MKPARSE – AN INTERACTIVE PARSER GENERATOR

by

QING ZHAO
B.Sc., Fudan University, China, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. R. Nigel Horspool

[redacted]

Dr. Michael R. Levy

[redacted]

Dr. Fayez H. El Guibaly

[redacted]

Dr. R. Lynn Kirlin

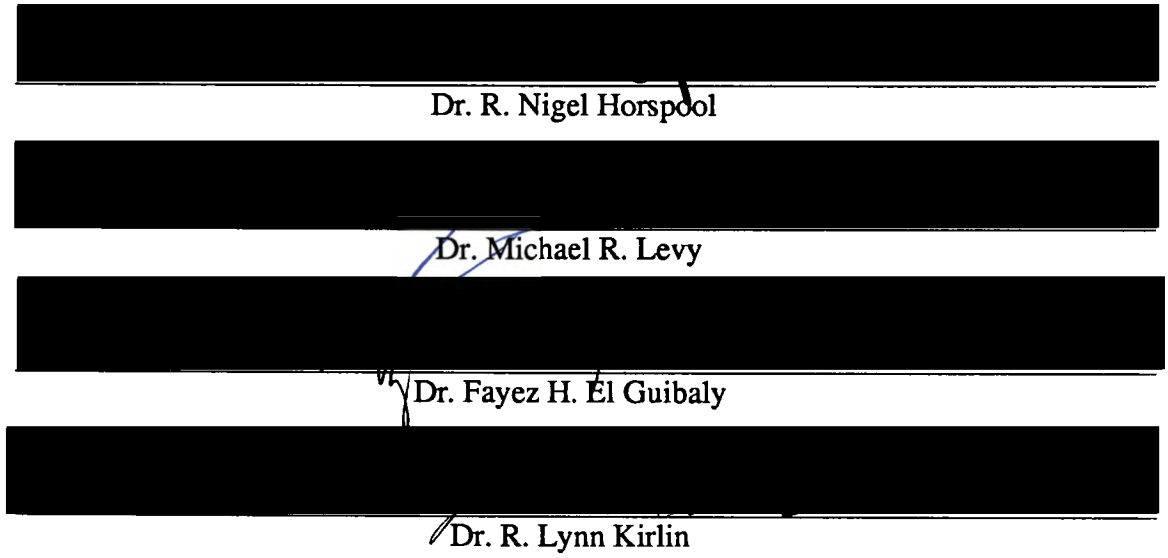
University of Victoria
July 1987

Supervisor: Professor R. N. S. Horspool

ABSTRACT

The design and implementation of an interactive parser generator, Mkparse, are presented in this thesis. Mkparse is the second tool in the Mk* series of tools for compiler construction. Mkparse encourages the user to design a language starting from its abstract syntax instead of from its concrete syntax. Furthermore, abstract syntax trees are output by parsers generated by Mkparse instead of parse trees. Unlike conventional batch-oriented tools, Mkparse is interactive and uses a full-screen interface. Much consideration has been put on the ease-of-use and on providing a user-friendly environment.

Examiners:



Dr. R. Nigel Horspool

Dr. Michael R. Levy

Dr. Fayez H. El Guibaly

Dr. R. Lynn Kirlin

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, Dr. Nigel Horspool, for his invaluable guidance of my research and his numerous suggestions and proof-reading during the writing of my thesis.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Compilers provide a basic interface between a programmer and the machine for which he/she is developing software. Each programming language on each machine needs a compiler to translate source programs from this language into the equivalent machine language programs. A parser forms one phase of a compiler and checks the syntactic correctness of a program, simultaneously performing related tasks such as building a parse tree or an abstract syntax tree and reporting syntax errors. The use of parsers is not restricted only to compilers. They can also be used to perform syntactic checking for some more general tasks. Parsing a command language and parsing a subset of English are common examples of applications of parsing techniques. Compilers are complicated software products and manual implementation of compilers can be very tedious work. As computer science has progressed, automated tools to assist in compiler construction have been developed. For instance, Lex [L1], Yacc [J1] and S/SL [H2] are all tools for compiler construction. With the help of those tools, the effort of writing compilers can be reduced and programmer productivity can be considerably increased. But, as will be explained in this thesis, most of those conventional tools are batch-oriented and tend not to be very user friendly.

A software environment provides automated support for a set of related tasks associated with software development, and helps integrate tools and techniques into a coherent software development methodology [W2]. It helps a programmer, in relative isolation from others, turn a specification into a working program. Reducing the duration of program development, improving the quality of software, easing its use, and leaving most of the tedious work to the computer, are all concerns of a software environment. Most current environments are designed to be full-screen interactive, menu-driven and user friendly.

Mk* is a project to develop a series of tools that assist in a compiler development effort. It is being developed at the University of Victoria. The tools are screen-oriented and use, wherever appropriate, a simple menu-driven interface. Each tool allows the user to create and to maintain one phase of a compiler. The first tool [H4] permits the user to create a lexical analyzer for a language. The second tool, **Mkparse**, creates the matching syntactic analyzer. Later tools in the series address the problems of type checking, semantic analysis and code generation.

Mkparse is an interactive parser generator for editing and generating parsers. It provides the user with an interactive, menu-driven, and easy to use interface for editing and creating parsers. The input of Mkparse is a definition of some language. The output of Mkparse is a parser for that language. The parser can be re-read by Mkparse to allow the definition of the language to be changed. Thus, Mkparse can be used

throughout the lifetime of its products. Mkparse encourages the user to think of the structure of a programming language in terms of its abstract syntax when he/she defines the syntax of the language. An abstract syntax tree (AST) is a representation of the abstract syntax of a program. The user specifies a language by defining its abstract syntax in terms of subtree generation rules. The user also supplies mappings from each AST structure to a corresponding concrete structure. The collection of all AST generation rules defines the abstract syntax of a language, while the collection of concrete structures is equivalent to a BNF grammar. The result of using the Mkparse tool is a parser that can automatically create an AST from its input.

Besides the main function of being a parser generator, Mkparse also provides ease of use and user friendliness. By using a simple, interactive and menu-driven user interface and by providing instant feedback to erroneous or conflicting specifications, the effort needed to create a new parser for a small language like Pascal should be reducible to under an hour. On-line information helps the user to understand the functions of each menu option and formats for the data entries. Mkparse allows the user to concentrate on the overall design of a language and not to be bothered by unimportant details. As much checking as possible is performed interactively.

To avoid repetition, Mkparse also permits the user to obtain information from the scanner file generated by Mkscan, the first tool in the Mk* series. Following normal principles of software engineering [M1], Mkparse has a modular structure. This

structure simplifies software maintenance of Mkparse itself.

Chapter 2 provides some background knowledge about compilers, parsing theory, software tools and environments. In Chapter 3, the design goals of Mkparse are presented from the point of view of the language structure and software development environments and tools. Module structure, menu organization, functions of each menu, and formats for defining abstract and concrete structures in Mkparse are described in Chapter 4. Simple examples to explain how Mkparse accepts the definition of a language are given in this chapter. Experience with Mkparse for a simple arithmetic expressions grammar, a Pascal grammar, and a grammar for a subset of the C language are discussed in Chapter 5. Finally, in Chapter 6, conclusions, future work and possible improvements to Mkparse are discussed. Appendix A includes a complete example of using Mkparse for a simple expression grammar. Appendix B shows a Pascal grammar mentioned in Chapter 5. Appendix C gives a grammar for a subset of the C language.

CHAPTER 2

BACKGROUND

2.1. Typical Compiler Structure

A compiler is a program, written in an implementation language, whose input is a program in a source language and which produces a semantically equivalent program in a target language. The source language may be a programming language such as C or Pascal. The target language is normally the machine language or the assembly language of some computer. In some cases, the target language may be another programming language. A source language is usually human-oriented whereas the target language is machine-oriented. Typically, a compiler consists of the following phases: a lexical analyzer (scanner), a syntax analyzer (parser), a semantic analyzer, an intermediate code generator, a code optimizer and a code generator. These phases, together with the symbol table manager and the error handler, constitute a compiler. Figure 2.1 shows a typical compiler structure [A2].

The scanner is the first phase of a compiler. Its job is to read input characters from the source program, recognize strings of characters defined as symbols called tokens, strip out comments and white space from the source program and output a

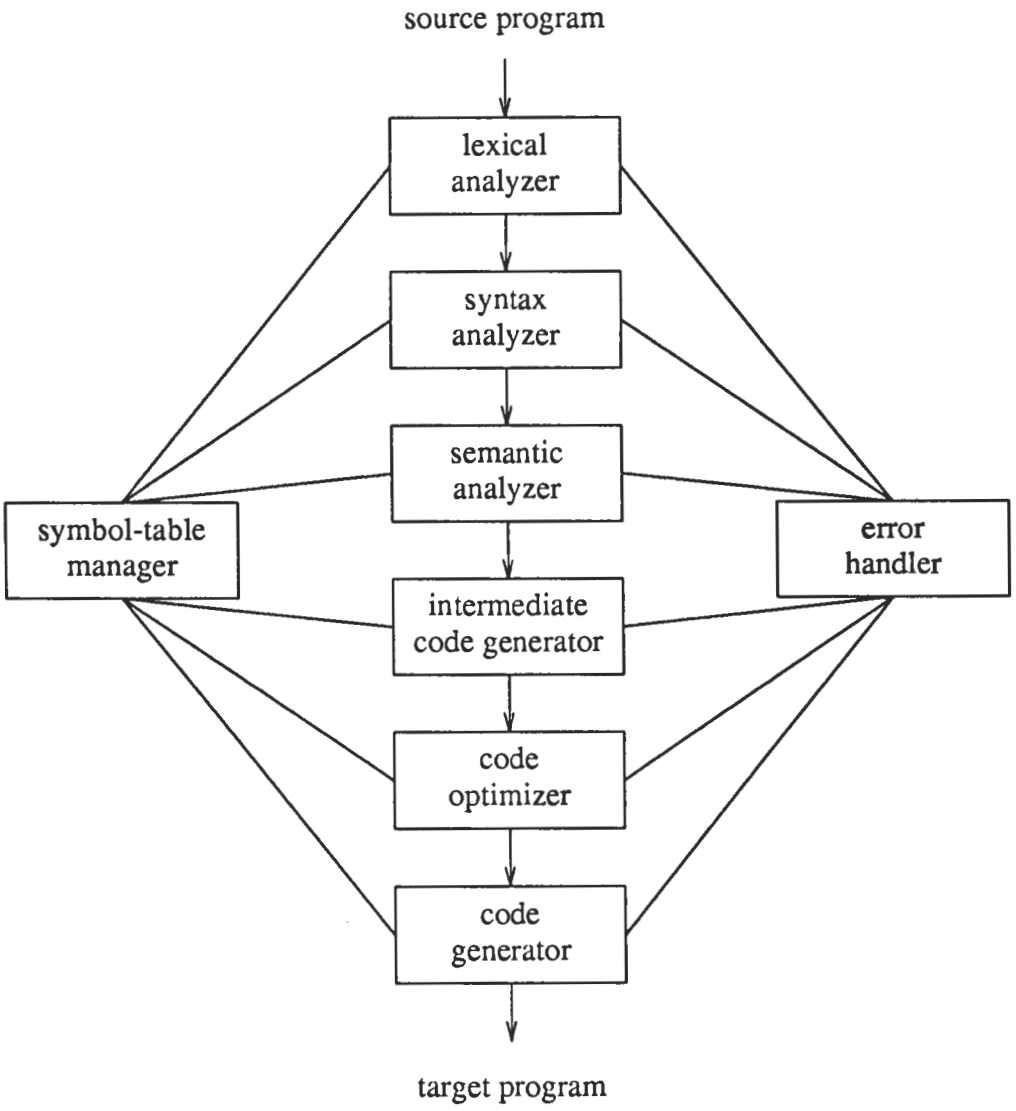


Figure 2.1: Typical compiler construction

sequence of tokens to the parser.

The parser performs syntax analysis of programs according to some grammar. It accepts as its input a sequence of tokens produced by the scanner and checks the sequence of tokens for syntactic correctness according to the grammar of the language. As part of the checking, it groups the tokens into grammatical phrases that are used by the compiler to synthesize output. It reports any errors and tries to recover from these errors by using error recovery facilities. Finally, if there are no errors or if all errors have been corrected by the error recovery procedures, it creates a representation of a parse tree or an abstract syntax tree, which represents the grammatical structure of the source program, along with additional information for use in other parts of the compilers.

The semantic analyzer determines those properties of a program that are classed as static semantics and verifies corresponding context conditions. Name checks, type checks, flow-of-control checks and uniqueness checks are performed in the semantic analysis phase.

The intermediate code generator accepts all the information produced by previous phases and outputs machine-independent intermediate code. One typical format is three address code which can be implemented by quadruples, triples or indirect triples. Another format is stack-oriented or, equivalently, tree-structured, which is more suitable for use when the intermediate code can be retained in main memory.

The scanner, parser, semantic analyzer and most of the intermediate code generator are independent of the target language and therefore these phases are sometimes grouped together to form the front end of the compiler.

The task of the code optimization phase is to improve the intermediate code so that it can be translated into faster or more compact machine code.

As indicated by its name, the final phase, the code generator, produces code of the target language, usually machine language or assembly language. Memory locations are selected for each of the variables used by the program. Then intermediate instructions are each translated into a sequence of machine instructions that perform the actions specified in the source program.

2.2. BNF Notation

A language can be defined by describing what programs in this language look like (the syntax of the language) and what its programs mean (the semantics of the language). The syntax of a language is normally defined by a grammar which consists of a set of rules that determines if a sentence is well-formed or not.

A BNF "grammar" is a formal notation for specifying a potentially infinite "language" (set of strings) in a finite way. Strings in the language are generated by starting with a string consisting of a distinguished "start" symbol and successively

rewriting the string according to a finite set of rules or ''productions''. Figure 2.2 gives an example of a grammar for a small subset of English [H3].

The sample grammar defines a subset of English sentences. An English *sentence* might be defined as a *subject phrase* followed by a *verb phrase*. The *subject phrase* might consist of the word 'the' followed by a *noun*. A *verb phrase* could be defined as a *verb* followed by an *object phrase*. The *object phrase* might consist of an *article* followed by a *noun*. The italics have been used here to denote syntactic elements that have been described in the sentence; they correspond to the grammar symbols enclosed by ''<'' and ''>'' characters and are called **nonterminal symbols**. A subset of the complete set of English sentences could be defined by allowing only the nouns 'man' and 'dog', the verb 'has' and the articles 'a' and 'the'. The words in inverted commas are English words which will actually appear in the English sentence produced and are

1 <sentence>	→	<subject_phrase> <verb_phrase>
2 <subject_phrase>	→	the <noun>
3 <verb_phrase>	→	<verb> <object_phrase>
4 <object_phrase>	→	<article> <noun>
5 <verb>	→	has
6 <article>	→	a
7 <article>	→	the
8 <noun>	→	man
9 <noun>	→	dog

Figure 2.2: An example grammar

therefore called **terminal symbols**. The notation used in the above grammar is called *Backus-Naur Form* (or **BNF**). The rules of this form are often called **productions**. The sentence 'the man has a dog' would be obtained by starting from <sentence> and performing substitutions using the listed productions:

Production used	<sentence>
1	<subject_phrase> <verb_phrase>
2	the <noun> <verb_phrase>
3	the <noun> <verb> <object_phrase>
4	the <noun> <verb> <article> <noun>
9	the <noun> <verb> <article> dog
8	the man <verb> <article> dog
5	the man has <article> dog
6	the man has a dog

2.3. LR Parsers and LR Family

Almost all parsing techniques fall into one of the two categories: top-down parsing and bottom-up parsing. Recursive-descent parsing and LL parsing are examples of top-down parsing techniques, while operator-precedence parsing and LR parsing are bottom-up parsing techniques.

Among these techniques, LR parsing plays an important role and has several advantages over the others. LR(k) is the most general non-backtracking method in common use and is one of the fastest methods. It can be used with a large class of

context-free grammars. The ''L'' stands for left-to-right scanning of input, the ''R'' for constructing a rightmost derivation in reverse, and the ''k'' for the number of input symbols of lookahead that are used in making parsing decisions. When (k) is omitted, k can normally be assumed to be 0 or 1.

An LR parser is usually implemented as a parse table and a driver program. The driver program can be the same for all LR parsers while the contents of the parse table depend on the grammar. A stack is needed to hold information during parsing. See Figure 2.3 [A2].

While parsing an input string of symbols, an LR parser first pushes the start state onto the stack. It then reads an input token and looks up the parse table for the next parse action. The legal parse actions are either a ''shift'' on the current input token or a ''reduce'' by one of the production rules. A shift action occurs when the parser consumes an input symbol by pushing it onto a stack, and the parser transfers to another state. A reduce action occurs when a complete grammatical structure, i.e. the right hand side of a production rule, has been recognized. The parser reduces this part by removing its symbols from the top of the parser stack and replacing them with the left-hand-side, which must be a nonterminal symbol. Immediately after a reduction, the parser transfers to a new state. This is called a ''goto'' action. Finally, if only the start state and the end symbol are on the stack, the parsing terminates successfully.

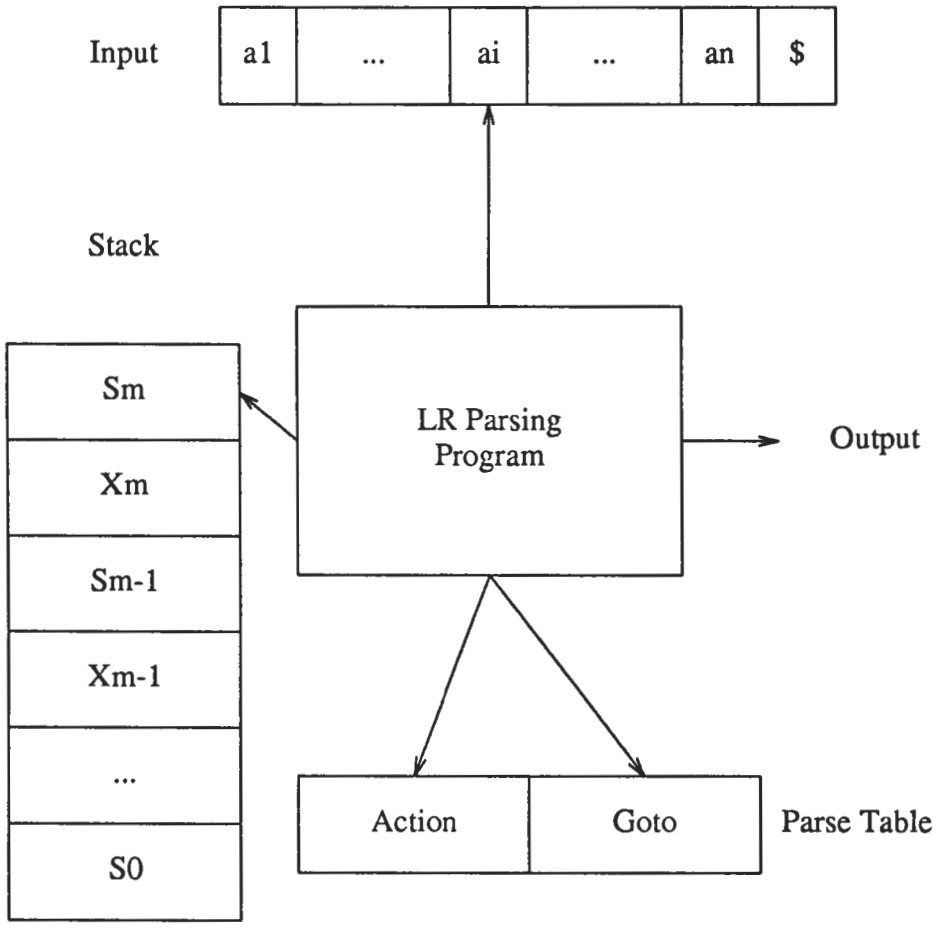


Figure 2.3. A working module of an LR parser

Since LR parsing technique was first proposed by Knuth in 1965, considerable effort has been put into its development. Initially, the LR parsing technique was not widely used because early presentations of the method made the theory seem forbiddingly difficult and direct implementations were very inefficient. The approach

become practical only when a number of optimizations were discovered by Aho [A1], Anderson [A3], DeRemer [D1], Korenjak and others [G2]. Now, LR parsing has become one of the most important and most popular parsing techniques.

A potential drawback of the method is that it is too much work for a human to construct a LR parser by hand for a typical programming language grammar. Fortunately, programs called parser generators exist to build parse tables mechanically.

The LR family of parsing techniques has many members, of which four predominate. They are LR(0), SLR(1) (''simple'' LR), LALR(1) (''lookahead'' LR), and LR(1) listed in order of increasing recognition power. LR(0) is the easiest to implement but the least powerful of the four. An LR(0) parser generator may fail to produce a parse table on certain grammars for which the others may succeed. SLR(1) is a simple extension to the LR(0) method. It reduces by a production rule only when the incoming symbol is in a set called ''follow'' of the left-side nonterminal symbol of the production rule. A follow set of a nonterminal symbol consists of symbols which can appear after that nonterminal in legal sentences of the grammar. The LALR(1) method is similar, except that a full analysis of the legal look-ahead symbols in each state of the LR(0) parser is performed. The LALR(1) table, the SLR(1) table and the LR(0) table all have the same number of LR parse states for the same grammar. The LALR(1) method is suitable for most programming languages and can be implemented efficiently. Though LR(1) is the most powerful method of the four, it almost always

has many more LR states than LR(0), SLR(1), or LALR(1) for the same grammar. Therefore, it normally requires much more computer memory. Figure 2.4 shows the numbers of states and memory sizes of each method for a Pascal grammar containing 162 production rules, 63 terminal symbols, and 53 nonterminal symbols. Some simple table optimizations of combining shift and reduce states wherever possible have been performed in building these parse tables.

While building the LR parse table, two different kinds of conflicts may occur if the grammar is not suitable for the parsing method. A shift-reduce conflict occurs when either a shift action or a reduce action could be chosen as the next parsing action on the same input symbol. A reduce-reduce conflict occurs when two or more rules are applicable as reduce actions on the same incoming grammar symbol.

Parser	Number of LR States	Size of Parse table(bytes)
LR(0)	203	47096
SLR(1)	203	47096
LALR(1)	203	47096
LR(1)	1040	241280

Figure 2.4: Comparison of numbers of LR states

Applications of a parser generator are not necessarily restricted to compiler writing. It could also be used to create parsers for many other syntax checking tasks, for example, in natural language processing.

2.4. Automated Tools For Writing Compilers

Programming used to be considered very tedious work because there were few tools to help the programmers. A compiler itself is a complicated program and was once considered extremely difficult to write. It took 18 staff-years to implement the first FORTRAN compiler in the 1950's [W3]. Since then, computer science has made considerable progress. Programming tools, such as editors, compilers, scanner generators and parser generators, have furnished programmers with a modest amount of automated assistance. Systematic techniques now exist for handling many important tasks that occur during compilation. Good implementation languages, programming environments and software tools have been developed. With these advances, a substantial compiler can be implemented in a relatively short period of time and with much less effort than before. Theoretical developments have permitted the construction of some parts of a compiler to be mechanized, and implementing these parts by automated tools has become standard. Tools like Lex, Yacc and S/SL, have helped with the implementation of hundreds of compilers.

2.4.1. Yacc

Yacc (Yet Another Compiler-compiler) [J1] provides a general tool for recognizing the grammatical structure of an input stream. The Yacc user prepares a specification of the input process including grammar rules which describe the input structure, code for semantic actions which is to be executed when these structures are recognized, and a low-level input routine – a lexical analyzer. Yacc then produces a parser coded in the C language based on a LALR parse table. This parser calls the lexical analyzer to extract tokens from the input stream and groups these tokens according to the grammar rules. While a rule is being recognized, associated code to perform a semantic action may be executed. Such actions have the ability to return values, make use of the values of other actions and call subprograms.

There are three sections in a Yacc specification. The first section is the declaration section in which token names must be declared and, if desired, precedences and associativities of tokens and rules can be declared. In the second section, the rule section, the grammar rules are defined. With each grammar rule, the user may associate actions to be performed each time the rule is used in the parsing process. An action is an arbitrary statement in the C language. It can do input and output, call subroutines, and alter external variables. The format used for a grammar specification is similar to that of BNF. The user must supply a lexical analyzer which reads the input stream and communicates tokens to the parser. The scanner, together with other subroutines such

as tree builder, error handler etc. can be placed in the third section, the program section.

Figure 2.5 is an example of a Yacc specification. This specification defines a syntactic checker for the language defined by the grammar in Figure 2.2.

In the first section, five tokens are defined. Grammar rules are placed in the second section. There is one semantic action shown, which is a function call to 'print_accept_sentence'. In the third section, the lexical analyzer 'yylex' and a function 'print_accept_sentence' are coded. The lexical analyzer can also be produced using the Lex scanner generator, which will be discussed later. A user can place arbitrary program fragments in the last part. The parser produced by Yacc from this specification is able to accept sentences in the language defined by the grammar of Figure 2.2. For example, 'the man has a dog' will be accepted. After a grammatically correct sentence is accepted, the function 'print_accept_sentence' is called to print the sentence.

A grammar is said to be ambiguous grammar if there exists a sentence in the language generated by the grammar such that more than one sequence of grammatical rules can be used to derive this sentence. That is, at least two different parsing trees can be built from the same input. For example, the production rules **E** $\rightarrow$ **E + E** and **E** $\rightarrow$ **E * E** will cause ambiguities. If a grammar contains both rules, the sentence **id + id * id** will have two distinct leftmost derivations:

```
%token          'has' 'a' 'the' ' man' 'dog'

%%
sentence        : subject_phrase verb_phrase {print_accept_sentence();}
                ;
subject_phrase  : 'the' noun
                ;
verb_phrase     : verb object_phrase
                ;
object_phrase   : article noun
                ;
verb            : 'has'
                ;
article         : 'a'
                | 'the'
                ;
noun            : 'man'
                | 'dog'
                ;
%%
        yylex(){
          ......
        }

        print_accept_sentence(){
          ......
        }
```

Figure 2.5: An example of Yacc specification

$$
\begin{array}{lll}
E & \rightarrow E + E & \\
 & \rightarrow \mathbf{id} + E & \\
 & \rightarrow \mathbf{id} + E * E & \\
 & \rightarrow \mathbf{id} + \mathbf{id} * E & \\
 & \rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} &
\end{array}
\qquad
\begin{array}{lll}
E & \rightarrow E * E & \\
 & \rightarrow E + E * E & \\
 & \rightarrow \mathbf{id} + E * E & \\
 & \rightarrow \mathbf{id} + \mathbf{id} * E & \\
 & \rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} &
\end{array}
$$

with the two corresponding parse trees shown in Figure 2.6.

An ambiguous grammar is never acceptable for the LR parsing technique, because one or more parser states will have conflicts. However, Yacc is able to handle some ambiguous grammars if disambiguating information in the form of operator precedences and associativities is also provided. In addition, two default disambiguating rules are used in Yacc to solve the shift-reduce and reduce-reduce conflicts while building the LALR table. For a shift-reduce conflict, the default action is to do the shift. For a reduce-reduce conflict, the default action is to reduce by the grammar rule which appears first in the grammar.

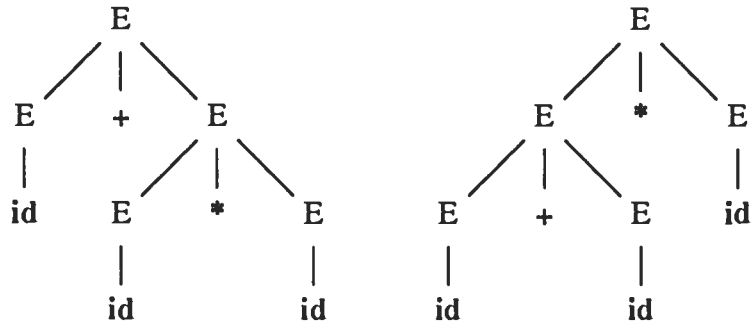


Figure 2.6: Two parse trees for **id + id * id**

The ability of Yacc to accept ambiguous grammars provides users with a way to write more natural and clearer BNF specifications. In addition, ambiguous grammars with appropriate disambiguating rules can be used to create parsers which are faster and smaller than parsers constructed from equivalent unambiguous grammars. Yacc's semantic actions part gives users the freedom to add code to perform specific tasks. As a couple of minor criticisms, the use of LALR parse tables in the parsers produced by Yacc is not sufficiently powerful to handle LR(1) grammars. It also seems unnecessary to define all the tokens first in Yacc.

2.4.2. Lex

Lex [L1] is a program generator designed to assist with lexical processing of character input streams. It accepts a source specification in the form of regular expressions for character string matching, and produces a program in the C language. A lexical analyzer produced by Lex partitions the input stream into strings that match the regular expressions. At the boundaries between strings, program actions provided by the user can be executed. The Lex source file allows each regular expressions to be associated with a program fragment. As each expression appears in the input to the lexical analyzer, the corresponding code fragment is executed. Lex accepts ambiguous specifications and chooses the longest match possible at each input point. In a similar manner to Yacc, Lex source input has three parts: definitions, rules and user subroutines. The following is an example of a Lex specification, which produces a scanner to

accept tokens defined by the grammar in Figure 2.2. and identifiers which begin with a letter followed by zero or more letters and digits.

```
A   [A-Z]
a   [a-z]
D   [0-9]
%%
" " |
"\t" |
"\n" ;
has                 return("has");
a                   return("a");
the                 return("the");
man                 return("man");
dog                 return("dog");
{A} ({A}{a}{D})*    fprint("unknown word \n");
%%
```

Figure 2.7: An example Lex specification

The scanner defined by this Lex source specification accepts five tokens as well as skipping over spaces and printing out information if a string which is not in the language is encountered. "%%" on a line by itself acts as a separator between Lex input sections.

2.4.3. S/SL

In Holt, Cordy and Wortman's paper [H2], they reported the implementation of a language called S/SL which is specifically designed for writing compilers. S/SL stands

for syntax and semantic language. This language can be divided into two parts, a subset called SL and the semantic mechanisms. SL is able to describe the first two phases of a compiler (scanner and parser) and has the same recognition power as a LR(k) parser. SL plus the semantic mechanisms in which semantic operations are carried out by a base language such as Pascal becomes the whole S/SL. It can be used to implement a whole compiler.

S/SL is a very simple language compared with other common programming languages. It has the following features. It contains sequencing, repetition, and selection of actions. It has input and output of tokens; output of error signals; subroutine calls; and finally execution of semantic operations defined in some other standard programming language. S/SL is a language without data or assignments and is a pure control language. Data can be manipulated only via the semantic operations hidden in the other language. Programming in S/SL implies an important programming methodology. This methodology breaks the problem into two parts. One is the abstract algorithm which can be implemented in S/SL itself and the other is the abstract data written in a base language such as Pascal. The abstract data is further divided into largely independent semantic mechanisms. Each semantic mechanism is an abstract machine that can carry out a well-defined set of instructions (its semantic operations). An S/SL program has an input stream of tokens and two output streams of output tokens and error tokens. It translates the input stream into an output stream and manipulates data by invoking semantic mechanisms. The interface to each semantic mechanism is

defined in S/SL but the implementation is hidden in a base language. This language has successfully been used to implement several compilers, including the EUCLID compiler at the University of Toronto.

```
Statement:
 [
  |identifier:
    @AssignmentOrCallStatement
  |'if':
    @IfStatement
  |'case':
    @CaseStatement
  |'while':
    @WhileStatement
  |'repeat':
    @RepeatStatement
  |'for':
    @ForStatement
  |'with':
    @WithStatement
  |'begin':
    @BeginStatement
  |'goto':
    @GotoStatement
  |*:
    % null statement
 ];
```

The above is an S/SL source specification for recognizing a Pascal statement. S/SL statements beginning with '@' are subroutine calls; '%' indicates a comment; '|' separates options; '*' denotes the default option. '[' and ']' are quote symbols to bracket the subroutine 'Statement'. Tokens are enclosed in single quote characters. Each kind of the statements, such as the **if-statement**, is handled by its own rule. The

following S/SL specification fragment describes the **if-statement**:

```
IfStatement:  % Just accepted 'if' token
 @Expression 'then' @Statement
 [
  |'else':
    @Statement
  |*:
 ];
```

The implementation of S/SL is interesting. First, an S/SL machine which contains thirteen machine-like instructions is defined. Each S/SL action can be represented using these thirteen instructions. Then, a program called the S/SL processor is used to map an S/SL source program to those instructions representing by a sequence of numbers called a table. Finally, a table ''walker'' walks through the table executing instructions by invoking corresponding procedures and thus carrying out the actions of S/SL program. This implementation is simple and efficient.

As opposed to the BNF notation of Yacc, a syntax graph definition of a language is the normal representation for writing an S/SL parser. The S/SL parser is translated into a recursive descent parser [A2].

2.5. User Interactive Environment

The compiler writing tools mentioned above are all batch-oriented tools. Like writing a conventional program, first the user needs to edit the source file, then feeds

the source file to those tools and gets the result. If there are any errors in the source file, the user has to go over the procedure again to modify the source specifications and so on. These tools are not always convenient and efficient to use, especially to people who are not experts in these fields. Better tools and better environments for developing software are important.

Software development environments have the general goals of trying to provide their users with a good environment, with a good set of tools, to reduce the software development period, to improve productivity and to help produce high-quality software. A good software development environment is crucial to programmers. Interactive tools have special advantages over batch-oriented tools. In using a batch tool, the user must have a detailed knowledge of the details of format, rules and special definitions of the source file. It takes time and is also difficult for beginners or infrequent users to use it. But using a tool with a good interactive interface, on the other hand, will be easier for a beginner since he/she can follow the paths provided by the system using prompts, menus, questions, answers and commands. There is also usually on-line help information which can be checked instantly. But the biggest advantage is that the user usually can obtain feedback immediately after he/she has entered the data, typed a command or selected a menu option. A user can learn quickly and easily while using an interactive system. Interactive tools such as spreadsheets and wordprocessors have shown the advantages and are used by many people. Generally speaking, interactive tools are easier to use, more user-friendly and more productive than batch-oriented

tools.

2.6. MK* -- A Series Of Tools For Developing Compilers

Mkparse is one of a series of tools to assist in a compiler development effort [H4]. Each tool allows the user to create and to maintain one phase of a compiler. The tool series is envisaged as follows.

1. Mkscan – a scanner generator
2. Mkparse – a parser generator
3. Mksymbol – a symbol-table/type-checker generator
4. Mksemantic – a semantic checker generator
5. Mkcodegen – a code generator generator

Each generator creates one or more program files which may be separately compiled and then linked to produce a complete compiler. The tools cooperate with each other to avoid the need for the user to ever re-input information that has already been provided to one of the other generator tools. The file created by Mkscan contains the source code for a complete lexical analyzer. The file is self-describing in that it can be re-input by Mkscan for the purpose of making modifications. Mkparse is able to read the scanner file to obtain token numbers defined in Mkscan. The parser file output by Mkparse will be one of the input files to its successor, Mksymbol (See Figure 2.8).

One general design goal has been to make the tools easy to use by novices. So, these tools are all screen-oriented and use a simple menu-driven interface. By using a

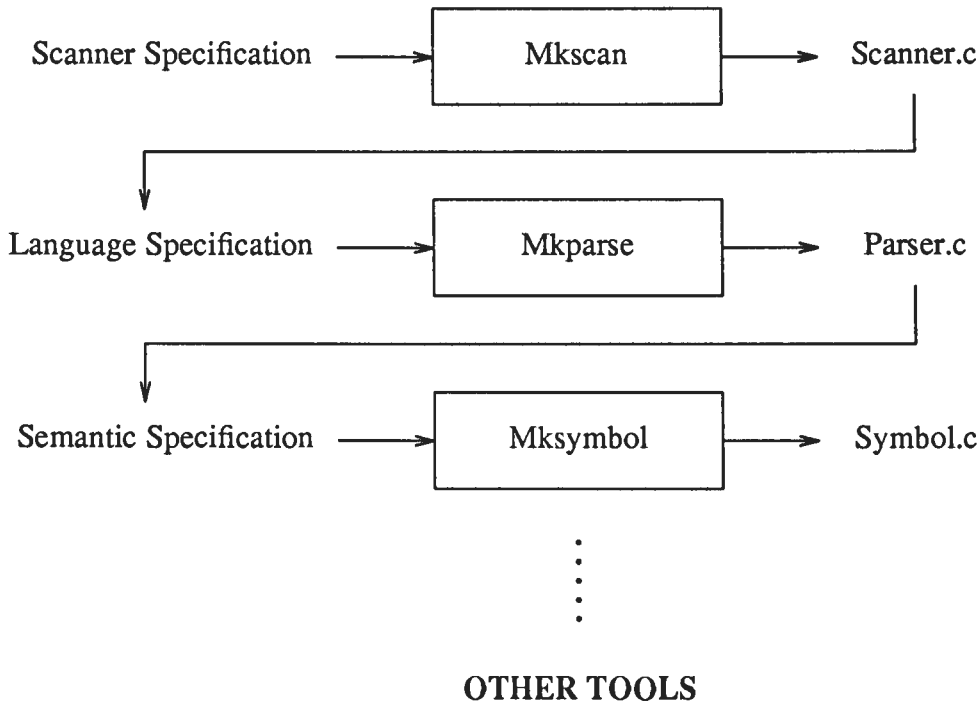


Figure 2.8: Mkparse in Mk*

simple, consistent user-interface and by providing instant feedback to erroneous or conflicting specifications, the effort needed to create a new compiler for a small language like Pascal could be reduced to just a few days.

CHAPTER 3

THE DESIGN GOALS OF MKPARSE

3.1. Abstract Structures Versus Concrete Structures

The abstract syntax of a programming language specifies the compositional structure of a program. It can be a useful place to start when designing or learning a language, or understanding the meaning of a program. An abstract syntax tree (AST) is a condensed and rearranged version of a parse tree. Most superfluous structure is discarded in an abstract syntax tree, leaving a more convenient computational object. Abstract syntax trees characterize the abstract syntax quite naturally and can be created during the parsing process. Figures 3.1a and 3.1b show the parse tree and an abstract syntax tree representation of the expression **x * x + (y - z)** for the following grammar:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & (E) \mid \text{id}
\end{array}
$$

The parse tree in Fig 3.1a contains long chains of reductions, such as y $\rightarrow$ id $\rightarrow$ F $\rightarrow$ T $\rightarrow$ E. However, in the corresponding abstract tree in Fig 3.1b, that long series of reductions becomes a short one y $\rightarrow$ id $\rightarrow$ E. The superfluous part has been removed,

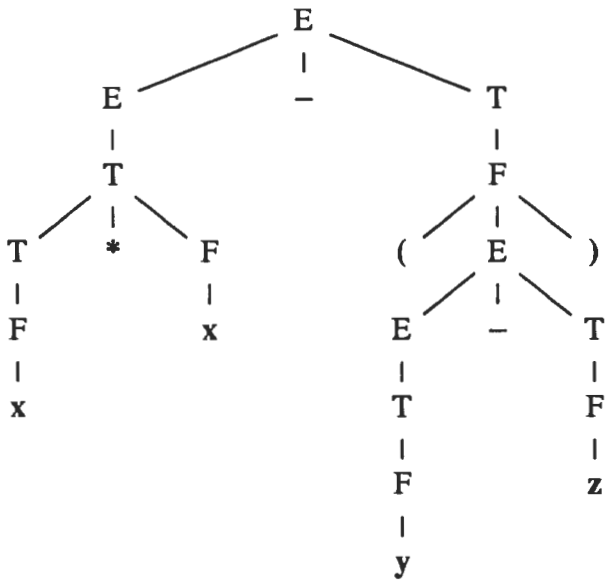


Figure 3.1a. A parse tree for **x * x - (y - z)**

while the semantic meaning remains the same. These superfluous parts are caused by production rules which are needed to define precedences and associativities of the operators. For example, in order to define that "*" has a higher precedence than "+" and that both associate to the left, productions $E \rightarrow E + T$, $E \rightarrow T$, $T \rightarrow T * F$, $T \rightarrow F$, and $F \rightarrow$ id are needed instead of just $E \rightarrow E + E$, $E \rightarrow E * E$, and $E \rightarrow$ id. Because the goal of the compiler's analysis task is to determine the meaning of the source program, complications such as operator precedence and certain keywords can be forgotten once the AST has been constructed.

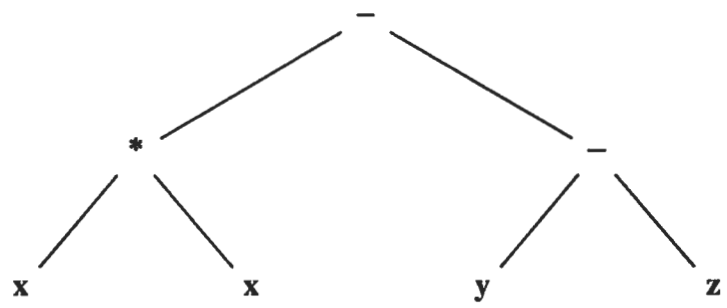


Figure 3.1b. An abstract syntax tree for **x * x - (y - z)**

Because abstract syntax captures the compositional structure and semantic meaning of programs and is simpler than its concrete counterpart, Mkparse encourages users to think in terms of the abstract structure when designing or specifying a language. The main semantic characteristics of a programming language are reflected by its abstract syntax tree definitions where each tree node represents a language construct with a well-defined semantic meaning. Mkparse has been designed to generate LR parsers with associated semantic actions that construct abstract syntax trees. Mkparse accepts both abstract definitions and corresponding concrete definitions of languages. The concrete definitions are used for building LR parser tables while the abstract definitions are needed to define semantic actions for building an AST. Because each node in an AST represents a semantic concept of the language, we will call such nodes *concepts*. In Mkparse, each concept has one or more definitions and each definition consists of two

parts. One part, called the *tree rule*, is its abstract definition while the other part is the corresponding concrete definition. Figure 3.2. is an example of one definition for the concept **statement**. It defines the **if statement** as one form of a **statement**. A tree rule consists of a two-level subtree of an AST. The first level is a node that represents the concept to be defined. The second level contains zero or more concepts. The concrete definition part of a concept consists of one or more production rules in BNF notation. Because a concept will often correspond directly to a nonterminal symbol in the BNF grammar, we will use the same identifier or symbol for both the concept and its corresponding nonterminal symbol.

There is one-to-one correspondence between a concept in a tree rule and a nonterminal symbol in its concrete definition part. There is an implicit assumption that the left-hand-side symbol in a BNF production rule (concrete definition) corresponds to the current concept being defined. Mkparse also accepts some convenient notations. The abstract definition

node ...

represents the list of zero or more concepts **node**,

node node ..... node.

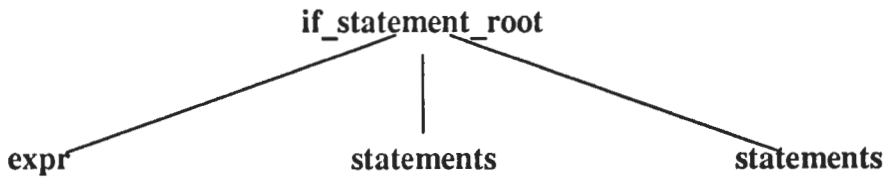
The corresponding concrete definition is

begin_symbols LIST0 node SEP a-term ENDLIST end_symbols

representing the list

begin_symbols node a-term node a-term ... node end_symbols.

Abstract Syntax:



Concrete Syntax:

if **expr** then **statements** else **statements**

Figure 3.2. An example of a definition for concept **statement.**

node is used as a concept and a nonterminal. **a-term** is used as a terminal symbol. **LIST0, SEP, ENDLIST,** and **...** are meta-symbols in Mkparse denoting a list, a separator, the end of a list, and the repetition of a concept. **begin_symbols** and **end_symbols** are two lists of terminal symbols, each of which contains zero or more terminals. Using the convenient notation describe above, the grammar structure

BEGIN statement ; statement ; ... statement END

can be represented by the tree rule (abstract syntax)

statement ...

and the BNF notation (concrete syntax)

BEGIN LIST0 statement SEP ; ENDLIST END

If a concept has no child in one of its abstract definitions, this concept could appear on the lowest level of the entire abstract syntax of the language and one of its corresponding concrete definitions will contain only terminal symbols.

3.2. A Menu-oriented Software Tool

It is anticipated that most users of Mkparse will be compiler writers. Some casual users who have only little knowledge of parser construction, who simply need parsers for their applications as soon as possible, are also expected. In order to serve novices, experts and occasional users, ease of use and user-friendliness have been chosen as design goals of Mkparse. The following points have been taken into consideration during the design of Mkparse [G1].

- Adaptability: a system must be adaptable to the physical, emotional, intellectual and knowledge traits of the people it serves.

- Transparency: a system must permit one's attention to be focussed entirely on the task or job being performed, without concern for the mechanics of the interface and other unimportant details.

- Naturalness: follow the way of the user's natural thinking.

- Predictability: every system action should be expected within the context of other actions that are performed.

- Forgiveness: a system should be tolerant of the human tendency to make errors.

- Efficiency: eye and hand movements must not be wasted. Attention should be paid to relevant controls and displays of information.

To achieve the design goals, Mkparse has been designed to provide an interactive environment to its users. According to John C. Thomas [T1], concern and consideration for the human interface and human factor must be given as much attention as to that placed on the accuracy of the tool's functions. While designing this interactive tool to provide better service and to improve productivity, the human-factor, which is the relationship between humans and computers and the reactions of the human to a computer interface, has also been taken into consideration. Human considerations in screen design represent the needs and requirements of users and are oriented towards clarity, meaningfulness, and ease of use. A good screen design is important in an interactive environment. The most desirable features of a screen layout are: an orderly, clear, and clutter-free appearance; an obvious indication of what is being shown and

what should be done with it; expected information where it should be; a clear indication of what relates to what; plain, simple English and a simple way to find out what is in the system and simple way to get it out. But in implementing the screen displays, the physical constraints imposed by typical terminals must also be considered. An assumption that the terminal is at least as powerful as a VT100 is made. To maximize software portability and ease the burden of programming the **curses** and **termlib/termcap** [A4] packages are used. These are standard packages on UNIX systems.

Menu screens are used extensively in Mkparse. Menus can tell users clearly what the system can do and what the system provides. A series of screens leads a person from general descriptors on the first screen through increasingly specific categories on following screens until the lowest level screens and the desired choices are reached. Menu screens are effective because they utilize the more powerful human capability of recognition rather than the weaker capability of recall. Working with menus reminds people of available options and information that they may not be aware of or have forgotten.

According to Galitz [G1], the information provided on one screen should be standardized. The number of options for one menu should not be less than three or more than eighteen. For short lists of options (seven or fewer), sorting the options by order of use or frequency of use is desirable. But for longer lists (eight or more

options) or short lists with no obvious frequency ordering or patterns of use, alphabetic order is more helpful. These design principles have been applied to the design of Mkparse menu screens. The menus are organized as a tree structure and users can traverse this tree freely by using functional options to go down one level and a quit option to go back up one level.

On-line help information, error diagnostics, and warning messages play a critical role in software systems. Mkparse tries to provide sufficient explanatory messages so that printed manuals are unnecessary. The on-line material can focus on the user's current task; it is very easy to locate; and it can be retrieved immediately. Associated with each menu screen and data entry screen, there is some on-line help information. General information about the current screen is given first. Then more detailed explanation of the meaning of each option, the work performed by each option and the relations among the options is displayed. Examples are also provided in some screens to help users understand the system better.

All options and actions are checked for validity. If the user's choice of action or input data is not permitted or contradicts data provided previously, warning messages will be displayed.

CHAPTER 4

MKPARSE

4.1. Overall Structure of Mkparse

Mkparse consists of several modules. These modules are organized as a tree structure. Each module was designed to handle one specific task and is used only by its predecessor module in the tree hierarchy. Mkparse was developed according to standard principles of software engineering. It is easily modifiable and maintainable during its lifetime. Figure 4.1 shows the overall structure of Mkparse.

Some modules perform operations on the data and provide the user with a list of choices to go down one level. Some modules on the lowest level perform just one operation, such as reading a file and initializing internal data structures. The seven screens used in Mkparse's full-screen interactive and menu-driven interface are grouped into three different types manipulated by three screen modules, screen1 to screen3. They provide all the screen display and update functions. In addition, Mkparse is designed as an editor for its products, the generated parsers. It therefore combines an interactive editor with automatic semantic checking and a parser generator. It may be used to edit a previously created parser in order to expand or modify the

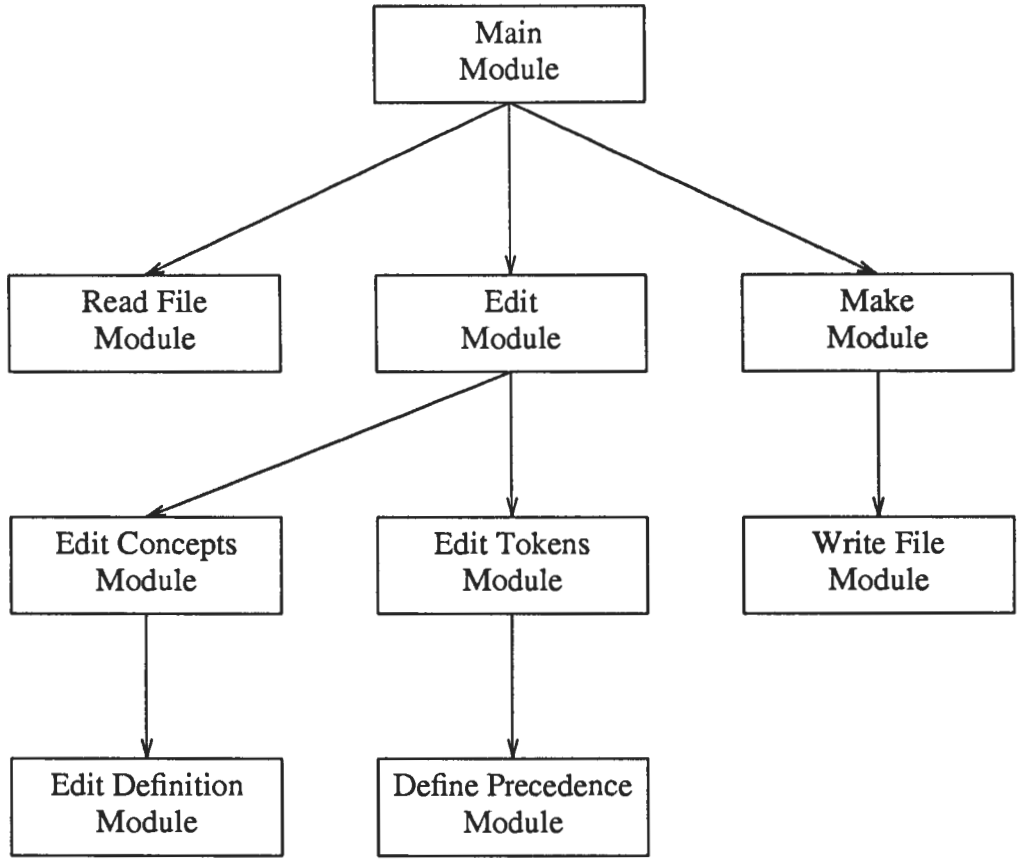


Figure 4.1. The module structure of Mkparse

language specification.

The name and purpose of each module is as follows.

- The **Main** module controls the other modules.

- The **Read File** module reads a file which contains a specification of a language. The specification is placed at the beginning of the file as a C comment.

- The **Edit** module performs editing of concepts and tokens. It leads the user down one level in the module hierarchy.

- The **Edit Token** module performs the task of assigning numbers to tokens by various mechanisms. These mechanisms include reading a scanner file to obtain token numbers, using default rules, and manual action.

- The **Edit Precedence** module permits the user to define precedences for some tokens -- normally tokens that represent operators.

- The **Edit Concept** module provides users with the ability to add or delete concepts. On each invocation, it permits the user to edit the definitions of a single concept.

- The **Edit Definition** module is a module at the lowest level of Mkparse. It allows the user to add or to modify the definitions of one concept, including the abstract definition, corresponding concrete definition, and, optionally, associ-

ativity and precedence.

- The **Mkparse** module outputs a parser file. The file consists of two parts. The first part is the specification of the language, stored at the beginning of the C source code file as a comment. This permits the file to be reread by Mkparse. The second part is the parser itself. If the definition of the language is complete, both parts are output. Otherwise, only the first part, which describes a partially defined language, is output. In this way, the user is able to save his/her partially defined language to be completed later. It invokes the Write File module to write the definition part.

- The **Write File** module writes a specification of a language to the parser file in a form suitable for re-input. A specification includes all the information the user has entered.

Mkparse supports two kinds of input files. They are scanner files and parser files. The latter may contain only the language specification part if the language is not completely defined. A scanner file, the product of Mkscan, contains the specification of the token number interface which is used to communicate between a scanner and a parser. Mkparse reads a scanner file to obtain these token numbers. See Figure 2.8, which shows Mkparse in Mk* series. A parser file, the final product of Mkparse, contains a

parser if the language is completely defined. It also includes the full grammar definition so that it can be re-input by Mkparse for subsequent modification.

Mkparse translates a user's specification of a language into a Yacc [J1] input file that includes semantic actions to build an AST. The Yacc parser calls an external scanner named "yylex" which can be a scanner generated either by Mkscan [H4] or by Lex [L1]. This allows the user to use the Yacc specification immediately to produce a parser file "y.tab.c", or to add other semantic actions to the Yacc specification and then run Yacc.

There are four main data entry areas in Mkparse. One called the abstract syntax definition part, or tree rule part, is designed for the entry of abstract syntax in the form of two-level sub-ASTs. Another called the concrete definition part, or BNF part, is designed for the entry of concrete syntax in a form similar to BNF productions. The third area is for entering the definition of operator associativities and precedence of the current rule like that in Yacc [J1]. The fourth one is for entering the definition of precedences for tokens. Two additional data entry areas are used for editing concepts and token numbers.

There are three types of symbols: concept symbols, token symbols and special symbols. Concept symbols represent semantic concepts of a programming language. They are used in defining sub-ASTs in the abstract definition part as the concepts of the

language, and in the concrete definition parts as nonterminal symbols in BNF notation. Token symbols are only used in the concrete definition part and the associativity and precedence definition part. Special symbols are used as metasymbols in Mkparse. The user cannot use these special symbols in the definition of language.

Figure 4.2 shows the overall Mkparse menu structure. Each menu contains help and quit options. If help is selected, information explaining the purpose of each menu

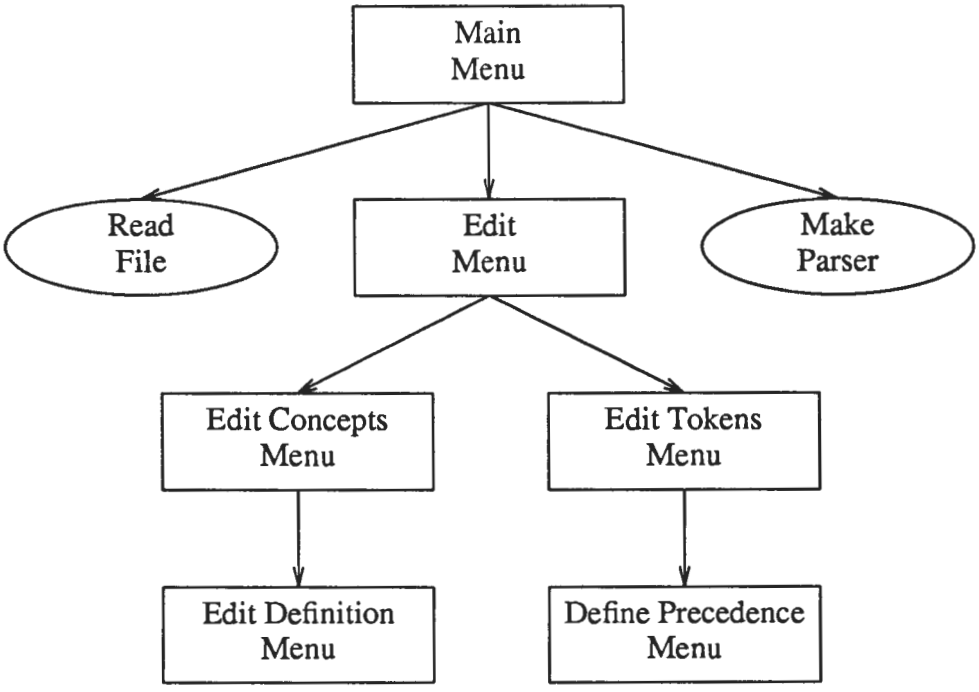


Figure 4.2. The menu organization for Mkparse

option is displayed. If quit is selected, the user is, with one exception, returned to the menu on the next higher level of the hierarchy. The exception occurs if quit is selected in the main menu and this causes Mkparse to be terminated. For every read or write file selection, a default file name is always displayed, and it will be used if a user does not provide a replacement name. The default name will be the last name used for this option or, failing this, an initial default name built into Mkparse. The following sections discuss these menus and their functions in more detail.

4.2. Main Menu

Menu

[e] edit

[r] read a grammar file

[m] output parser

[h] help

[q] quit

Functions

1. List the above menu.

2. Invoke the edit module to edit grammar rules and token numbers.

3. Read a specification file to obtain a specification of a language. The file containing a partially or fully defined language has been written by Mkparse previously.
4. Invoke the Make Parser module to output a parser.
5 Display a help message.
6 Terminate Mkparse.

This is the first menu on the top level in the menu organization of Mkparse. Before reading files, the file statuses are always checked. For an input file, relevant information is displayed if a file does not exist, is unreadable, or has inappropriate attributes. While reading a file, the correctness of the file contents is checked. The reading process will quit if the format is not correct or the file does not contain a proper definition. Before terminating Mkparse, if the language has been changed since the last time it was written to a file, a warning message is displayed to avoid quitting by mistake.

Before outputting a parser, the whole definition is checked. If the definition is not complete, only the specification will be written. If the definition is complete, both the definition and a parser are written. Mkparse writes a parser as a Yacc specification with associated semantic actions to build an AST according to abstract syntax definition. Before writing to a file, relevant information is always displayed if a file with the same name exists and if the file has inappropriate attributes. If the file exists, a

user will be further asked if the file is allowed to be over-written. In a C comment part at the beginning of the file, a complete definition of the language is written for the purpose of simplifying re-input.

4.3. Edit Menu

Menu

[c] edit grammar concepts

[t] edit token numbers

[h] help

[q] quit

Functions

1. Display the above menu.
2. Invoke the edit grammar concepts menu.
3. Invoke the edit token numbers menu.

This menu divides the editing work into two parts. One is to edit the token numbers. The other is to edit the definition of a language.

4.4. Edit Token Numbers

Menu

[r] read scanner file

[a] assign defaults to all tokens

[p] partial default

[c] change token number

[d] define precedence

[h] help

[q] quit

cursor movements:

b(begin),e(end),n(next),l(last),j(down),k(up)

Functions

1. Display the above menu and a list of tokens with their numbers. Using the cursor movement options, the user can easily move the cursor up, down, to the beginning or the end of the token list, and to the next or the previous page of the list if the tokens can fill more than one screen.
2. Read the scanner file to obtain token numbers for tokens whose names are the same as those in the scanner file.

3. Assign default token numbers to all tokens. These numbers are assigned in sequence, starting from zero.

4. Repeatedly assign a token number, which is the smallest unused natural number, to a token which has not yet been assigned.

5. Change or assign a token number to a token by typing the number from the keyboard.

6. Invoke the Define Precedence module to define precedences for some tokens. This is an optional choice, as will be explained later.

Because token numbers are used to communicate between a scanner and a parser, the same terminal symbol (token) must be represented by the same number (the identifier of the symbol) in the scanner and the parser. Mkparse provides a convenient way to obtain token numbers from a scanner file by reading the scanner file if it exists. Mkparse can recognize the token definition part in a scanner file and obtain the token numbers for those tokens having the same names. Because each token normally has its own syntactic meaning, it has a unique token number. For example, the keywords "begin" and "end" in the Pascal language represent different syntactic units, they must not use the same token number. However, there are also cases in which several terminal symbols have the same syntactic meaning and therefore share the same token number. For example, the terminal symbols "&" and "and" in the Modula-2 language have the same token number. Relevant information will always be displayed

after a duplicate token number has been typed in by using the [c] option. Before quitting to a higher level, a warning message will also be displayed if there are duplicate token numbers or if not all tokens have been assigned numbers.

4.5. Define Precedence

Menu

[a] add

[i] insert

[c] change

[d] delete

[q] quit

k-up, j-down

Function

1. Display the above menu and the list of token precedences if they are defined.
2. Add, insert, delete, and change one level in the token precedence list.

This menu permits the user to define the precedences for tokens. The precedences appear as a list of lines on the two-line window on the screen. Each line, with a number denoting the precedence level, contains one or more tokens. Tokens on the same line have the same precedence. Each token is allowed to appear only once in a

precedence definition. To simplify use, only tokens which have been used so far in the concept definitions can be assigned precedences. The checks are performed to avoid inconsistent and duplicate definitions. Defining precedences for symbols which have been used as concepts or are metasymbols is not allowed and therefore will be rejected by Mkparse. Up to twenty levels of precedence can be defined. The definition of precedences for the operators,

+ - * / < > =

might look as following:

level 0: + -
level 1: * /
level 2: < > =

4.6. Edit Concepts

Menu

[c] define concept

[#] modify ?th definition

[a] add a new concept

[i] insert a new concept

[d] delete a concept

[h] help

[q] quit

Functions

1. Display a list of concepts. A user can locate a concept by moving cursor up, down, to the beginning or the end of the concept list, and to the next or the previous page of the list if the list can fill more than one window.
2. Invoke concept definition module to edit the definitions of the current concept(which the cursor is on). If the concept has at least one definition, the first definition will be reached. If the concept has not been defined yet, a new definition format will be displayed.
3. To modify the n-th definition according to the number n input by the user.
4. Add a new concept after the current cursor position.
5. Insert a new concept before the current cursor position.
6. Delete a concept from the list, but only if the concept is not used and not defined.

We assume that the first concept in the list is the first nonterminal symbol in the corresponding BNF notation. Each concept is displayed with some additional information. The information includes the number of times it has been used in definitions (including both in the abstract part and concrete part), and the number of definitions it has (the number of production rules with this left-hand-side symbol). The information is updated dynamically while a user is editing. Before quitting to a high level, a warning message will be displayed if there are concepts which have been used but not

defined. A concept is not allowed to have the same name as a token and a concept is only allowed to appear in the list once. Therefore, each time the user attempts to enter a new concept, Mkparse must check these criteria. If the check fails, relevant information will be displayed and the input concept will be rejected. A concept can be deleted only when it has no definitions and has no references.

To simplify use, a concept can either be added to the list explicitly in this menu or implicitly by a sub-AST definition. Every time a new symbol is entered in the tree rule part, it is considered to be a new concept and added to the concept list automatically. However, the first concept needs to be entered in this menu since a user cannot reach a concept definition part before the concept appears on the list. In this way, Mkparse allows the user to concentrate on the structure of a language without bothering the user with less important details. Like concepts, any symbols first appearing in a concrete definition part but not in a sub-AST will be considered to be tokens and added to the token list automatically.

4.7. Define Concept

Menu

[i] insert

[a] add

[d] delete

[n] next form

[p] previous form

[w] new form

[u] undo

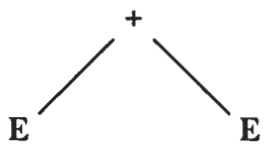
[h] help

[q] quit

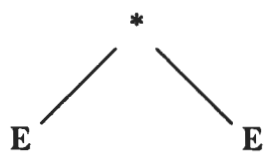
Functions

1. Display a screen for abstract syntax, concrete syntax and associativity and precedence definitions and list the above menu.
2. Insert or add one or more symbols at the root part of the sub-AST, or at the leaf part of the sub-AST, or in the concrete structure. Add the associativity definition or precedence definitions.
3. Delete a symbol from the abstract definition part or the concrete definition part. Delete the associativity definition or precedence definition.
4. Undo the last action(including addition, insertion, and deletion).
5. Edit the next form of definition for the current concept being edited.
6. Edit the previous form of the definition for the current concept.
7. Edit a new form.

The screen is divided into three parts for entry of abstract syntax, concrete syntax, associativity and precedence. Mkparse allows the user to input a subtree of the abstract syntax tree in the abstract syntax definition part. As mentioned before, the abstract part consists of a two-level sub-AST. Normally in the tree rule part, on the root level there is only one symbol, which is used as a tag for this sub-AST when building the AST during the parsing process. A tag is used to mark a node in an AST. Because most terminal symbols are not stored in an AST, different sub-ASTs might have the same children. For example, sub-AST



and sub-AST



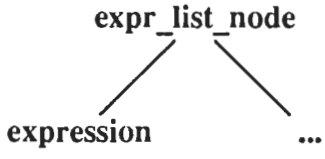
have the same children. So tag "+" and tag "*" are used to differentiate them. There

are usually several symbols at the leaf level. Corresponding to the sub-AST, in the concrete syntax part, there is a string of nonterminal and terminal symbols representing a right hand side of a BNF rule. For each nonterminal symbol, there is a corresponding concept in the sub-AST. The user might see a normal case of definition in Mkparse for the concept **statement** in Figure 3.2.

There are four special cases in which special symbols are used to facilitate the definitions.

In the first case, for a rule with an empty right-hand-side, the special word **EMPTY** is needed in the concrete structure to indicate this special case.

In the second case, a list of the repetition of a nonterminal symbol and a string of terminal symbols can be represented using two simple formats in the abstract and the concrete parts. The following is an example of how the formats look in Mkparse. In the abstract syntax definition of the concept **expression_list**, the user might see



While, in the concrete syntax definition, the user might see

begin LIST1 expression SEP ; ENDLIST end

to represent the grammatical structure of a list which has the structure

begin expression ; expression ; ..... ; expression end.

While outputting parsers, Mkparse translates the above structure into the following three BNF rules:

```
expression_list    ::=   begin LISTexpression end
LISTexpression     ::=   expression
                       |  LISTexpression ; expression
```

where, **expr_list_node** is the node name of this sub-AST, **expression_list** and **expression** are concepts, **LISTexpression** is a new nonterminal symbol created by Mkparse and used only in output, **begin** and **end** are terminal symbols used in the language being defined, **;** is a terminal symbols used as the list separator, **LIST1**, **ENDLIST**, **SEP**, and **...** are metasymbols of Mkparse.

The complete format accepted by Mkparse is as follows.

α LIST1 nonterm β SEP γ ENDLIST δ

where,

α, β, γ, and δ are strings of zero or more terminal symbols, **LIST1**, **SEP**, and **ENDLIST** are metasymbols in Mkparse, **nonterm** is a concept (nonterminal symbol). α represents the symbols that precede the list. **LIST1** implies that the list contains one or more occurrences of the concept **nonterm** with each occurrence followed by the symbols in β. **SEP** introduces the following symbols, i.e. the symbols in γ, as the list

separator. **SEP** is optional, i.e. **SEP** can be omitted if **γ** is empty. **ENDLIST**, which can be omitted if **δ** is empty, denotes the following symbols, i.e. the symbols in **δ**, appear at the end of the entire list. The above rule will be expanded into following three BNF rules in the output,

nontermlist	**::=**	**α LISTnonterm δ**
LISTnonterm	**::=**	**nonterm β**
	|	**LISTnonterm γ nonterm β**

In a similar manner to the second case, the third case allows the user to define the list which may be empty. The format is:

α LIST0 nonterm β SEP γ ENDLIST δ.

LIST0 is a metasymbol and represents the list which contains zero or more occurrences of the concept **nonterm** and the symbols in **β**. The tree rule used in this case is the same as in **LIST1** case. In order to deal with the empty case, one more rule

nontermlist ::= α δ

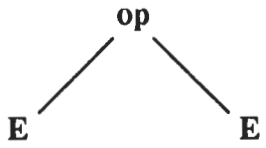
is needed here. The whole definition is expanded into four BNF rules in the output. They are:

nontermlist	**::=**	**α LISTnonterm δ**
	|	**α δ**
LISTnonterm	**::=**	**nonterm β**
	|	**LISTnonterm γ nonterm β.**

The last special case occurs when the user defines the grammar structure, like arithmetic expressions. For example, the grammar rules

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E$$

have the same left-hand-side symbol and a tree structure with the form



where **op** represents each of four operator symbols. Mkparse permits these kinds of rules be entered in a single definition form:

which is equivalent to four sub-ASTs with the root names of +, -, *, and /. In the concrete syntax part, Mkparse permits a similar form to that used in BNF:

$$E + E \mid E - E \mid E * E \mid E / E$$

where | is a metasymbol representing alternation.

In the associativity definition part,

LEFT + - * /

could be used to denote that these four operators associate to the left.

Every time the user inputs a symbol, Mkparse does some checking and prints out a message informing the user what kind of symbol it is. If the newly input symbol is not allowed to be in the current part (tree, concrete or associativity and precedence parts), an explanatory message will be given.

Before quitting from the Edit Definition module or transferring to another form of definition, Mkparse performs general checking of the current definition format to avoid later errors. Mkparse checks the following points. In a normal case, every concept on the leaf level of the sub-AST must match a nonterminal symbol in the concrete definition part. Each sub-AST must have a name. In special circumstances, each case must be checked individually. If the current definition does not satisfy the checking conditions, error messages will be displayed and the user will be asked to modify the format or simply delete the whole definition before quitting or transferring.

CHAPTER 5

EXPERIENCE WITH MKPARSE

We have used Mkparse with three different grammars to test its reliability and usability. The grammars we used were a grammar for arithmetic expressions, a grammar for the Pascal language, and a grammar for a subset of the C language. As a result of our experience in using Mkparse on these grammars, we changed some functions and formats and, at the same time, made it simpler and more flexible. The main improvements we made include: the method of defining precedences, the format for defining **list** structures, and the maximum number of precedence levels allowed.

5.1. Defining Precedences

The arithmetic expression grammar was used mainly to test the functions of defining precedences and associativities for tokens. The BNF form of this grammar is as follows (See Appendix A for a complete example of using Mkparse for this grammar) :

```
Expr    ::=   Expr + Expr
          |   Expr - Expr
          |   Expr * Expr
          |   Expr / Expr
```

| Expr ** Expr
| (Expr)
| identifier
| constant

with operator precedences separately defined as :

level 0: + -
level 1: * /
level 2: **

and operator associativities as :

LEFT: + - * /
RIGHT: **

In the first version of Mkparse, the user had to define precedences of tokens in separate **Define Concept** screens where the relevant tokens are used. Several operators could be written in a single line separated by the metasymbol **!!** to indicate groupings into precedence levels. For example, on the screen to display one form of concept definition, the user might see

PRECEDENCE: + - !! * /

where + and - are shown to have the same precedence, which is a lower precedence than * and /. Similarly * and / have the same precedence. On another screen where ** is used, the user might see,

**PRECEDENCE: * !! **

where ** is defined to have higher precedence than *. Thus the precedences for all the operators are ordered as :

+ - < * / < **

However, this specification style has two potential problems. One problem would be revealed as a definition cycle while the other might cause an ambiguity. For example, if the following precedences are defined in two forms,

PRECEDENCE: + - !! * /
PRECEDENCE: / !! ** !! +

then the precedences would have the following partial orderings:

+ - < * / and / < ** and ** < +

When combined, there is a cycle.

And in the following definitions,

PRECEDENCE: + !! ++
PRECEDENCE: ** !! ++

the relative precedence levels of + and ** can not be determined.

Although it can be argued that the above two cases are caused by the user's carelessness and could be detected by Mkparse before outputting a parser, they become impossible when the current format of defining precedences is used. In the current scheme, the above two examples would be entered by the user as follows :

level 0: + -
level 1: * /
level 2: **

and as :

level 0: +
level 1: **
level 2: ++

With this style of input, the user can define precedences for tokens which have already been referenced in concept definitions. The user is allowed to add, insert, delete, or modify any precedence levels. Each token is allowed to appear only once, i.e. in only one precedence level.

Creating a parser for a simple experimental grammar like this takes only a couple of minutes using Mkparse.

5.2. Defining List Structures

The second grammar we used was a grammar for the Pascal language (See Appendix B for this LALR(1) grammar). In Mkparse, the concrete BNF rules for list structures, such as

type_list ::= type | type_list , type

are replaced by an extended notation. For example,

LIST1 type SEP , ENDLIST

is used instead of the above rules. We found that this format is both more convenient to use and more understandable.

We soon found that many lists of nonterminal symbols can be empty, i.e. a list can contain zero occurrences of the nonterminal symbol. But our early version of Mkparse offered only the **LIST1** format which was designed to define only non-empty lists (a list containing one or more occurrences of a nonterminal symbol). If we were forced to use **LIST1** format to define a list which may contain an empty structure, we would have to add two more rules. For example, if we define the **statement_list** concept, which may contain zero or more statements separated by semicolons, as follows:

```
statement_list    ::=   LIST1  a_statement  SEP  ;  ENDLIST
a_statement       ::=   EMPTY
                    |   statement
```

To simplify usage, we added a new notation, **LIST0**, to the early version of Mkparse to describe empty lists. Using **LIST0**, the concrete syntax for the above definition can be re-written as follows,

LIST0 statement SEP ; ENDLIST

Furthermore, we found that it was more convenient for the user to be able to define a starting string of terminal symbols α, an ending string of terminal symbols δ, a repetition of nonterminal symbols following by zero or more terminal symbols β, and a string of terminal symbols used as a separator γ. Therefore, the complete syntax for lists in Mkparse have the general structure:

α LIST0 N β SEP γ ENDLIST δ

α LIST1 N β SEP γ ENDLIST δ

where α, β, γ, and δ are all optional strings of terminal symbols, and **N** is the concept

that may be repeated.

The grammatical metasymbols, **LIST0** and **LIST1**, conceal some grammatical complications. These metasymbols are automatically eliminated when a parser is output. Some new nonterminal symbols will be added to the BNF grammar during the elimination process. For example, if the **A_list** concept is defined as follows,

LIST1 A SEP , ENDLIST

which will be automatically expanded to

```
A_list        ::=   $SA_list  LISTA_list  $EA_list
LISTA_list    ::=   A
                  |  LISTA_list , A
```

where nonterminal symbols **$SA_list**, **$EA_list**, and **LISTA_list** are all new nonterminal symbols created by Mkparse. The nonterminals **$SA_list** and **$EA_list** are defined to have empty righthand sides. They are used to help construct the abstract syntax tree during the parsing process. When possible, Mkparse tries to create meaningful names. If one of those names, such as **$EA_list** here, is involved in a grammatical conflict, the names used in the error messages will lead the user to the definitions which cause the error. Meaningless names, such as **0001**, which are often generated by automated tools, would not help the debugging process.

5.3. Precedence Levels, etc.

The last example we tried was a grammar for a subset of the C language [H1][H5], in which the "typedef" construct and related rules were removed from the C language (See Appendix C for this subset of C). All features of Mkparse were tested and worked well. However, the number of precedence levels accepted by Mkparse had to be enlarged from ten to twenty to deal with the C language which has fifteen precedence levels. While Mkparse was being used, help information, checking information, and error message were also improved to be more precise and intelligible.

Combining rules with the same sub-AST structure proved to be very convenient. With Mkparse, we found that most time is spent mainly on typing in the grammar rules. Since most of the erroneous symbols that are entered are detected by Mkparse at the time they are typed in, much subsequent debugging work is avoided.

Because the definition of the syntactical structure of a language is usually more complicated than the definition of its lexical structure, Mkparse needs more interaction with the user than Mkscan. Much extra interaction would normally be caused by LR conflicts in the grammar. Integration of Mkparse with an incremental parser generator would be a good improvement because grammatical problems would be detected immediately. This will be discussed in the next chapter as future work.

Due to limitations imposed by CRT screens with 24 lines and 80 columns, symbols with short names are encouraged. This would tend to reduce the comprehensibility of names. In conjunction with Mkscan, the previous tool of Mkparse in the Mk* series, or with Lex [L1], Mkparse provides an easy and efficient way of generating a parser.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1. Conclusions

The interactive parser generator, Mkparse, described in this thesis was developed by the author. It can be used to generate parsers for LALR(1) grammars. Mkparse follows design principles and has features that are consistent with the other tools in the Mk* series.

Mkparse has some unique features. The use of abstract syntax to think about the structure of a language and to define a language makes it easier and more natural to capture the main structure of the language without being bothered by other less important details such as the keywords of the language. To define the abstract syntax of a language by the two-level sub-ASTs and the concrete syntax by BNF rules is direct and natural.

Instead of outputting a conventional parse tree, a parser generated by Mkparse creates an AST after successfully parsing text. The AST, contains all the information about the text needed by later phases in a compiler, and is less complicated than a parse

tree for the same text. It therefore occupies less memory than a parse tree. It is also more efficient for the later phases in a compiler to operate on an AST than on a parse tree.

Mkparse has much in common with other interactive software tools. With the menu-driven and full screen interaction interface, the user is clearly informed what happens and what can be performed. The user can traverse the menu structure easily in Mkparse by choosing menu options. With the on-line help information, the user can get to know the details of the functions performed by each menu. It helps a novice understand the structure of the tool and the functions the tool performs. These help to achieve the goal of reducing the user's learning effort.

While running Mkparse, as many dynamic checks as possible are performed and relevant messages are fed back to the user. Every time the user inputs a symbol or chooses an option on a menu, he/she gets the feedback information immediately. In this way, many obvious errors and inconsistencies, such as using a token as a concept, can be avoided. Having chosen an option to finish one definition screen (for concepts, tokens, etc.) or to generate a parser, overall checks are also performed to see if the current definition is incomplete. This protects the user from more substantial errors, such as incompleteness, inconsistency, or conflicts in definitions of concepts, associativity and precedence. The information is important to a novice. During the design and the implementation of Mkparse, effort was also made to reduce the movements of

users' eyes and the number of key strokes.

Besides the many checks and the many feedback messages generated during the editing of languages, the final LR grammar diagnostic information including LR conflicts detected is available to help the user debug the language. However, more helpful information such as the traces of the conflicts in LR states and better methods of checking a language definition as could be provided by an incremental LR parser generator would be desirable features as well. These features would also have been provided if the time had permitted. However, they must be left as future work.

6.2. Future Work

It is a desirable feature of an interactive LR parser generator to report any conflicts immediately after the user adds a new grammar rule. A straightforward approach to perform this job would be to re-apply the LR state building algorithm to the new grammar after each change. However, re-building all LR states as each production rule is added for a small size grammar like Pascal requires a considerable amount of CPU time. The time spent on checking every partially defined grammar just to detect conflicts is longer than what a user is likely to tolerate. The computation cost is expensive too. Therefore this method is impractical and unacceptable. An alternative approach would be to use a proper incremental LR parser generator.

An incremental LR construction algorithm would work as follows. Instead of building the LR states from very beginning, as for a new grammar, it adds new LR states to the set of LR states that were built previously. Most LR states are unchanged by the addition of a new grammar rule. New states are kept only if there are no conflicts detected, otherwise, the new states should be abandoned. The information reported to the user could include if the new grammar is a SLR(1), or an LALR(1), or an LR(1) grammar, what types of conflicts have occurred, and details of the conflict states, etc. Inefficiencies may arise if the user deletes or modifies an early definition such as the first rule in the grammar. The correct LR states are maintained dynamically while adding, deleting, or changing the language definition. The biggest advantage of an incremental LR construction algorithm is in reporting errors, including shift/reduce and reduce/reduce conflicts, immediately. It should greatly improve the efficiency and productivity of the user.

Besides the incremental LR construction algorithm, helpful and easily understandable error information is important to help the user locate errors and debug the grammar. In addition to conventional error information, diagnostic output should also include the explanations of LR conflicts, perhaps similar to the information provided by DeRemer's method for computing LALR look-ahead sets [D2]. Reporting the error traces is more useful than reporting only the errors themselves.

The above two features will certainly make a parser generator more sophisticated, but implementation of these features would require much work.

BIBLIOGRAPHY

[A1] Aho, A.V., Ullman, J.D., "Optimization of LR(k) parsers", J. Computer and System Science, 6, 6, P573-602, 1972.

[A2] Aho, A.V., Sethi, R., Ullman, J.D., "Compilers: Principles, Techniques, and Tools", Addison Wesley, 1986.

[A3] Anderson, T., Eve, J., Horning, J.J., "Efficient LR(1) Parsers", SIAM J. Computing, 2, 2, P106-127, 1973.

[A4] Arnold, K.C.R.C., "Screen Updating and Cursor Movement Optimization", UNIX system document.

[B1] Backhouse, R.C., "Syntax of Programming Languages Theory and Practice", Prentice-Hall International, 1979.

[C1] Cleaveland, J.C., Uzgalis, R.C., "Grammars For Programming Languages", Elsevier Computer Science Library, 1977.

[D1] DeRemer, F.L., "Simple LR(k) Grammars", Comm. ACM, 14, P453-460, 1971.

[D2] DeRemer, F.L., Pennello, T., "Efficient Computation of LALR(1) Look-Ahead Sets", ACM TOPLAS, Vol.4, No.4, P615-649, Oct 1982.

[G1] Galitz, W.O., "Handbook of Screen Format Design", QED Information Sciences, Inc., Wellesley Hill Massachusetts, 1986.

[G2] Goos, G., Hartmanis, J., "Lecture Notes in Computer Science: Compiler Construction", Springer-Verlag, 1976.

[G3] Goos, G., Hartmanis, J., "Lecture Notes in Computer Science: Semantics-Directed Compiler Generation", Springer-Verlag, 1980.

[H1] Harbison S.P., Steele Jr., G.L., "C -- A Reference Manual", Prentice-Hall, Inc., Englewood Cliffs, N.J., 1984

[H2] Holt, R.C., Cordy J.R., Wortman, D.B., "An Introduction to S/SL: Syntax/Semantic Language", ACM TOPLAS, Vol.4, No.2., P149-178, April 1982.

[H3] Hopgood, F.R.A., "Compiling Techniques", MacDonald Computer Monographs, 1969.

[H4] Horspool, R.N., Levy, M.R., "Mkscan -- An Interactive Scanner Generator", Software -- Practice and Experience, to appear.

[H5] Horspool, R.N., "C Programming in the Berkeley UNIX Environment", Prentice-Hall Canada Inc, 1986.

[J1] Johnson, S.C., "YACC: Yet Another Compiler - Compiler", UNIX system documentation, also available as: Computing Science Tech. Report 32, AT & T Bell Laboratories, Murray Hill N.J., 1978.

[K1] Kernighan, B.W., Ritchie, D.M., "The C Programming Language", Prentice-Hall Inc., 1978.

[L1] Lesk, M.E., Schmidt, E., "Lex - A Lexical Analyzer Generator", UNIX system documentation, also /vailable as: Computing Science Tech. Report 39, Bell Laboratories, Murray Hill N.J., 1975.

[M1] Marca, D., "Applying Software Engineering Principles", Little, Brown & Company, 1984.

[T1] Thomas, J., "Organizing for Human Factors", in *Human Factors and Interactive Computer Systems*, Y. Vassilous (ed.), Ablex, Norwood, NJ, 1984.

[W1] Waite, W.M., Goos, G., "Compiler Construction", Springer-Verlag New Youk Inc., 1984.

[W2] Wasserman, A.I., "Tutorial: Software Development Environments", IEEE Computer Society, Computer society Press, The Institute of Electrical and Electronics Engineers, Inc., 1981

[W3] Wexelblat, R.L., "History of Programming Languages", Sperry Univac, Blue Bell, Pennsylvania, 1981.

APPENDIX A

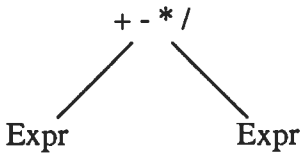
AN EXAMPLE OF USING MKPARSE FOR AN EXPRESSION GRAMMAR

GRAMMAR:

Expr	::=	Expr + Expr
	|	Expr - Expr
	|	Expr * Expr
	|	Expr / Expr
	|	Expr ** Expr
	|	(Expr)
	|	identifier
	|	constant

CONCEPT DEFINITIONS:

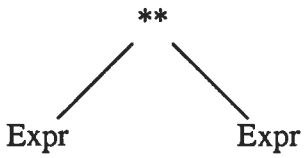
CONCEPT Expr FORM # 1 (OLD)



Expr + Expr !! Expr - Expr !! Expr * Expr !! Expr / Expr

ASSOCIATIVITY: LEFT + - * /
PRECEDENCE:

CONCEPT Expr FORM # 2 (OLD)



Expr ** Expr

ASSOCIATIVITY: RIGHT **
PRECEDENCE:

CONCEPT Expr FORM # 3 (OLD)

()
|
Expr

(Expr)

CONCEPT Expr FORM # 4 (OLD)

const_node id_node

---

constant !! identifier

PRECEDENCE DEFINITIONS:

level 0: + -

level 1: * /

level 2: **

TOKEN NUMBERS:

identifier	1
constant	2
(	3
)	4
+	5
-	6
*	7
/	8
**	9

APPENDIX B

A PASCAL GRAMMAR USED FOR TESTING

```
prog        PROGRAM id ( id_list ) ; block .

block       declarations BEGIN statements END

declarations  optlabels optconsts opttypes optvars optprocs

optlabels
        | labeldecs ;

opttypes
        | types ;

optconsts
        | consts ;

optvars
        | vars ;

optprocs
        | procs ;

labeldecs       LABEL integer
        | labeldecs , integer

consts      CONST id = constant
        | consts ; id = constant

types       TYPE id = type
        | types ; id = type

vars        VAR id_list : type
        | vars ; id_list : type

procs       procdef
        | procs ; procdef
```

```
procdef     PROCEDURE id ; body
        | PROCEDURE id ( fp_list ) ; body
        | FUNCTION id : id ; body
        | FUNCTION id ( fp_list ) : id ; body
        | FUNCTION id ; block

body        block
        | FORWARD

type        id
        | ( id_list )
        | id .. constant
        | const_expr .. constant
        | ^ id
        | packed ARRAY [ type_list ] OF type
        | packed RECORD field_list END
        | packed SET OF type
        | packed FILE OF type

packed
        | PACKED

type_list   type
        | type_list , type

field_list   invariant_part variant_part

invariant_part field
        | invariant_part ; field

field
        | id_list : type

variant_part
        | CASE id OF variant_cases
        | CASE id : id OF variant_cases

variant_cases variant_case
        | variant_cases ; variant_case

variant_case
        | constant_list : ( field_list )
```

```
constant_list constant
        | constant_list , constant

statements    statement
        | statements ;  statement

statement     labels matched_else
        | labels unmatched_else

labels
        | integer : labels

matched_else
        | id := expression
        | var_expr := expression
        | id
        | id ( arglist )
        | BEGIN statements END
        | IF expression THEN labels matched_else ELSE labels matched_else
        | CASE expression OF case_body END
        | WHILE expression DO labels matched_else
        | REPEAT statements UNTIL expression
        | FOR id := expression to expression DO labels matched_else
        | WITH var DO labels matched_else
        | GOTO integer

unmatched_else IF expression THEN statement
        | IF expression THEN labels matched_else ELSE labels unmatched_else
        | WHILE expression DO labels unmatched_else
        | FOR id := expression to expression DO labels unmatched_else
        | WITH var DO labels unmatched_else

to          TO
        | DOWNTO

case_body     case_part
        | case_body ; case_part

case_part
        | constant_list : statement

arglist         arg
```

```
                | arglist , arg

arg             expression optwidth

optwidth        optwidth : expression
                |

expressions     expression
        | expressions , expression

fp_list         fp
        | fp_list ; fp

fp              id_list : id
        | VAR id_list : id
        | PROCEDURE id ( fp_list )
        | FUNCTION id ( fp_list ) : id
        | PROCEDURE id
        | FUNCTION id : id

expression      simple_expression
        | simple_expression relop simple_expression

relop           =
        | <>
        | >
        | <
        | >=
        | <=
        | IN

simple_expression term
        | + term
        | - term
        | simple_expression addop term

addop           +
        | -
        | OR

term            factor
        | term mulop factor
```

```
mulop       *
          | /
          | DIV
          | MOD
          | AND

factor      integer
          | real
          | id
          | string
          | NIL
          | var_expr
          | id ( expressions )
          | NOT factor
          | ( expression )
          | [ set ]
               | [ ]

set         expression
          | expression .. expression
          | set , expression
          | set , expression .. expression

constant    id
          | const_expr

const_expr  + id
          | - id
          | integer
          | real
          | + integer
          | + real
          | - integer
          | - real
          | string

var         id
          | var_expr

var_expr    id [ expressions ]
          | id . id
          | var_expr [ expressions ]
```

```
        | var_expr . id
        | id ^
        | var_expr ^

id_list  id
        | id_list , id
```

APPENDIX C

A GRAMMAR FOR A SUBSET OF THE C LANGUAGE

```
goal          bof program eof

program       program top_decl
              |

top_decl      data_decl
              | func_decl

data_decl     opt_type_spec ;
              | opt_type_spec init_list ;

parm_list     parm_list parm_decl
              |

func_decl     opt_type_spec dcltr parm_list cmpd_stmt

decl          type_spec ;
              | type_spec init_list ;

parm_decl     type_spec ;
              | type_spec list_dcltr ;

type_decl     type_spec abs_dcltr

formals_decl  formal_decl
              | formals_decl , formal_decl

formal_decl   opt_type_spec dcltr

opt_type_spec type_spec
              |

type_spec     tc_spec
              | type_spec tc_spec
```

```
tc_spec         stg_class
                | type

stg_class       auto
                | static
                | extern
                | register

type            stnd_type
                | enum_type

stnd_type       char
                | float
                | double
                | int
                | short
                | long
                | unsigned
                | void

enum_type       enum id
                | enum enum_list
                | enum id enum_list

enum_dlist      enum_dcltr
                | enum_dlist , enum_dcltr

enum_list       { enum_dlist }
                | { enum_dlist , }

enum_dcltr      id
                | id = exp

p1_dcltr        id
                | ( dcltr )

p2_dcltr        p1_dcltr
                | p2_dcltr ( formals_decl )
                | p2_dcltr ( )
                | p2_dcltr [ ]
                | p2_dcltr [ exp_list ]
```

```
dcltr           p2_dcltr
                | * dcltr

init_dcltr      dcltr
                | dcltr = init_exp

init_exp_list   init_exp
                | init_exp_list , init_exp

init_exp        exp
                | { init_exp_list }
                | { init_exp_list , }

p1_abs_dcltr    ( p3_abs_dcltr )

p2_abs_dcltr    p1_abs_dcltr
                | p2_abs_dcltr ( )
                | ( )
                | p2_abs_dcltr [ ]
                | p2_abs_dcltr [ exp_list ]
                | [ ]
                | [ exp_list ]

p3_abs_dcltr    p2_abs_dcltr
                | * p2_abs_dcltr
                | *

abs_dcltr       p3_abs_dcltr
                |

init_list       init_dcltr
                | init_list , init_dcltr

list_dcltr      dcltr
                | list_dcltr , dcltr

cmpd_stmt       { stmts }

stmts           stmts decl_or_stmt
                |

stmt            bal_stmt
```

```
                | unbal_stmt

decl_or_stmt    decl
                | stmt

basic_stmt      exp_list ;
                | cmpd_stmt
                | do stmt while ( exp_list ) ;
                | break ;
                | continue ;
                | return_stmt
                | goto id ;
                | ;

bal_stmt        basic_stmt
                | while ( exp_list ) bal_stmt
                | bal_for_stmt
                | if ( exp_list ) bal_stmt else bal_stmt
                | switch ( exp_list ) bal_stmt
                | label bal_stmt

unbal_stmt      while ( exp_list ) unbal_stmt
                | unbal_for_stmt
                | if ( exp_list ) stmt
                | if ( exp_list ) bal_stmt else unbal_stmt
                | switch ( exp_list ) unbal_stmt
                | label unbal_stmt

0014            ; ) bal_stmt
                | ; exp_list ) bal_stmt

0015            ; 0014
                | ; exp_list 0014

bal_for_stmt    for ( 0015
                | for ( exp_list 0015

0016            ; ) unbal_stmt
                | ; exp_list ) unbal_stmt

0017            ; 0016
                | ; exp_list 0016
```

```
unbal_for_stmtfor ( 0017
                | for ( exp_list 0017

return_stmt     return ;
                | return exp_list ;

label           id :
                | case exp :
                | default :

literal         dec_int
                | oct_int
                | hex_int
                | in_float
                | in_char
                | in_string

paren_exp       ( exp_list )

primary_p1_expid
                | literal
                | paren_exp
                | sizeof ( type_decl )

primary_p2_expprimary_p1_exp
                | primary_p2_exp [ exp_list ]
                | primary_p2_exp ( )
                | primary_p2_exp ( exp_list )
                | primary_p2_exp . id
                | primary_p2_exp -> id

primary_exp     primary_p2_exp

postfix_exp     primary_exp
                | postfix_exp postfix_op

postfix_op      ++
                | --

prefix_exp      postfix_exp
                | sizeof prefix_exp
                | prefix_op cast_exp
```

```
                | * cast_exp
                | & cast_exp
                | negate_op cast_exp

prefix_op++
                | --

negate_op       -
                | !
                | ~

cast_exp        prefix_exp
                | ( type_decl ) cast_exp

mul_exp         cast_exp
                | mul_exp mult_op cast_exp

mult_op         *
                | /
                | %

add_exp         mul_exp
                | add_exp add_op mul_exp

add_op          +
                | -

shift_exp       add_exp
                | shift_exp shift_op add_exp

shift_op        <<
                | >>

rel_exp         shift_exp
                | rel_exp rel_op shift_exp

rel_op          <
                | <=
                | >=
                | >

equ_exp         rel_exp
```

```
                | equu_exp equ_op rel_exp

equ_op          ==
                | !=

band_exp        equ_exp
                | band_exp & equ_exp

bxor_exp        band_exp
                | bxor_exp ^ band_exp

bor_exp         bxor_exp
                | bor_exp | bxor_exp

and_exp         bor_exp
                | and_exp && bor_exp

or_exp          and_exp
                | or_exp || and_exp

cond_exp        or_exp
                | or_exp ? exp_list : cond_exp

exp             cond_exp
                | cond_exp asgn_op exp

asgn_op         =
                | +=
                | -=
                | *=
                | /=
                | %=
                | >>=
                | <<=
                | &=
                | ^=
                | |=

exp_list        exp
                | exp_list , exp
```

VITA

Surname: Zhao Given Name: Qing

Place of Birth: Shanghai, China Date of Birth: January 13, 1961

Educational Institutions Attended, with Dates of Entering and Leaving:

Fudan University, Shanghai, China 1979 to 1983

University of Victoria, B.C., Canada 1986 to 1987

Degrees, Diplomas, Etc., Awarded, with Dates and Names of Institutions:

B.Sc.(Computer Science) 1983 Fudan University

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make *single copies only* for such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Mkparse -- An Interactive Parser Generator

Author [redacted]

Zhao, Qing

July 2, 1987

Date