)  CALL
251)  LD  164    253)  JSR 75 1   256)  LC  164    258)  RNUM
259)  J  239     261)  LC  165    263)  WTXT       264)  WNL
265)  LD  111    267)  WTXT       268)  LC  166    270)  WTXT
271)  WNL        272)  LC  176    274)  WTXT       275)  WNL
276)  CALL       277)  LD  114    279)  JSR 163 1  282)  LC 177
284)  WTXT       285)  WNL        286)  LC  178    288)  WTXT
289)  LD  115    291)  WNUM       292)  LC  191    294)  WTXT
295)  WNL        296)  HALT

Compilation error(s): 0
Compilation terminated
```

Appendix D

This appendix contains a demonstration of the error-recovery mechanism incorporated in CORGI system. A generated yacc specification which deals only with the error-recovery (no tree is being built) is also given. Shorter demonstrations were already discussed in chapter 7. The demonstration given in this appendix consists of running an erroneous MSL program through the generated front-end. The results shown here are produced from the CORGI system running on HLH ORION 1.05.

```
/****  A generated yacc specification for    ****/
/****  MSL that does not contain actions      ****/
/****  for building the AST. It deals only   ****/
/****  only with the error-recovery           ****/


/*  operator tokens  */

%token     OP1          /*    OP1  = "&"          */
%token     OP2          /*    OP2  = "&"          */
%token     OP5          /*    OP5  = "*"          */
%token     OP3          /*    OP3  = "+"          */
%token     OP4          /*    OP4  = "-"          */
%token     OP6          /*    OP6  = "/"          */
%token     OP7          /*    OP7  = "<"          */
%token     OP8          /*    OP8  = "<="         */
%token     OP9          /*    OP9  = "<>"         */
%token     OP12         /*    OP12  = "="         */
%token     OP11         /*    OP11  = ">"         */
%token     OP10         /*    OP10  = ">="        */

/*  delimiter tokens  */

%token     DEL1         /*    DEL1  = "!"         */
%token     DEL2         /*    DEL2  = "("         */
%token     DEL3         /*    DEL3  = ")"         */
%token     DEL4         /*    DEL4  = ","         */
%token     DEL5         /*    DEL5  = "."         */
%token     DEL6         /*    DEL6  = ":="        */
%token     DEL7         /*    DEL7  = "@"         */
%token     DEL8         /*    DEL8  = "\""        */
%token     DEL9         /*    DEL9  = "#"         */

/*  lexeme tokens  */

%token     IDR
%token     INT
%token     TEXT
```

```
%token     BOOLEAN

/*  The keyword tokens  */

%token     CALLSYM
%token     DOSYM
%token     ELSESYM
%token     ENDSYM
%token     FISYM
%token     IFSYM
%token     ODSYM
%token     PROCSYM
%token     READSYM
%token     RESERVESYM
%token     THENSYM
%token     WHILESYM
%token     WRITESYM
%token     TRUESYM
%token     FALSESYM

/* Operator precedence and associativity  */

%left   OP1 OP2 OP5 OP3 OP4 OP6 OP7 OP8 OP9 OP12 OP11 OP10

%%

Start: Program;

Program: newrule1 newrule2 Series DEL5
       | newrule1 newrule2 Series error;

newrule2: /* empty */
        |  newrule2 ProcDec { yyerrok;}
        |  newrule2 error;

newrule1: /* empty */
        |  RESERVESYM INT
        |  RESERVESYM error;

ProcDec: PROCSYM IDR newrule3 Series ENDSYM
       | PROCSYM error

newrule3: /* empty */
        |  DEL2 Formalparams DEL3
        |  DEL2 error;

Formalparams: IDR newrule4
            | IDR error;

newrule4:    /* empty */
        |  newrule4 DEL4 IDR  { yyerrok;}
        |  newrule4 error IDR { yyerrok;}
        |  newrule4 error
        |  newrule4 error IDR { yyerrok;}
        |  newrule4 DEL4 error;

Series: Stmt newrule5;

newrule5: /* empty */
        |  newrule5 Stmt { yyerrok;}
        |  newrule5 error;
```

```
Stmt: Assignst |  Whilest |  Ifst |  Callst |  Readst |  Writest;

Assignst: StoreAccess DEL6 Expr| StoreAccess error Expr;

Whilest: WHILESYM Expr DOSYM Series ODSYM
       | WHILESYM error;

Ifst: IFSYM Expr THENSYM Series newrule6 FISYM
    | IFSYM error;

newrule6: /* empty */
        |  ELSESYM Series
        |  ELSESYM error;

Callst: CALLSYM IDR newrule7
      | CALLSYM error;

newrule7: /* empty */
        |  DEL2 Exprlist DEL3
        |  DEL2 error;

Exprlist: Expr newrule8;

newrule8: /* empty */
        |  newrule8 DEL4 Expr { yyerrok;}
        |  newrule8 error
        |  newrule8 error Expr { yyerrok;}
        |  newrule8 DEL4 error;

Readst: READSYM Optionalplus Optionalhash Readinlist
      | READSYM error;

Writest: WRITESYM Optionalplus Optionalhash Writeoutlist
       | WRITESYM error;

Readinlist: StoreAccess newrule9;

newrule9: /* empty */
        |  newrule9 DEL4 Optionalhash StoreAccess  { yyerrok;}
        |  newrule9 error
        |  newrule9 error Optionalhash StoreAccess { yyerrok;}
        |  newrule9 DEL4 error StoreAccess { yyerrok;}
        |  newrule9 DEL4 Optionalhash error;

Writeoutlist: Expr newrule10;

newrule10: /* empty */
        |  newrule10 DEL4 Optionalhash Expr { yyerrok;}
        |  newrule10 error
        |  newrule10 error Optionalhash Expr { yyerrok;}
        |  newrule10 DEL4 error Expr { yyerrok;}
        |  newrule10 DEL4 Optionalhash error;

Optionalplus: /* empty */
            |  OP3;

Optionalhash: /* empty */
            |  DEL9;

Expr: Operand newrule11;
```

```
newrule11: /* empty */
          |  newrule11 OP1 Operand
          |  newrule11 OP2 Operand
          |  newrule11 OP3 Operand
          |  newrule11 OP4 Operand
          |  newrule11 OP5 Operand
          |  newrule11 OP6 Operand
          |  newrule11 OP7 Operand
          |  newrule11 OP8 Operand
          |  newrule11 OP9 Operand
          |  newrule11 OP10 Operand
          |  newrule11 OP11 Operand
          |  newrule11 OP12 Operand
          |  newrule11 error Operand;

Operand:  INT
        |  TEXT
        |  TRUESYM
        |  FALSESYM
        |  DEL7 IDR
        |  DEL7 error
        |  StoreAccess
        |  DEL2 Expr DEL3
        |  DEL2 error;

StoreAccess: IDR newrule12
            | IDR error;

newrule12: /* empty */
         |  DEL1 Operand
         |  DEL1 error;

%%
main()
{
#ifdef LEXDEBUG
  debug = 1;          /* debug = 1 for debugging and 0 otherwise */
#endif
extern  int  yynerrs;
int flag = 0;

      flag = yyparse();
      printf("Compilation error(s): %d\n", yynerrs);
      if(flag)
         printf("Compilation aborted\n");
      else
         printf("Compilation terminated\n");
}
```

/**** File = MSL erroneous program ****/

```
RESERVE   20
PROC   start
      s  0
      WRITE  "START",
      -- should be END here
PROC p (a,, b, ptr)
      ptr!x := a + b
      IF true THEN
             j := b
      -- FI should be here
END
PROC q(i,j k)
      CALL p(j, k+1, @z)
      x = x  1
END
WHILE i < 10 DO
      CALL q(a, b)
      i := i + 1
      -- OD  should be here
CALL start
i  30
CALL r(x+1)
j + 1
.
```

/**** Error report from the above ****/
/**** MSL erroneous program ****/

```
[error 1] line 5 near "0": expecting:  DEL6
[error 2] line 10 near "PROC": expecting:  DEL2 DEL7 IDR INT TEXT
TRUESYM FALSESYM
[error 3] line 10 near "PROC": expecting:  ENDSYM
[error 4] line 10 near ",": expecting:  IDR
[error 5] line 16 near "END": expecting:  FISYM
[error 6] line 18 near "k": expecting:  DEL3
[error 7] line 21 near "=": expecting:  DEL6
[error 8] line 21 near "1": expecting:  ENDSYM
[error 9] line 21 near "1": expecting:  IDR CALLSYM IFSYM PROCSYM
READSYM WHILESYM WRITESYM
[error 10] line 32 near "30": expecting:  DEL6
[error 11] line 34 near "+": expecting:  DEL6
[error 12] line 35 near ".": expecting:  ODSYM

Compilation error(s): 12
Compilation terminated
```

Appendix E

This appendix contains CORGI specifications for full Modula-2 together with the test programs used to run the CORGI generated front-end to show that the generated parser successfully parses the test programs provided. The same is done for Pascal.

The test programs are taken from Koffman (1988) for Modula-2 and Findlay & Watt (1881) for Pascal .

```
/****  file: CORGI specification for Modula2  ****/

STARTCOMMENT       "(*"      NESTED.
ENDCOMMENT         "*)".
LEXEME              ident string integer real.
KEYWORD_CASE       KEY_SIG.
OPERATORS
MulOperator = \L { "*" "/" "DIV" "MOD" "AND" }.
AddOperator = \L { "+" "-" "OR" }.
relation    = \L { "<" "<=" "#" "=>" ">" "=" "IN" }.

%%

CompilationUnit = DefinitionModule
                | ["IMPLEMENTATION"] ProgramModule.

ProgramModule = "MODULE" ident [priority] ";" {import} block ident
                ".".

ident = letter { letter | digit }        ==> evaluate_idr (0).

letter = uplow_case.

priority = "[" ConstExpression "]".

ConstExpression = expression.

import = [ "FROM" ident ] "IMPORT" IdentList ";".

IdentList = ident { "," ident }.

block = { declaration } ["BEGIN" StatementSequence] "END".

declaration = "CONST" { ConstantDeclaration ";" }
            | "TYPE" { TypeDeclaration ";" }
            | "VAR" { VariableDeclaration ";" }
            | ProcedureDeclaration ";"
            | ModuleDeclaration ";".
```

```
ConstantDeclaration = ident "=" ConstExpression.

TypeDeclaration = ident "=" type.

VariableDeclaration = IdentList ":" type.

ProcedureDeclaration = ProcedureHeading ";" block ident.

ProcedureHeading = "PROCEDURE" ident [ FormalParameters ].

FormalParameters = "(" [ FPSection { ";" FPSection } ] ")".

FPSection = ["VAR"] IdentList ":" FormalType.

FormalType = ["ARRAY" "of"] qualident.

qualident = ident {"." ident }.

type = SimpleType | ArrayType | RecordType | SetType
     | PointerType | ProcedureType.

SimpleType = qualident | enumeration | SubrangeType.

enumeration = "(" IdentList ")".

SubrangeType = [ qualident ] "[" ConstExpression
                ".." ConstExpression "]".

ArrayType = "ARRAY" SimpleType {"," SimpleType} "OF" type.

RecordType = "RECORD" FieldListSequence "END".

FieldListSequence = FieldList { ";" FieldList }.

FieldList = [ IdentList ":" type | "CASE" [ident] ":" qualident
"OF" variant { "|" variant } [ "ELSE" FieldListSequence ] "END"].

variant = [CaseLabelList ":" FieldListSequence ].

CaseLabelList = CaseLabels {"," CaseLabels }.

CaseLabels = ConstExpression [ ".." ConstExpression ].

SetType = "SET" "OF" SimpleType.

PointerType = "POINTER" "TO" type.

ProcedureType = "PROCEDURE" [FormalTypeList].

FormalTypeList = "(" [ ["VAR"] FormalType { "," ["VAR"]
                 FormalType }] ")" [ ":" qualident ].

expression = SimpleExpression [ relation SimpleExpression ].

SimpleExpression = [sign] term { AddOperator term }.

sign = "+" | "-".

term = factor { MulOperator factor }.

factor = number | string | set | designator [ ActualParameters]
```

```
        | "(" expression ")" | "NOT" factor.

set = [ qualident ] "{" [ element {"," element }]"}".

element = expression [ ".." expression ].

ActualParameters = "(" [ ExpList ] ")".

ExpList = expression { "," expression }.

designator = qualident {"." ident | "[" ExpList "]" | "^" }.

number = integer | real.

integer = digit{digit}                        ==> evaluate_denary
        | octalDigit { octalDigit} ( "B" |"C")
                                              ==> evaluate_Mod2_Octal
        | digit { hexDigit} "H"               ==> evaluate_Mod2_Hex.

hexDigit = hexdigit.

octalDigit = octaldigit.

real = digit {digit} "."{digit} [ScaleFactor].

string = "'" {anybut_SQ_NL} "'"                   ==> evaluate_Pstr
       | """" {anybut_DQ_NL} """"                 ==> evaluate_Pstr.

ScaleFactor = "E" [ sign ] digit {digit}.

StatementSequence = statement {";" statement}.

statement = [assignment | ProcedureCall | IfStatement
          | CaseStatement| WhileStatement | RepeatStatement
          | LoopStatement | ForStatement | WithStatement
          | "EXIT" | "RETURN" [expression]].

assignment = designator ":=" expression.

ProcedureCall = designator [ ActualParameters ].

IfStatement = "IF" expression "THEN" StatementSequence
            { "ELSIF" expression "THEN" StatementSequence }
            [ "ELSE" StatementSequence ] "END".

CaseStatement = "CASE" expression "OF" case { "|" case }
                [ "ELSE" StatementSequence ]  "END".

case = [ CaseLabelList ":" StatementSequence ].

WhileStatement = "WHILE" designator "DO" StatementSequence "END".

RepeatStatement = "REPEAT" StatementSequence "UNTIL" expression.

ForStatement = "FOR" ident ":=" expression "TO" expression
             ["BY" ConstExpression] "DO" StatementSequence "END".

LoopStatement = "LOOP" StatementSequence "END".

WithStatement = "WITH" designator "DO" StatementSequence "END".
```

```
ModuleDeclaration = "MODULE" ident [priority] ";"
                     {import} [export] block ident.

export = "EXPORT" ["QUALIFIER"] IdentList ";".

DefinitionModule = "DEFINITION" "MODULE" ident ";"
                     {import}{definition}"END" ".".

definition = "CONST" {ConstantDeclaration ";"}
           | "TYPE" { ident [ "=" type] ";"}
           |"VAR" {VariableDeclaration ";"}
           | ProcedureHeading ";".
```

/**** <u>file: Modula2 test progam "modula.progl"</u> ****/

```
MODULE multipay;
(* finds and prints gross pay and net pay for several employees.*)

(* THIS LINE JUST TESTS (* NESTED COMMENTS *) USED IN CORGI *)

      FROM InOut  IMPORT
            ReadCard, WriteString, WriteLn;
      FROM ReadInOut IMPORT
            ReadReal, WriteReal;
      TYPE
            hello = ARRAY [0..9] OF CHAR;
      VAR
        numemp,          (* input - total number of employees *)
        contemp : CARDINAL; (* loop control - count of employees
                              processed *)
      PROCEDURE modplay;
            CONST
              taxbracket = 100;
              tax = 25;            (* tax amount   *)
            VAR
              hours, rate, (* input - hours worked, hourly rate *)
              gross, net : REAL; (* output - gross pay, net pay *)

      BEGIN (* modpay  *)
            (* enter hours and rate *)
            WriteString ('Hours worked? ');
            ReadRead (hours);  WriteLn;
            WriteString ('Hourly rate? ');
            readread (rate);  WriteLn;

            (* computer gross salary  *)
            gross := hours * rate;

            (* computer net salary *)
            IF gross > taxbracket THEN
                  net := gross - tax  (* deduct a tax amount   *)
            ELSE
                  net := gross         (*  deduct no tax  *)
            END; (* IF  *)

            (*  print gross and net  *)
            WriteString ('gross salary is $');
            WriteReal (gross,12);  WriteLn;
            WriteString ('net salary is $');
            WriteReal (net,12);  WriteLn
      END modpay;

 BEGIN  (* multiplay  *)
      (*  enter total number of employees  *)
      WriteString ('how many employees? ');
      ReadCard (numemp);  WriteLn;

      (* computer gross pay and net pay for numemp employees  *)
      FOR countemp := 1 TO numemp DO
            modpay;             (*  process next employee  *)
            WriteLn
      END  (*  FOR  *)
 END multipay.
```

/**** <u>file: Modula2 test progam "modula.prog2"</u> ****/

```
MODULE PrintReverse;
(* reads a sequence of characters and displays it in reverse
order. Uses the abstract data type stack.
*)

    FROM InOut IMPORT
      Write, Read, WriteString, WriteLn, EOL;

    FROM StackADT IMPORT
      Stack,                         (* data type  *)
      CreateStack, Push, Pop,        (* procedures *)
      IsFull, IsEmpty;               (* functions *)

    VAR
      S : Stack;                     (* the next character *)

    PROCEDURE FillStack (VAR S (* in/out*) : Stack);

 (* Reads data (* characters and pushes *) them  onto the stack.*)

      VAR
        NextCh : CHAR;               (* the next character *)
        Success : BOOLEAN;           (* flag *)
    BEGIN (* FillStack *)

      WriteString ('Enter a string of one or more characters.');
      WriteLn;  WriteString ('Press return when done.');
      WriteLn;
      REPEAT
       Read(NextCh); Write(NextCh); (* read and echo character *)
       Push (S, NextCh, Success)          (* push it onto stack *)
      UNTIL (NextCh = EOL) OR NOT Success;
      WriteLn;

      (* Print an error if stack overflows. *)
      IF NOT Success THEN
         WriteString ('Stack overflow error - string too long');
         WriteLn
      END  (* IF *)
    END FillStack;

    PROCEDURE DisplayStack (VAR S (* in/out *) : Stack);

    (* Pops each stack (* character and *) displays iy. *)

      VAR
        NextCh : CHAR;          (* the next character *)
        Success : BOOLEAN;            (* flag *)

    BEGIN (* DisplayStack *)
      Pop (S, NextCh, Success);
      WHILE Success DO
            Write (NextCh);
            Pop (S, NextCh, Success)
      END; (* WHILE *)
      WriteLn
    END DisplayStack;

 BEGIN (* PrintReverse *)
```

```
    CreateStack (S);                (* start with an empty stack *)

    (* Fill the stack *)
    FillStack (S);

    (* Display the characters in reverse order *)
    DisplayStack (S);

    (* Display status of stack S *)
    IF IsEmpty (S) THEN
      WriteString('Stack is empoty')
    ELSIF IsFull (S) THEN
      WriteString ('Stack is full')
    END;
    WriteLn
END PrintReverse.
```

/**** file: CORGI specification for Pascal ****/

```
STARTCOMMENT        "{" | "(*".
ENDCOMMENT          "}" "*)" | "}" "*)".
LEXEME              identifier character_string label
                    unsigned_integer unsigned_real.
KEYWORD_CASE        KEY_NONSIG.
OPERATORS
multiplying_operator     = \L { "*" "/" "DIV" "MOD" "AND" }.
adding_operator          = \L { "+" "-" "OR" }.
relational_operator      = \L { "<" "<=" "<>" "=>" ">" "=" "IN" }.

%%

program = program_heading ";" program_block ".".

program_heading = "program" identifier [ "("
                   program_parameters ")"].

program_block = block.

program_parameters = identifier_list.

identifier_list = identifier { "," identifier }.

identifier = letter { letter | digit }        ==>evaluate_idr (0).

block = label_declaration_part constant_definition_part
        type_definition_part variable_declaration_part
        procedure_and_function_declaration_part statement_part.

letter = uplow_case.

label_declaration_part = [ "label" label { "," label } ";" ].

label = digit_sequence                     ==> evaluate_denary.

digit_sequence = digit { digit }.

constant_definition_part = [ "const" constant_definition ";"
                             { constant_definition ";" } ].

constant_definition = identifier "=" constant.

constant = [sign] ( unsigned_number | constant_identifier )
         | character_string.

unsigned_number = unsigned_integer | unsigned_real.

sign = "+" | "-".

unsigned_integer = digit_sequence          ==>evaluate_denary.

unsigned_real = unsigned_integer "." fractional_part
                  [ "e" scale_factor ]
              |unsigned_integer  "e" scale_factor.

fractional_part = digit_sequence.

scale_factor = signed_integer.
```

```
signed_integer = [ sign ] unsigned_integer.

constant_identifier = identifier.

character_string = "'" string_element { string_element } "'"
                                              ==>evaluate_Pstr.

string_element = apostroph_image | string_character.

apostroph_image = "''".

string_character = anybut_NL.

type_definition_part = [ "type" type_definition ";"
                         { type_definition ";" } ].

type_definition = identifier "=" type_denoter.

type_denoter = type_identifier | new_type.

type_identifier = identifier.

new_type = new_ordinal_type | new_structured_type
         | new_pointer_type.

new_ordinal_type = enumerated_type | subrange_type.

enumerated_type = "(" identifier_list ")".

subrange_type = constant ".." constant.

new_structured_type = [ "packed" ] unpacked_structured_type.

unpacked_structured_type = array_type | record_type | set_type
                         | file_type.

array_type = "array" "[" index_type { "," index_type } "]"
             "of" component_type.

index_type = ordinal_type.

ordinal_type = new_ordinal_type | ordinal_type_identifier.

ordinal_type_identifier = type_identifier.

component_type = type_denoter.

record_type = "record" field_list "end".

field_list = [ ( fixed_part [ ";" variant_part ]
               | variant_part ) [ ";" ] ].

fixed_part = record_section { ";" record_section }.

record_section = identifier_list ":" type_denoter.

variant_part = "case" variant_selector "of" variant {";" variant}.

variant_selector = [ tag_field ":" ] tag_type.
```

```
tag_field = identifier.

tag_type = ordinal_type_identifier.

variant = case_constant_list ":" "(" field_list ")".

case_constant_list = case_constant { "," case_constant }.

case_constant = constant.

set_type = "set" "of" base_type.

base_type = ordinal_type.

file_type = "file" "of" component_type.

new_pointer_type = "^" domain_type.

domain_type = type_identifier.

variable_declaration_part = [ "var" variable_declaration ";"
                 { variable_declaration ";" } ].

variable_declaration = identifier_list ":" type_denoter.

procedure_and_function_declaration_part = [( procedure_declaration
       | function_declaration ) ";" ].

procedure_declaration = procedure_heading ";" identifier
                      | procedure_identification ";" procedure_block
                      | procedure_heading ";" procedure_block.

procedure_heading = "procedure" identifier
                        ["(" formal_parameter_list ")" ].

procedure_identification = "procedure" procedure_identifier.

procedure_identifier = identifier.

procedure_block = block.

formal_parameter_list = value_parameter_specification
                      | variable_parameter_specification
                      |procedural_parameter_specification
                      | functional_parameter_specification
                      | conformant_array_parameter_specification.

value_parameter_specification = identifier_list
                                ":" type_identifier.

variable_parameter_specification = "var" identifier_list
                                   ":" type_identifier.

procedural_parameter_specification = procedure_heading.

functional_parameter_specification = function_heading.

conformant_array_parameter_specification =
                value_conformant_array_specification
          |variable_conformant_array_specification.
```

```
value_conformant_array_specification = identifier_list
                                   ":" conformant_array_schema.

conformant_array_schema = packed_conformant_array_schema
                        | unpacked_conformant_array_schema.

packed_conformant_array_schema = "packed" "array"
"[" index_type_specification "]" "of" type_identifier.

index_type_specification = identifier ".." identifier
                        ":" ordinal_type_identifier.

unpacked_conformant_array_schema = "array"
"[" index_type_specification { ";" index_type_specification } "]"
"of" ( type_identifier | conformant_array_schema).

variable_conformant_array_specification = "var" identifier_list
                                       ":" type_identifier.

function_declaration = function_heading ";" identifier
                     | function_identification ";" function_block
                     | function_heading ";" function_block.

function_heading = "function" identifier
                        [formal_parameter_list]";" result_type.

function_identification = "function" function_identifier.

function_identifier = identifier.

function_block = block.

result_type = simple_type_identifier | pointer_type_identifier.

simple_type_identifier = type_identifier.

pointer_type_identifier = type_identifier.

statement_part = component_statement.

component_statement = "begin" statement_sequence "end".

statement_sequence = statement { ";" statement }.

statement = [ label ":" ]
      ( simple_statement | structured_statement ).

simple_statement = empty_statement | assignment_statement
                 | procedure_statement  | goto_statement.

empty_statement = ";".

assignment_statement = ( variable_access | function_identifier )
                       ":=" expression.

variable_access = entire_variable | component_variable
                | identified_variable | buffer_variable.

entire_variable = variable_identifier.

variable_identifier = identifier.
```

```
component_variable = indexed_variable | field_designator.

indexed_variable = array_variable "[" index_expression
            { "," index_expression } "]".

array_variable = variable_access.

index_expression = expression.

expression = simple_expression
            [relational_operator simple_expression].

simple_expression = [sign] term {adding_operator term}.

term = factor {multiplying_operator factor}.

factor = variable_access | bound_identifier | unsigned_constant
|  function_designator | set_constructor | "(" expression ")"
| "not" factor.

bound_identifier = identifier.

unsigned_constant = unsigned_number | character_string
                  | constant_identifier | "nil".

function_designator = function_identifier
                      [ actual_parameter_list ].

actual_parameter_list = "(" actual_parameter
                        { "," actual_parameter } ")".

actual_parameter = expression | variable_access
                 | procedure_identifier | function_identifier.

set_constructor = "[" [ member_designator { ","
                  member_designator } ] "]".

member_designator = expression [ ".." expression ].

field_designator = record_variable "." field_specifier
                 | field_designator_identifier.

record_variable = variable_access.

field_specifier = field_identifier.

field_identifier = identifier.

field_designator_identifier = identifier.

identified_variable = pointer_variable "^".

pointer_variable = variable_access.

buffer_variable = file_variable "^".

file_variable = variable_access.

procedure_statement = procedure_identifier
( [ actual_parameter_list ] | read_parameter_list
```

```
| readln_parameter_list | write_parameter_list
| writeln_parameter_list ).

read_parameter_list = "(" [ file_variable "," ] variable_access
                      {"," variable_access } ")".

readln_parameter_list = [ "(" ( file_variable | variable_access )
{"," variable_access} ")" ].

write_parameter_list = "(" [ file_variable "," ] write_parameter
                       { "," write_parameter } ")".

write_parameter = expression [ ":" expression
                  [ ":" expression ] ].

writeln_parameter_list = [ "(" ( file_variable | write_parameter )
                          { "," write_parameter } ")" ].

goto_statement = "goto" label.

structured_statement = component_statement | conditional_statement
                     | repetitive_statement | with_statement.

conditional_statement = if_statement | case_statement.

if_statement = "if" boolean_expression "then" statement
               [ else_part ].

boolean_expression = expression.

else_part = "else" statement.

case_statement = "case" case_index "of" case_list_element
                 { ";" case_list_element } [";" ] "end".

case_index = expression.

case_list_element = case_constant_list ":" statement.

repetitive_statement = repeat_statement | while_statement
                     | for_statement.

repeat_statement = "repeat" statement_sequence
                   "until" boolean_expression.

while_statement = "while" boolean_expression "do" statement.

for_statement = "for" control_variable ":=" initial_value
( "to" | "downto" ) final_value "do" statement.

control_variable = entire_variable.

initial_value = expression.

final_value = expression.

with_statement = "with" record_variable_list "do" statement.

record_variable_list = record_variable { "," record_variable }.
```

/**** file: Pascal test progam "pascal.progl" ****/

```
PROGRAM traffic(input,output);
CONST
    terminator          = 0;
    timingsignal        = 1;
    vehiclesignal = 2;
VAR
    vehicles,errors,seconds,starttime,longest,signal: INTEGER;

BEGIN (*  prepare to process signals *)
    vehicles := 0;  errors := 0;  seconds := 0;
    READ(signal);
    WHILE signal <> terminator DO
     BEGIN
    { process a signal or interval and read the next signal  }
        IF signal = vehiclesignal THEN
            BEGIN (*  process a vehicle signal and
                                        read the next signal *)
                vehicles := vehicles + 1;
                READ(signal)
            END
        ELSE
        IF signal > vehiclesignal THEN
        BEGIN  (* process an error signal and
                                  read the next signal }
            errors := errors + 1;
            READ(signal)
        END
        ELSE
        BEGIN  { time a vehicle-free interval and
                                        read the next signal *)
         (* note the start of an interval *)
            starttime := seconds;
            WHILE signal = timingsignal DO
            BEGIN (* process a timing signal
                     and read the next signal *)
                seconds := seconds + 1;
                READ(siganl)
            END;
         (* note the end of an interval *)
         IF seconds - starttime > longest THEN
            longest := seconds-starttime
         END
     END;
     (* output the results *)
     WRITE('  No. of         No.of        elapsed        longest');
     WRITELN;
     WRITE('  vehicles       errors        time               gap');
     WRITELN;
     WRITE(vehicles:12, errors:12, seconds:12, longest:12);
     WRITELN
  END  (* traffic *).
```

/**** file: Pascal test progam "pascal.prog2" ****/

```
PROGRAM weeklypay(oldpayroll,newpayroll,output);

CONST
    taxrate = 0.33;

TYPE
    grades = (manual,skilled,clerical,managerial);
    money = integer;
    employeerecords =
         RECORD
              name : PACKED ARRAY[1..16] OF CHAR;
              number: 0..9999;
              grade: grades;
              payrate: money;
              service: 0..MAXINT
         END;

VAR
    oldpayroll, newpayroll: FILE OF employeerecords;
    oneemployee: employeerecords;

PROCEDURE process (VAR employee: employeerecords);
VAR
    nettpay, taxdeduction: money;

BEGIN
    WITH employee DO
         IF grade <> managerial THEN
         BEGIN
              taxdeduction := taxrate * (payrate-taxallowance);
              nettpay := payrate-taxdeduction;
              writeln(name,number:8,payrate/100:12:2,
                        taxdeduction/100:12:2,nettpay/100:12:2);
              service := service+1
         END
END (* process *);

BEGIN (* weeklypay *)

    RESET(oldpayroll);
    REWRITE(newpayroll);
    WRITELN('name':16,'number':8,'grosspay':12,
            'tax':12,'nettpay':12);
    WRITELN;
    WHILE NOT EOF(oldpayroll) DO
    BEGIN
         READ(oldpayroll,oneemployee);
         process(oneemployee);
         WRITE(newpayroll,oneemplyee)
    END
END (* WEEKLYPAY *).
```

Appendix F

This appendix contains the syntax of the language accepted by CORGI, given in the form required for processing by CORGI.

```
                    /****   file = corgi.spec   ****/

STARTCOMMENT         "/*"   NESTED.
ENDCOMMENT           "*/".
LEXEME               Identifier Literal.
KEYWORD_CASE         KEY_SIG.

%%
InputSpec    = Annotation "%%" Grammar [ "%%" Routines ].
Annotation   = [ Comments ] [ Key-case] Lexemes [ Operators] .
Comments     = "STARTCOMMENT" S-sym { "|" S-sym} [ "NESTED" ] "."
               "ENDCOMMENT" E-sym { E-sym } { "|" E-sym{E-sym}} ".".
S-sym        = Literal.
E-sym        = S-sym   |   "NEWLINE".
Key-case     = "KEYWORD-CASE" ( "CASE-SIG" | "CASE-NONSIG"
               | "CASE-NOTMIXED"  )   "."
Lexemes      = "LEXEME" ( Identifier | Literal )
                     { ( Identifier | Literal ) } ".".
Operators    = "OPERATORS"  ProdS  ".".
ProdS        = Production ProdS | Production.
Production   = Identifier "=" PrecedS ".".
PrecedS      = Preced  PrecedS | Preced.
Preced       = [ Assoc ]  "{" Op-sym { Op-sym  } "}".
Assoc        = "\L" | "\R" | "\N".
Op-sym       = Literal | Identifier.
Anychar      = any_PR_char. /* any_PR_char is a predefined token */
Grammar      = Rule { Rule } ".".
Rule         = Identifier "=" Expression".".
Expression   = Term { "|" Term  }.
Term         = Factor {Factor } [ Action ].
Factor       = ["#"] ( Identifier |  Literal )
               |"(" Expression ")" | "[" Expression "]"
```

```
             | "{" Expression "}".
Literal      = """"  Anychar { Anychar } """"    ==> evaluate_Pstr.
Action       = "==>" Identifier [ "(" ParamS ")" ].
ParamS       = Param { "," Param }.
Param        = Flag | Literal.
Flag         = "0" | "1".
Identifier = uplow_letter { uplow_letter | digit | "_" }
                                ==> evaluate_idr(1).
Routines     = { Anychar }.
```