Compiler Design ReferencePoint Suite

SkillSoft. (c) 2003. Copying Prohibited.

Table of Contents

Point 7: Compiler Construction Using Flex and Bison

Abhiram Mishra

A compiler consists of a lexical analyzer and a parser. A lexical analyzer is a tool that scans an input file for specific patterns and returns the tokens that match the pattern. A parser is a tool that takes an input from a lexical analyzer and translates the input into the target language. The target language can be a high level language, such as C, or a low level one, such as an assembly language.

Fast Lexical Analyzer (Flex) is a tool you can use to automate the process of creating lexical analyzers. Bison is a tool you can use to automate the process of creating parsers.

Flex uses an input file that contains the description of patterns and generates the C code for the lexical analyzer. Bison uses an input file that contains the syntax of the language and generates a parser as output. The lexical analyzer generated by Flex and the parser generated by Bison combine to form a compiler.

This ReferencePoint describes how to install Flex and Bison on various platforms, such as Windows and UNIX. It also explains how to use Flex and Bison to construct a compiler for a new programming language.

Flex and Bison Overview

To construct a compiler using Flex and Bison, you need to create a pattern file for Flex and a grammar file for Bison. The pattern file contains the specification of the lexical analyzer. This file is specified in the pattern action style, where patterns are specified by regular expressions and actions are performed on the matching text in the input stream. The Flex input file contains both the patterns and the actions that are performed on text that matches the specified patterns. These patterns that are specified in the Flex input file are called rules.

Flex generates the file, lex.yy.c, as an output based on the input pattern file. The lex.yy.c file is a complete lexical analyzer and contains the yylex() function. The parser calls this function to invoke the lexical analyzer. Figure 1-7-1 shows the input received by Flex and the output generated by it:

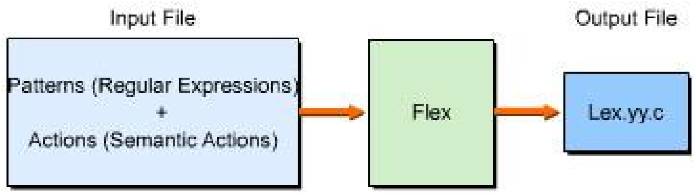

Figure 1-7-1: Input and Output in Flex

Bison takes the grammar file as the input. The grammar file has a .y extension. For example, xxxx.y is a grammar file extension. Based on the input grammar file, Bison generates the xxxx.tab.c and xxxx.tab.h output files. The grammar file contains the syntax of the language the parser needs to be built for. The syntax is written as context-free grammar, which provides a set of rules to control the ordering of words in a sentence.

The xxxx.tab.h and xxxx.tab.c files contain the token definitions and the yyparse() function, respectively. In addition, the xxxx.tab.c and lex.yy.c files use the tokens defined in the xxxx.tab.h file. Figure 1-7-2 shows the input received by Bison and the output it generates:

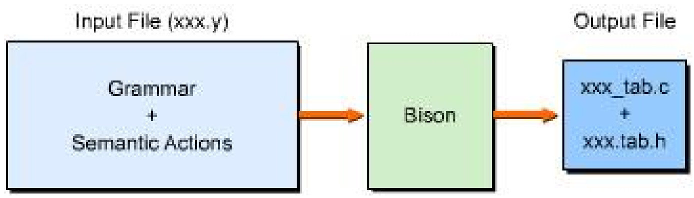

Figure 1-7-2: Input and Output in Bison

Both Flex and Bison produce C files as output. Figure 1-7-3 shows the process to construct a compiler from the output files generated using Flex and Bison:

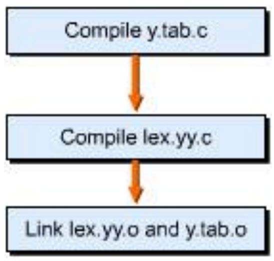

Figure 1-7-3: Creating a Compiler

Understanding Flex

Flex is a commonly used tool to develop lexical analyzers. You can install Flex on various platforms, such as Microsoft Windows, UNIX, and Linux, and construct a lexical analyzer after the rule description file is written.

Note The rule description file contains the regular expressions that define how the input is tokenized.

Installing Flex

You can download Flex for free from the Internet. To learn more about downloading Flex and licensing rights, see http://www.gnu.org/software/flex/flex.html.

To install Flex on Microsoft Windows NT, Windows 98, and Windows 2000:

1. Download the Flex compressed archive file, Flex 2.5.4a-bin.zip.
2. Decompress the file in a specified directory.

Note Before you run Flex, ensure the msvcrt.dll file is in the Windows/System folder.

To install Flex on a UNIX or Linux platform:

1. Download the Flex compressed archive file, Flex 2.5.4a-bin.zip.
2. Decompress the file in a specified directory.

3. At the command prompt, type configure –srcdir=DIR to run the ./configure script, which is supplied with the Flex archive. Here, DIR is the directory of the Flex source code.

4. Type make install to build the Flex binary.

Pattern Scanning Mechanism of a Lexical Analyzer Generated Using Flex

Flex converts the input patterns in a pattern file, which are specified as regular expressions, into the associated C code. Whenever the input file contains patterns that match the rules specified in the Flex rule definition file, the Flex-generated lexical analyzer executes the action associated with the pattern. If the lexical analyzer finds multiple matches, it takes the one that matches the most text.

A lexical analyzer generated by Flex provides access to the yytext and yyleng global pointers. The yytext variable points to the text in the input stream that matches the pattern. The yyleng variable determines the length of the text string pointed to by yytext. Flex also provides a global file pointer, called yyin. You can point yyin to the input stream to access the text from that stream. Figure 1-7-4 depicts the process of a lexical analyzer generated by Flex:

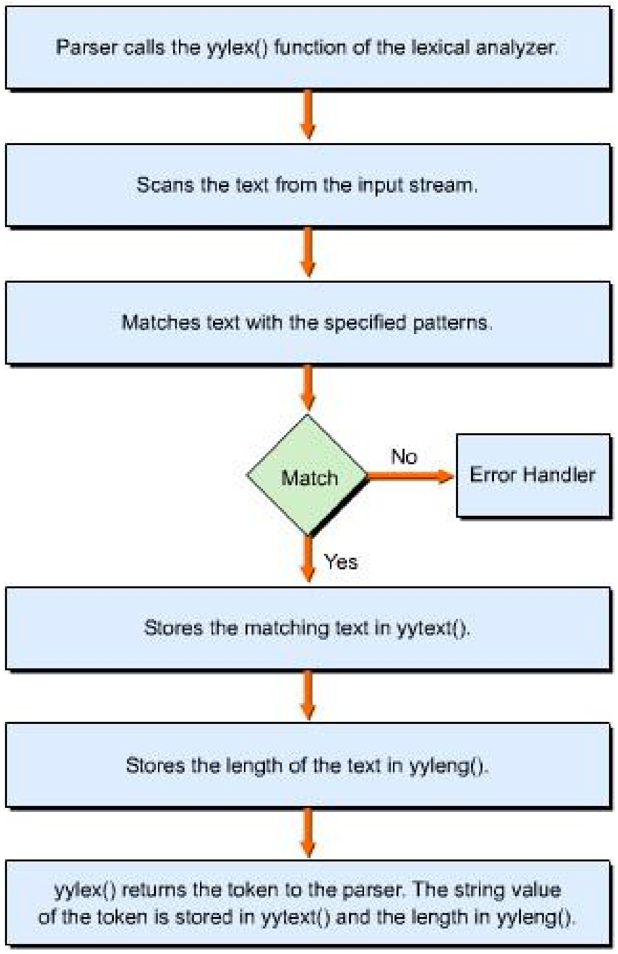
Parser calls the yylex() function of the lexical analyzer.
Scans the text from the input stream.
Matches text with the specified patterns.
Match
No
Error Handler
Yes
Stores the matching text in yytext().
Stores the length of the text in yyleng().
yylex() returns the token to the parser. The string value of the token is stored in yytext() and the length in yyleng().

Figure 1-7-4: Lexical Analyzer Process

Understanding the Syntax of a Flex Input File

A Flex file is divided into three sections - definition, rule, and user subroutine. These sections are separated by two percentage (%%) symbols. The Flex input file has a .l extension.

A valid Flex file begins with the %{ symbol in the first column of the first line. The %{ and %} symbols indicate the beginning and end of the definition section, respectively. The definition section can include C declarations such as variable declaration and other C statements.

The rule section contains the pattern specification and the C code that is executed on the matched pattern. The user subroutine section contains the user defined subroutines that are called from the rules section. The user subroutine section is copied verbatim into the lexical analyzers generated using Flex. Listing 1-7-1 shows a sample Flex specification file:

Listing 1-7-1: Flex Specification File

```
%{
#include "string.h"
int line = 0;
%}
%%
[\t]+      {;}
[0-9]+  { return NUMBER;}
\n        {return NEWLINE;}
%%
#include <stdio.h>
int main(int argc, char *argv)
{}
```

This listing shows a sample Flex specification file that consists of two sections - definition and rules. The definition section contains a header file and a declaration. The rules section contains regular expressions and their associated actions.

Writing Rules in a Flex Input File

Rules in a Flex input file consist of two parts - patterns and C code. The patterns are in the form of regular expressions. Table 1-7-1 lists commonly used patterns:

Table 1-7-1: Commonly Used Patterns

Pattern	Examples of Inputs that Match the Pattern
ABC	"ABC"
[0-9]	0, 1, 3, 4
[0-9]+	1, 123, 3434, 213
-?[0-9]+	-1, 4, 3434
[a-zA-Z][a-z-A-Z0-9]?	A1, asd23, a

Each pattern has an associated action written in C. These actions are copied verbatim into the lexical analyzer file, lex.yy.c, which is generated by Flex. Listing 1-7-2 shows a sample Flex file that counts the number of characters, lines, and words in the input:

Listing 1-7-2: Counting the Number of Lines, Words, and Characters in an Input File

```
%{
int lines = 0;;
int chars = 0;
int words = 0;
%}
%%
[^ \t\n]+        {words++; chars = chars+ yyleng;}
[\n]               { chars++; lines++; }
[\t]                { chars++;}
.                   { chars++;}
%%
#include <stdio.h>
int yywrap(){
        return(1);
}
void main(int argc, char ** argv)
{
FILE* fp;
if(argc>1)
{
fp = fopen (argv[1],"r");
if(fp == NULL)
printf("File Could not be opened");
yyin = fp;
}
yylex();
printf("chars = %d, lines  = %d, words = %d", chars, lines,words);
}
```

In the above code, the definition section of the input file declares three variables - lines, words, and chars. The rules section describes the patterns and actions to match the words, characters, and lines. The [\n] symbol matches a newline, [\t] matches a tab, and [^ matches everything apart from a newline, tab, and space. The user defined subroutines section contains the definition of two functions, main() and yywrap(). The main() function calls the yylex() function to start the lexical analyzer. The yylex() function calls the yywrap() function whenever yylex() reaches the end of the file. The yywrap() function returns one, which indicates that no more input is available.

Understanding Bison

Bison is the GNU version of the parser tool, Yet Another Compiler Compiler (YACC). Parsers generated by Bison take the input token the Flex generated lexical analyzers provide and group these tokens to match the syntax specified by the grammar file.

Installing Bison

Bison is available for various platforms, enabling you to construct a parser on various platforms after the grammar file is written.

You can download Bison for free from http://ftp.gnu.org/gnu/bison/.

To install Bison on Windows:

1. Download the compressed Bison archive, Bison-1.35-bin.zip.

2. Decompress the file in a specified directory.

3. Download the file, libintl.dll, which is a part for the Bison package.

4. Copy the libintl.dll file in the bin sub directory of Bison.

5. Download libiconv.dll.

6. Copy the libiconv.dll file in the bin subdirectory of Bison.

To install Bison on UNIX or Linux:

1. Download the compressed Bison archive, Bison-1.35-bin.zip.

2. Decompress the file in a specified directory.

3. In the directory that contains the Bison source code, type ./configure'.

4. Type make install to compile the package.

Working of Parsers Generated Using Bison

A parser generated using Bison receives tokens from the lexical analyzer and uses the tokens to match the rules specified in the grammar file. The parser then tries to find the set of rules that fully or partially match a set of collected tokens. Each time the parser reads a token and does not find any completely matching rule, it sends that token to an internally maintained stack. This process of sending tokens to the internal stack is called shifting. After shifting the token to the stack, the parser looks for the tokens that partially match the rules.

Bison generated parsers are state driven. Each state is a collection of grammar rules that can be matched. Each input token changes the state of the parser, which is based on the tokens in the internal stack. When there are enough tokens in the stack to match a rule in the state, the rule is matched. This process of matching a rule is called reduction.

When a rule is matched, the tokens that match the rule are removed from the stack and the actions associated with the rule are executed. Sometimes there are multiple rules that match the set of collected tokens. This condition is called the reduce/reduce conflict. The grammar that generates the reduce/reduce conflict cannot be parsed. This grammar sometimes also has the shift/reduce conflict, which occurs when a set of tokens completely matches one rule and partially match another rule. In such cases, the parser can either shift the token into the stack or reduce the rule. In addition, the grammar can be modified to remove the shift/reduce conflict. In the case of a shift/reduce conflict, the parser shifts the token in the stack.

Figure 1-7-5 shows the process of the Bison-generated parser:

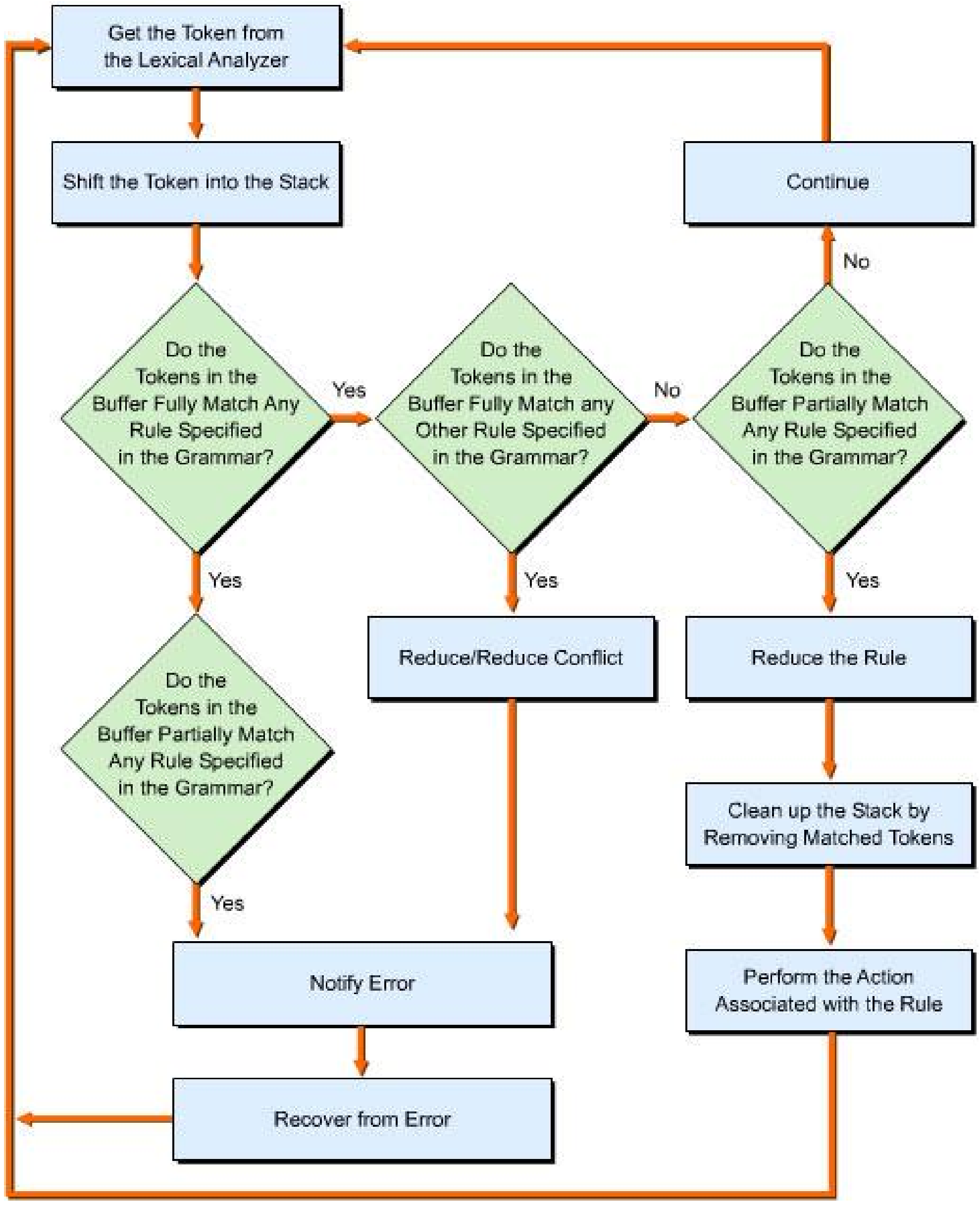
Get the Token from the Lexical Analyzer
Shift the Token into the Stack
Continue
No
Do the Tokens in the Buffer Fully Match Any Rule Specified in the Grammar?
Yes
Do the Tokens in the Buffer Fully Match any Other Rule Specified in the Grammar?
No
Do the Tokens in the Buffer Partially Match Any Rule Specified in the Grammar?
Yes
Yes
Yes
Do the Tokens in the Buffer Partially Match Any Rule Specified in the Grammar?
Reduce/Reduce Conflict
Reduce the Rule
Clean up the Stack by Removing Matched Tokens
Yes
Notify Error
Perform the Action Associated with the Rule
Recover from Error

Figure 1-7-5: How a Bison-Generated Parser Works

Understanding the Syntax of a Bison Grammar File

A Bison grammar file is structurally similar to a Flex input file and is divided into three sections - definition, rules, and user defined subroutine. All the sections are separated by the %% symbol.

The definition section includes token declaration, value types, declarations of tokens, and other C declarations. Every token declaration begins with a % symbol. The rule section contains the description of grammar rules and the associated actions. The rules in a grammar file are written in terms of terminal and non-terminal symbols.

Note A terminal symbol is a token returned by a Flex-generated lexical analyzer. You can express a non-terminal symbol in terms of terminal and non-terminal symbols. The left-hand side of each rule always has a non-terminal symbol. To learn more about terminal and non-terminal symbols, see the Developing Compilers Using C ReferencePoint.

Each terminal symbol has an associated value. For example, if a symbol represents a number, its value would be a particular number. The value of a non-terminal symbol is represented in terms of the terminal symbols.

The user subroutine section contains the user defined subroutine and functions such as main (). Listing 1-7-3 shows a sample Bison grammar file:

Listing 1-7-3: A Sample Bison Grammar File

```
%{
#include<stdio.h>
#include<string.h>
extern char* yytext;
extern FILE* yyout;
static GSASTRING currSegment;
%}
%token <cval> IDENTIFIER
%token <float_val>CONSTANT STRING_LITERAL
%%
enumerator : IDENTIFIER
           | IDENTIFIER '=' constant_expression
        ;{return 0;}
constant_expression : conditional_expression
         ;{}
%%
```

This listing shows a sample Bison specification file that consists of two sections - definition and rules. The definition section contains two header files and declarations. The rules section contains the grammar and actions.

Writing a Simple Bison Grammar File

To write a Bison grammar file, first define the valid input tokens. Listing 1-7-4 shows the definition section for a Bison grammar file:

Listing 1-7-4: Definition Section of a Grammar File

```
%{
#include <stdio.h>
#include <math.h>
#define YYSTYPE float
%}
%token NUMBER
%left '-', '+'
%left '*', '/'
```

In this listing, the %token directive defines the NUMBER token. The declaration of this NUMBER token is present in the header file, xxx.tab.h, which is generated by the parser. A Flex-generated lexical analyzer returns this token whenever it encounters a number. The %left directive indicates the associativity of the operators, which indicates how the operands of the operator are evaluated.

Note An operator can have either left-to-right or right-to-left associativity. For example, if an operator has left-to-right associativity, it evaluates the expression in the operand to its left and then the operand to its right. In right-to-left associativity, the evaluation begins at the right and moves to the left.

The rule section of the grammar file contains the syntax description of the expression, as shown in Listing 1-7-5:

Listing 1-7-5: Rules Section of a Grammar File

```
%%
expression_list: expression '\n'
               | expression_list expression '\n'  { printf("value = %f\n",$2); }
                  ;
expression: expression '+' expression  { $$ = $1 + $3; }
              | expression '-' expression { $$ = $1 - $3;}
              | expression '*' expression {$$ = $1*$3;}
              | expression '/' expression { if($3 == 0.0)
                        yyerror("Divide by Zero");
                        else
                                $$ = $1/$3;
                        }
        | NUMBER      {$$ = $1;}
        ;
```

In this listing, the rules are written in context-free grammar. Each rule has two sides - the left-hand side and the right-hand side. The left and right-hand sides of the rule are written before and after a colon (:) symbol, respectively. Each rule is expressed in terms of terminal and non-terminal symbols.

Each symbol defined in the grammar file has an associated value. The value of a symbol can vary from the string value to the numeric value of the variable. For example, the value of the terminal symbol, NUMBER, is a floating-point number. You can access the value of this symbol using $x, where x can be $, 1, 2 or 3. $$ is the value of the non-terminal symbol on the left hand side of the rule. $1 and $2 are the values of the first and second symbols on the right hand side of the rule. For example, you can write $1 to access the value of the symbol, NUMBER, in the grammar file in Listing 1-7-5.

The following code is the final section of the grammar file that describes the user-defined subroutines:

```
%%
void main(int argc, char* argv[])
{
yyparse();
}
```

Designing a Compiler for a New Programming Language

To design a compiler for a new programming language, determine the programming language constructs the compiler will need to support such as the while loop. The sample compiler compiles a new programming language that resembles C and supports simple expressions, variable names, function calls, the while loop, and comments.

Designing the Basic Framework of a New Programming Language

The new programming language and its lexical analyzer provide support to evaluate simple expressions. Listing 1-7-6 shows a Flex file, simcom.l, for the lexical analyzer part of the new programming language:

Listing 1-7-6: Flex File simcom.l

```
%{
#include  "simcom.tab.h"
#include "math.h"
extern int lineno;
%}
%%
[\t]          ;  /*whitespaces ignored*/
[0-9]+\.?|[0-9]*\.?[0-9]+  {  yylval.dval = atof(yytext);return NUMBER;   }
"="                       { return('='); }
"-"                       { return ('-'); }
"+"                       { return('+'); }
"*"                       { return('*'); }
"/"                       { return('/'); }
[\n] { lineno++;return '\n'';}
.                         {;/* ignore bad characters */ }
%%
int yywrap()
{
return 1;
}
```

This listing shows the lexical analyzer for the new programming language. When this Flex-generated lexical analyzer encounters \n, it increments the line number and returns the \n token to the parser. This lexical analyzer returns the NUMBER token whenever it encounters a number. The global variable, yylval, stores the value of the token. This variable is defined in the Bison-generated parser. Flex does not contain any definition of the main() function. The main() function is defined in the user-defined subroutine section of the grammar file.

The grammar file declares the token, NUMBER. The Flex file also includes the Bison-generated simcom.tab.h file. This file contains the declaration of all the tokens the parser supports.

Note To learn more about parsers, see the *Developing Compilers Using C* ReferencePoint.

Listing 1-7-7 shows how to create a parser for the new programming language:

Listing 1-7-7: A Sample Parser

```
%{
#include <stdio.h>
#include <math.h>
#define YYSTYPE float
%}
%token NUMBER
%left '-', '+'
%left '*', '/'
%%
expression_list: expression '\n'
               | expression_list expression '\n'  { printf("value = %f\n",$2); }
                    ;
expression: expression '+' expression  { $$ = $1 + $3; }
              | expression '-' expression { $$ = $1 - $3;}
              | expression '*' expression {$$ = $1*$3;}
              | expression '/' expression { $$ = $1/$3;
                        }
        | NUMBER      {$$ = $1;}
;
%%
int main( int argc, char** argv ){
        ++argv;--argc;
        yyin = fopen( *argv, "r" );
        while( ! feof( yyin ) ){
                yyparse();
        }
        return 0;
}
```

This listing shows how to create a basic expression parser. This code does not support any error-checking or recovery features. The %token defines the token of the type NUMBER. %left indicates that the minus and plus signs have left associativity. $$ is the value of the non-terminal symbol on the left of the colon (:) symbol.

Adding Support for Variable Names in the New Programming Language

To add support for variable names in the new programming language, you need to add the symbol table to the parser. A symbol table is a structure that stores the variable names encountered in the program. The structure of the symbol table can vary from a simple array to complex structures such as hashes. Listing 1-7-8 shows the structure of the symbol table:

Listing 1-7-8: Structure of Symbol Table

```
#define MAXSYMB 50
typedef struct  sym
{
char *symname;
double value;
}symtab;
```

To add a symbol table to the above code, you also need to define a function to add and delete entries from the symbol table. The symtab.h file contains both the structure definition of the symbol table and the function declaration. The symtab.c file contains the definition of symbol table manipulation functions.

Listing 1-7-9 shows the contents of the symtab.h file:

Listing 1-7-9: Contents of symtab.h

```
#ifndef _SYM_HEAD_
#define _SYM_HEAD_
#define MAXSYMB 50
typedef struct  sym
{
char *symname;
double value;
}symtab;
symtab * syminsert( char * symbol);
symtab * symlook(char * symbol);
#endif
```

This listing shows the contents of the symtab.h file, which contains the structure definition and function declaration of the symbol table.

The symtab.c file defines the functions to manipulate the symbol table, as shown in Listing 1-7-10:

Listing 1-7-10: Contents of symtab.c

```
#include <stdio.h>
#include <math.h>
#include  "symtab.h"
symtab * symbolstore[MAXSYM];
symtab *  symlook (char * symbol)
{
        int count;
        symtab * sympointer;
        for (count = 0 ;count <MAXSYM;count++)
        {
                if !strcmp(symbolstore[count]->symname, symbol)
                return &symbolstore[count];
        }
        }
        symtab * syminsert ( char * symbol)
        {
                int count;
                for (count = 0 ;count <MAXSYM;count++)
                {
                if (!symbolstore[count]->symname)
                symbolstore[count]->symname = strdup(symbol);
                }
        }
}
```

This listing shows the contents of the symtab.c file, which contains the definitions of the syminsert and symlook functions.

Listing 1-7-11 shows the modified Flex file to add variable support to the new programming language:

Listing 1-7-11: Modified simcom.l

```
%{
#include  "simcom.tab.h"
#include "math.h"
#include  "symtab.h"
extern int lineno;
%}
[\t];  *whitespaces ignored*/
[0-9]+\.?|[0-9]*\.?[0-9]+  {  yylval.val = atof(yytext);return NUMBER;   }
"\n"  { lineno++;return '\n';}
[a-zA-Z]a-zA-Z0-9]*        {
                           symtab  * s;
s = symlook(yytext)
if(s == NULL)
                           s = syminsert(yytext);
yylval.sym = s;
return VARIABLE;
}
"="                        { return('='); }
"-"                        { return ('-'); }
"+"                        { return('+'); }
"*"                        { return('*'); }
"/"                        { return('/'); }
"("                        { return ('('); }
")"                        { return(')'); }
%%
int yywrap()
{
return 1;
}
```

This listing shows the modified Flex file. Whenever the Flex-generated lexical analyzer finds the identifier, it searches for the identifier in the symbol table. If the lexical analyzer does not find the identifier in the symbol table, the lexical analyzer adds it to the symbol table.

Listing 1-7-12 shows the modified grammar file to add variable support in the new programming language:

Listing 1-7-12: Modified simcom.y

```
%{
#include <stdio.h>
#include <string.h>
#include <symtab.h>
%}
%union {
double val;
symtab *sym;
}
%token <sym> VARIABLE
%token <val >NUMBER;
% left '-', '+'
%left '*'.'/'
%  left UMINUS
% type<val> expression
%%
statement_list: statement ';'
                statement_list statement ';'
statement: VARIABLE '=' expression { $1->value = $3;}
          | expression { }
expression: expression '+' expression  { $$ = $1 + $3; }
              | expression '-' expression { $$ = $1 - $3;}
              | expression '*' expression {$$ = $1*$3;}
              | expression '/' expression {
if($3 == 0.0)
                                yyerror ("divide by zero");
else
$$ = $1/$3;
                        }
        | NUMBER      {$$ = $1;}
        | '-' expression %prec UMINUS {$$ = -$2;}
             |  '(' expression ')'  { $$ = $2;}
             | NUMBER
             | VARIABLE   {$$ = $1->value;}
;
```

This listing shows the modifications made in the grammar file to add variable support.

Adding Support for Comments in the New Programming Language

The new programming language provides support for comments similar to those in the C programming language. To add support in the compiler for comments similar to C, add the following line of code to the user-defined rule subsection of the Flex file:

```
"/*"{ comment(); }
```

Listing 1-7-13 shows the code for the comment() function:

Listing 1-7-13: comment() Function

```
void comment(){
        char c, c1;
loop:
        while ((c = input()) != '*' && c != 0)
                putchar(c);
        if ((c1 = input()) != '/' && c != 0){
                unput(c1);
                goto loop;
        }
        if (c != 0)
                putchar(c1);
}
```

Adding Support for Complex Language Constructs in the New Programming Language

You can also add support to the new programming language for complex language constructs such as relational operators and if loops. Listing 1-7-14 shows the code for the modified Flex file to add support for the if, while, and for loops in the new programming language:

Listing 1-7-14: Modified Flex File with Support for if, while, and for Loops

```
#include "simcom.tab.h"
#include "math.h"
#include "symtab.h"
extern int lineno;
%}
[\t];  *whitespaces ignored*/
[0-9]+\.?|[0-9]*\.?[0-9]+  {  yylval.val = atof(yytext);return NUMBER;   }
"\n"  { lineno++;return '\n';}
[a-zA-Z]a-zA-Z0-9]* {
symtab  * s;
s = symlook(yytext)
if(s == NULL)
s = syminsert(yytext);
yylval.sym = s;
return VARIABLE;
}
"=" { return('='); }
"-" { return ('-'); }
"+" { return('+'); }
"*" { return('*'); }
"/" { return('/'); }
[i][f]{ return (IF); }
[e][l][s][e]{ return (ELSE); }
[w][h][i][l][e]{ return (WHILE); }
[f][o][r]{ return (FOR); }
"("  { return ('('); }
")" { return(')'); }
"{" {return('{'); }
"}" {return('}'); }
%%
int yywrap()
{
return 1;
}
```

This listing returns three tokens, IF, WHILE, and FOR, in addition to the tokens generated by the code given in Listing 1-7-9.

Listing 1-7-15 shows the modified Bison file to add support for the if, while, and for loops:

Listing 1-7-15: Modified Bison File

```
%{
#include <stdio.h>
#include <string.h>
#include <symtab.h>
%}
%union {
double val;
symtab *sym;
int intval;
}
%token <sym> VARIABLE
%token <val>NUMBER
%token <intval> FOR,IF,WHILE
% left '-', '+'
%left '*'.'/'
%  left UMINUS
% type<val> expression
```

```
%%
statement_list: statement '\n'
                statement_list statement '\n'
statement: VARIABLE '=' expression { $1->value = $3;}
            |   WHILE '(' expression ')' statement
{ }
            |        IF '(' expression ')'  statement
                {}
            |        IF '(' expression ')' statement ELSE statement
                {}
          | FOR '(' expression ';' expression ';' expression ')' statement
        {}
   | '{ statement_list '}'
 | expression { }
statement_list: statement_list '\n'
                  | statement_list statement
                ;
expression: expression '+' expression  { $$ = $1 + $3; }
              | expression '-' expression { $$ = $1 - $3;}
              | expression '*' expression {$$ = $1*$3;}
              | expression '/' expression {
if($3 == 0.0)
                                yyerror ("divide by zero");
else
$$ = $1/$3;
                        }
        | NUMBER      {$$ = $1;}
        | '-' expression %prec UMINUS {$$ = -$2;}
             |  '(' expression ')'  { $$ = $2;}
             | NUMBER
             | VARIABLE   {$$ = $1->value;}
```

This listing shows the grammar to parse the if, while, and for statements, without adding any action for the if, while, and for loops.

Related Topics

For related information on this topic, you can refer to:

- Introducing Compiler Design
- Principles of Compiler Design
- Introducing Optimization in Compiler Design
- Error Detection and Recovery