Compiler Design ReferencePoint Suite

Table of Contents

Point 6: Understanding YACC

Ruchi Srivastava

A compiler converts a program written in high-level languages, such as C, Java, and Pascal, to a low-level language, such as assembly language. A compiler consists of a lexical analyzer and a parser. The lexical analyzer scans the input file for specific patterns and returns tokens that match the pattern. The parser accepts input from the lexical analyzer and translates the input into the target language, which can be a high-level language such as C or a low-level language such as the assembly language.

Yet Another Compiler Compiler (YACC) is a standard parser generation tool for the Unix operating system. YACC generates Look Ahead LR (1) (LALR (1)) parsers based on a grammar written in Backus Naur Form (BNF). YACC generates the code for the parser in the C language.

LALR parsers are a type of LR parser that you can use for context-free grammars. LR parsers read the input from left to right and produce the rightmost derivation for the input. LALR (1) parsers can understand several types of context-free grammars.

This ReferencePoint describes the various phases of a compiler. It explains the features of YACC, the input that YACC accepts to generate the target language, and how you can use YACC to build parsers. It also explains how to create an application that generates a parser using YACC.

Phases of a Compiler

There are four phases of a compiler:

- Lexical analysis: Scans the code and generates tokens, which are a sequence of characters associated with a regular expression in the code.
- Syntactic analysis or the parsing: Uses the tokens generated by the lexical analyzer to parse the input code according to the grammar specified for the language.
- Semantic analysis: Checks the code for semantic errors.
- Code generation: Converts the intermediate code generated after parsing and semantic analysis to assembly language.

Figure 1-6-1 shows the various phases of a compiler:

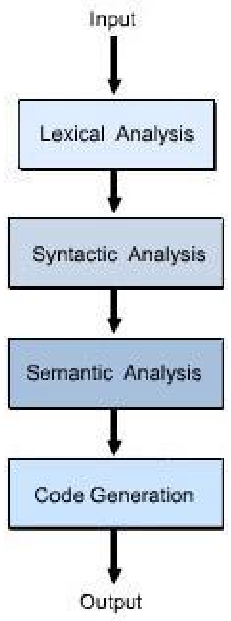


Figure 1-6-1: Phases of a Compiler

The Lexical Analyzer

The lexical analyzer scans the input from left to right and generates tokens. A token is a variable that is assigned to each regular expression in the code. The tokens generated by the lexical analyzer are used by the next phase of the compiler for syntactic analysis. Figure 1-6-2 shows the interaction between the lexical analyzer and the syntactic analyzer:

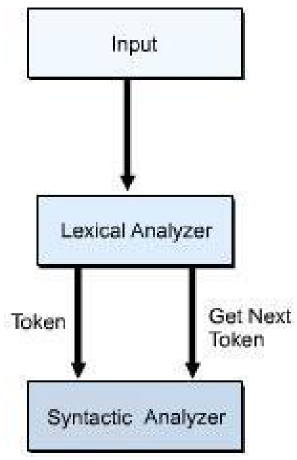


Figure 1-6-2: Interaction Between the Lexical Analyzer and the Syntactic Analyzer

There are different tokens defined for different programming languages. For example, the tokens defined in the C language include keywords and identifiers. Table 1-6-1 describes various tokens and the corresponding regular expressions in C:

Table 1-6-1: Tokens and Regular Expressions

Token	Regular Expression	Description
Char	[A-Z a-z]	Any character from A to Z or a to z.
Number	[0-9]+	One or more occurrences of digits ranging from 0 through 9.
Word	[char]+	One or more occurrences of characters.
Blank	" "	A blank space.

Note To learn more about lexical analysis, see the Understanding Lex ReferencePoint.

The Syntactic Analyzer or the Parser

Each language has a grammar associated with it. A grammar is a set of production rules. Every grammar has a root, which is the first non-terminal symbol in the first production rule of the grammar. The parser or syntactic analyzer parses the tokens and checks whether or not the input conforms to the grammar you specify.

Note In a programming language, the statements are terminated with a comma and semi-colon. These are called terminal symbols. All the other symbols are called non-terminal symbols.

The production rules consist of three parts:

- Left Hand Side (LHS): Consists of a single non-terminal symbol, which can be expanded.
- Colon: Separates the LHS and RHS of the production rule.
- Right Hand Side (RHS): Consists of non-terminal and terminal symbols or tokens. Terminal symbols cannot be expanded. When the LHS is a single symbol, the RHS symbols are ORed using the | symbol. The terminal symbols also include operators such as *, =, and +.

An example of a grammar with three production rules is:

```
numerical: first|second
first: VALUE '+' VALUE
second: VALUE '-' VALUE
```

In the above example, the production rules consist of the non-terminal symbols, numerical, first, and second. The terminal symbols are VALUE, +, and -. VALUE is the token generated by the lexical analyzer. An OR operation is performed between the symbols, first and second, because they have the same LHS, numerical.

You need to always use a non-terminal symbol or token in LHS. You cannot use a terminal symbol in LHS.

All the operators used in the grammar are assigned precedence levels. An operator with a high precedence level is grouped with the operands first before the operator with a low precedence level is grouped. For example, if the * operator is assigned a higher precedence level than the + operator, g*h+f is evaluated as (g*h)+f.

In an expression, if there are operators with the same precedence level, the associativity rule decides how the operands are to be grouped. There are three types of associativity that can be assigned to an operator:

- Left associativity: If an operator is left associative, the operands to the left of the expression are grouped first before grouping the operands to the right of the expression. For example, if the operator + is left associative, then the expression g+h+j is grouped as (g+h)+j.
- Right associativity: If an operator is right associative, the operands to the right of the expression are grouped first before grouping the operands to the left of the expression. For example, if the operator + is right associative, then the expression g+h+j is grouped as

g+(h+j).

- No associativity: If an operator is not assigned any associativity, the operands cannot be grouped. For example, if the operator > is not assigned any associativity, the expression g>h>j cannot be grouped.

Note To learn more about parsing techniques, see Parsing Techniques ReferencePoint.

Overview of YACC

YACC is a tool that generates LALR(1) parsers used in syntactic analysis. The number 1 in LALR(1) indicates that the parser can look ahead only one token at a time. The YACC generated parser scans the input from left to right. The rightmost non-terminal symbol is expanded. If the non-terminal does not satisfy any production rule, the symbol is pushed to a stack. This action is called shift. The RHS is expanded until it satisfies a production rule. Then, the RHS symbols are removed from the stack and replaced by the LHS of the production rule. The LHS of the rule is then pushed to the stack. This action is called reduction.

The process of expansion and replacement is repeated until the root is reached, which signifies that the input is valid and is in accordance with the grammar that you have specified. For example, the specified grammar may be:

```
numerical: RESULT=first
first: VALUE '+' VALUE| VALUE '-' VALUE
```

In the first line of the above code, the non-terminal symbols are numerical and first. In the second line of the above code, the terminal symbols are VALUE, +, and -. The lexical analyzer generates the terminal symbols, VALUE and RESULT.

If the input is abc = 78-45, the first action taken by the LALR parser is shift. The terminal symbols are pushed to a stack until a rule is satisfied, as given in the code below:

```
abc
abc=
abc=78
abc=78-
abc=78-45
```

In the above code, the RHS satisfies the second production rule, that is, first: VALUE '-' VALUE.

The second action taken by the parser is reduction. The RHS is removed from the stack, which implies that 78, –, and 45 are removed from the stack and replaced by the LHS, abc = first. The LHS is then pushed to the stack. As a result, abc = first matches the RHS of the first production rule, numerical: RESULT = first. The RHS symbols, abc, =, and first, are replaced with 78, -, and 45. The parser reaches the root, numerical, which indicates that the input is validated. Figure 1-6-3 shows the parse tree generated by the LALR(1) parser:

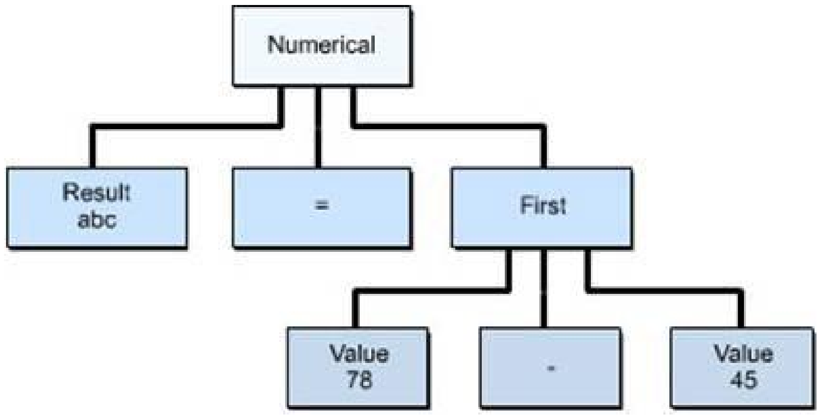


Figure 1-6-3: The Parse Tree
Sometimes, there may be more than one rule that matches the set of collected tokens. This condition is called the reduce/reduce conflict. The YACC generated parser cannot parse the grammar that generates the reduce/reduce conflict. Sometimes, the grammar may also have a shift/reduce conflict that occurs when the tokens completely match one rule while partially matching another rule. In this case, the YACC generated parser has two options:

- Shifting the token into the stack
- Reducing the rule

The YACC generated parser can modify the grammar to remove the shift/reduce conflict by shifting the token in the stack. Figure 1-6-4 shows the working of a YACC generated parser:

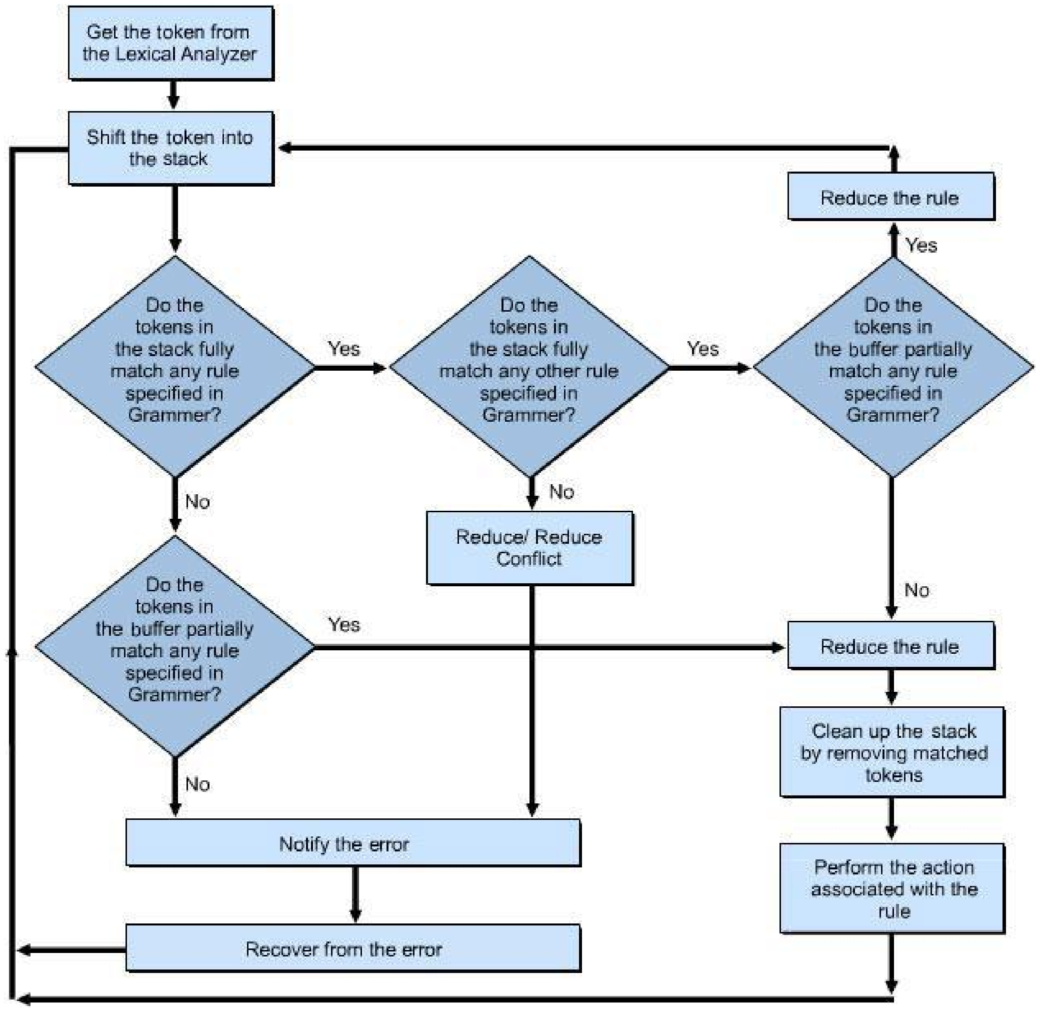

Get the token from the Lexical Analyzer
Shift the token into the stack
Reduce the rule
Yes
Do the tokens in the stack fully match any rule specified in Grammer?
Yes
Do the tokens in the stack fully match any other rule specified in Grammer?
Yes
Do the tokens in the buffer partially match any rule specified in Grammer?
No
No
Reduce/ Reduce Conflict
No
Do the tokens in the buffer partially match any rule specified in Grammer?
Yes
Reduce the rule
Clean up the stack by removing matched tokens
No
Notify the error
Perform the action associated with the rule
Recover from the error

Figure 1-6-4: Working of a YACC Generated Parser

YACC Input

YACC input is the input that the YACC tool parses. It consists of three parts:

- The definition section
- The rules section
- The user subroutine section

The Definition Section

The definition section constitutes the first part of the YACC grammar and is an optional section. This section is copied to the parser generated by YACC and consists of the following parts:

- %{C declarations and include files %}: Contains C declarations and header files.
- %token: Contains the declaration of the tokens generated by the lexical analyzer and used by YACC. The syntax of the declaration is:

```
%token TOKEN_NAME1,TOKEN_NAME2
```

In the above code, TOKEN_NAME1 and TOKEN_NAME2 are the tokens generated by the lexical analyzer.

- %union: Contains the definitions for symbols such as tokens and non-terminal symbols used in the LHS of the grammar. The syntax of the declaration is:

```
%union {
           int hel;
           double ert;
       }
```

- %start: Contains the name of the grammar rule from where the parser will start parsing. The syntax of the declaration is:

```
%start numerical
```

In the above code, numerical is the name of the rule from where the parsing is to be started.

Note If the first rule is the start rule, you need not specify the %start declaration.

- %type: Contains the declaration of the non-terminal symbols used in the grammar. The data type assigned to a non-terminal is declared in the union declaration. The syntax of the declaration is:

```
%type <data_type> name1, name2
```

In the above code, name1 and name2 are the names of the non-terminals used in the grammar.

- %left: Assigns left associativity to operators. The syntax of the declaration is:

```
%left + -
```

The operators + and – are assigned left associativity.

- %right: Assigns right associativity to operators. The syntax of the declaration is:

```
%right + -
```

The operators + and – are assigned right associativity.

- %noassoc: Assigns no associativity to operators. The syntax of the declaration is:

  ```
  %noassoc >
  ```

  In the above code, > is not assigned any associativity

Listing 1-6-1 shows the definition section:

Listing 1-6-1: The Definition Section

```
%{
#include<stdio.h>
%}
%union{
           int hel;
           double ert;
      }
%token <ert> VALUE
%token <hel> RESULT
%left + -
%right * /
%noassoc >
%type<hel> first
```

The above code shows the definition section that contains the declaration of the header file, stdio.h, and the declaration of the two tokens, VALUE and RESULT, generated by the lexical analyzer. The code shows the use of %type, %union, %left, %right, and %noassoc declarations.

The Rules Section

The rules section is enclosed between the symbols, %% %%. This section contains user-specified rules, which consist of the grammar and the associated actions. The actions can be assigning a value to a variable or executing a C statement such as printf().The actions are written inside braces { }.YACC allows actions to be written in the middle of the rule.

The parser generated by YACC uses the user-specified grammar that consists of the LHS and RHS separated by a colon. A semicolon terminates the rule. You can use $$ as a variable to set a value that is to be returned by the action part of the rule. You can use $1, $2... to retrieve values of the components from left to right in the RHS of the grammar. An example of user-specified grammar is:

```
first : FT{ $$=89}
```

The action part of the grammar returns 89 to the parser.

```
second: VALUE NAME RESULT
```

$1 contains the value returned by VALUE

$2 contains the value returned by NAME

$3 contains the value returned by RESULT

The following code shows the rules section:

```
%%
//rule consisting of grammar and action
first: second AND third  {printf("This is done\n");};
second: SET;
third : RESET;
%%
```

The above code shows the rules section containing the grammar and the action. A semicolon terminates the rules. The grammar consists of three production rules. The non-terminal symbols are first, second, and third. The terminal symbols generated by the lexical analyzer are AND, SET, and RESET. If you provide the input, SET AND RESET, then the parser validates the input according to this grammar and prints this input on the screen.

The following code shows the rules section that uses the variable $$ and $1, $2:

```
%%
  second: NAME {$$=79} VALUE RESULT {printf("the value is %d",$2);};
%%
```

The above code prints the value 79 on the screen. In the above code:

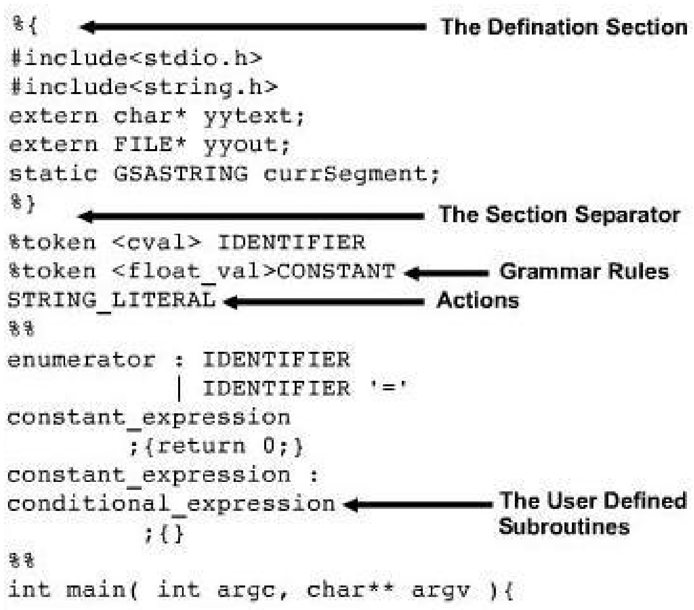
- $1: Contains the value returned by NAME.
- $2: Contains the action part {$$=79} that returns 79.
- $3: Contains the value returned by VALUE.
- $4: Contains the value returned by RESULT.
- $5: Contains the value returned by the action {printf ("the value is %d",$2);}.

The User Subroutine Section

The user subroutine section contains the main() function and subroutines such as yyparse() and yyerror(). An example of the main() function is:

```
main()
{
        yyparse();
}
```

Figure 1-6-5 shows the various sections of a YACC 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
         ;{}
%%
int main( int argc, char** argv ){
```

Figure 1-6-5: Sections of a YACC Grammar File

YACC Routines and YACC Macros

The YACC library contains YACC routines to perform tasks such as checking for errors in the input, reporting the syntax errors, and parsing the input. You can write YACC macros to handle situations such as calling a function to enter an error recovery phase, terminating the parsing phase, and discarding the lookahead token if the token is read.

YACC Routines

The YACC library contains different routines such as main() and yyerror().Table 1-6-2 describes the various routines provided by the YACC library:

Table 1-6-2: YACC Routines

YACC Routine	Description
main()	Checks for errors, opens files, and accepts command line arguments.
yyerror()	Reports syntax errors.
yyparse()	Parses an input.
yyrecovering()	Returns a non-zero value if the parser is in the recovery mode, else returns zero. If the parser is in recovery mode, errors are not reported.

YACC Macros

Table 1-6-3 describes the various YACC macros used in the action part of the grammar that you specify:

Table 1-6-3: YACC Macros

YACC Macros	Description
yyerror()	Calls the yyerror() routine to enter the error recovery mode when a syntax error is detected.
yyerrork()	Informs the parser to come back to the normal mode from the error recovery mode. The parser reports errors in the normal mode.
yyaccept()	Makes the yyparse() routine return a zero value indicating success.
yyabort()	Makes the yyparse() routine return a non-zero value indicating failure. This function is called when you need to stop parsing.
yyclearin()	Discards a lookahead token if the token is already read.

Limitations of YACC

YACC cannot parse two types of grammars:

- Ambiguous grammars
- Grammars that require more than one token of lookahead

Ambiguous Grammars

Ambiguous grammars contain the same non-terminal symbol in the RHS of a production rule more than once. YACC encounters shift/reduce conflicts due to ambiguous grammar. This implies that YACC cannot decide whether or not to reduce a rule or shift a token to the stack. More than one parse tree is generated for the same input. An example of ambiguous grammar is:

```
numerical: numerical '+' numerical
                 | numerical '*' numerical
                 | numerical '-' numerical
                | VALUE;
```

This grammar is ambiguous because it contains the same non-terminal symbol, numerical, more than once in the RHS of the production rule. If the input is 5+8-3, there will be more than one parse tree generated for the same input using the ambiguous grammar.

Figure 1-6-6 shows the parse tree generated for the ambiguous grammar:

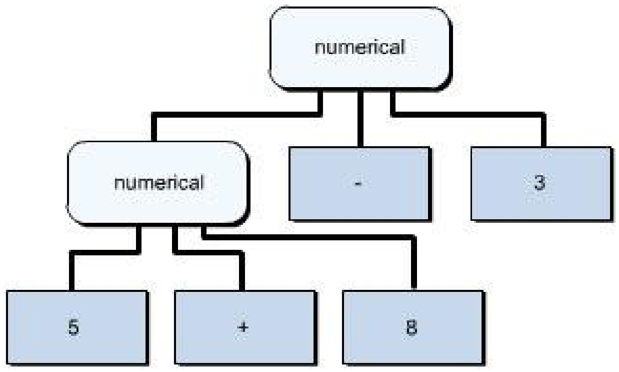


Figure 1-6-6: First Parse Tree for the Ambiguous Grammar
Figure 1-6-7 shows the second parse tree generated for the same grammar:

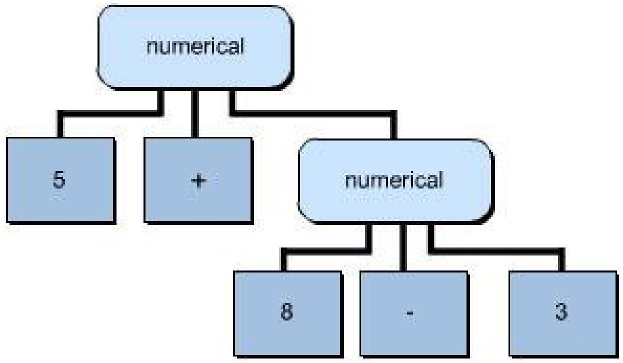


Figure 1-6-7: Second Parse Tree for the Ambiguous Grammar

Grammars That Require More Than One Token of Lookahead

YACC cannot parse the input when the specified grammar requires more than one token of lookahead. An example of a grammar requiring more than one token of lookahead is:

```
final: first HEL RESET
           | second HEL SET
first : RET| TER
second: RUT|TUR
```

The above code shows a grammar that a YACC parser cannot use to parse an input. The grammar is not ambiguous, but it requires two tokens of lookahead, HEL and RESET or HEL and SET. If the input is RET HEL RESET, the parser cannot parse the input because it requires looking ahead by more than one token, HEL and RESET, to validate the input.

Handling Conflicts

The easiest way to handle ambiguous grammars and the conflict in grammars that require more than one token of lookahead is to change the grammar for your language from time to time. Sometimes, you may be using a grammar, which was previously defined and which needs to be modified for the present application. You can simplify the ambiguity by making minor modifications to the grammar for your language. Other methods you can use to reduce conflicts are changing the location of the keywords in your language and incorporating common ambiguous grammar syntax in your grammar definition.

Using YACC

You can download YACC from http://www.tuxfinder.com/packages/?defaultname=byacc. To install YACC on Unix/Linux:

1. In the command prompt, change the active directory to the directory containing the source code of YACC.

2. Type the following command to configure YACC:

   ```
   /configure'
   ```

3. Type the following command to compile the package:

   ```
   make install
   ```

To use the YACC tool:

1. Save the file with the Lex specification as file_name.l
2. Save the file with the YACC grammar as file_name.y
3. Compile and link both the C files generated by the Lex tool and YACC tool and link them with the libraries.
4. Provide the input to be parsed.

Listing 1-6-2 shows how YACC is used to generate a parser that parses the input according to the grammar specified in the rules section of the YACC input:

Listing 1-6-2: YACC Input

```
//file_name.y
//definition section containing a header file
%{
#include<stdio.h>
%}
//tokens AND, SET and RESET as generated by the //lexical analyzer
%token  AND,SET.RESET
%%
//rules section consisting of grammar and action
first: second AND third  {printf("This is done\n");};
second: SET;
third : RESET;
%%
//user subroutine section containing the //main()function
main()
{
yyparse();
}
```

The above code shows the YACC input that is provided to the YACC tool. The definition section contains a header file, stdio.h. The rules section consists of the grammar and the associated actions. The user subroutine section consists of the main() function that calls the YACC routine, yyparse(). The Lex tool generates the tokens used by the YACC tool according to the regular expressions specified in the rules section of the Lex specification.

Listing 1-6-3 shows the Lex specification that is provided as input to the Lex tool in order to generate tokens:

Listing 1-6-3: Lex Specification

```
//file_name.l
//definition section containing the header file y.tab.h generated by the YACC tool
%{
#include "y.tab.h"
%}
//rules section containing the regular expressions
%%
[and]+ {return AND;}
[set]+ {return SET;}
[qwr]+ {return RESET;}
%%
```

The above code shows the Lex specification provided as input to the Lex tool. The Lex specification consists of two parts, definition section and rules section. The definition section in the code consists of the header file, y.tab.h. The rules section in the code consists of regular expressions and their associated actions. The tokens generated by Lex are AND, SET, and RESET. The YACC tool uses these tokens to parse the input.

Sample Applications to Generate a Parser Using YACC

You can create applications that use the YACC tool to generate parsers. For example, you can create a calculator application that generates a parser to parse the input according to the grammar specified in the YACC specification. The application also evaluates mathematical expressions containing operators such as *, ?, %, +, and -. You can also create a sentence recognizer application that generates a parser, which recognizes the sentence as plain or complex according to the grammar specified in the rules section of the YACC specification. Plain sentences refer to the simple English sentences used for regular purposes. Complex sentences may contain expressions and conditions in a sentence.

The Calculator Application

Listing 1-6-4 shows the YACC input provided to the YACC tool for the calculator application:

Listing 1-6-4: YACC Specification

```
//definition section containing a header file and declarations
%{
        #include <stdio.h>
        int start;
        int asdf[30];
%}

%start list
//tokens returned by the lexical analyzer
%token QWERTY ALPHA
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS
%%
//rules section containing grammar and associated actions
list:          |
        list stat '\n'
         |
        list error '\n'
         {
           yyerrok;
         }
         ;
stat:    math
         {
           printf("%d\n",$1);
         }
         |
         ALPHA '=' math
         {
           asdf[$1] = $3;
         }
         ;
math:    '(' math ')'
         {
           $$ = $2;
         }
         |
         math '*' math
         {
           $$ = $1 * $3;
         }
         |
         math '/' math
         {
           $$ = $1 / $3;
         }
         |
         math '%' math
         {
           $$ = $1 % $3;
         }
         |
         math '+' math
         {
           $$ = $1 + $3;
         }
          |
         math '-' math
         {
           $$ = $1 - $3;
         }
         |
         math '&' math
         {
           $$ = $1 & $3;
         }
         |
         math '|' math
         {
           $$ = $1 | $3;
         }
         |
        '-' math %prec UMINUS
         {
           $$ = -$2;
         }
         |
         ALPHA
         {
           $$ = asdf[$1];
         }
         |
         number
         ;
number:   QWERTY
         {
           $$ = $1;
           start = ($1==0) ? 8 : 10;
         }        |
         number QWERTY
         {
           $$ = start * $1 + $2;
         }
         ;
%%
main()
{
         return(yyparse());
}
yyerror(s)
char *m;
{
          fprintf(stderr, "%s\n",m);
}
yywrap()
{
          return(1);
}
```

The above code shows the YACC input. The definition section in the code contains a header file and declarations. The rules section contains the grammar and the associated actions. The grammar consists of rules that evaluate mathematical expressions involving operators such as *, ?, %, +, and -. The YACC specification uses the tokens QWERTY and APLHA returned by the Lexical analyzer. The user subroutine section in the code contains the main() function that calls the YACC routines, yyparse(), yywrap(), and yyerror().

Listing 1-6-5 shows the Lex specification provided as input to the Lex tool:

Listing 1-6-5: Lex Specification

```
//definition section containing header files and declarations
%{

        #include <stdio.h>
        #include "y.tab.h"
        int k;
        extern int yylval;
%}
//rules section containing regular expressions and their associated actions
%%
 [0-9]      {
             k = yytext[0];
             yylval =  - '0';
            return(QWERTY);
           }
[^a-z0-9\b]     {
                  k = yytext[0];
                  return(k);
}
[a-z]      {
             k = yytext[0];
             yylval = k - 'a';
             return(ALPHA);
           }
```

The above code shows the Lex specifications. The definition section in the above code contains declarations and two header files, stdio.h and y.tab.h. The parser generates the y.tab.h header file. The rules section in the code contains regular expressions and their associated actions. The rules section returns two tokens, QWERTY and ALPHA, which are used by the parser.

The Sentence Recognizer Application

Listing 1-6-6 shows the YACC input provided to the YACC tool:

Listing 1-6-6: YACC Specification

```
//definition section containing a header file, stdio.h
%{
        #include <stdio.h>
%}
//tokens generated by the Lex tool
%token NAME YOU CURRENT CONNECTION ADJ
//rules section consisting of the grammar and their
//associated actions that recognize whether or not
//a sentence is plain or complex
%%
first: second   { printf("the sentence is plain.\n"); }
        | third { printf("the sentence is complex.\n"); }
        ;

second: vbnm curr fghg
        |       vbnm curr fghg phrase
        ;

third: second CONNECTION second
        |       third CONNECTION second
        ;

vbnm:   NAME
```

```
        |       YOU
        |       ADJ vbnm
        ;

curr:           CURRENT
        |       curr CURRENT
        ;

fghg:           NAME
        |       ADJ fghg
        ;

phrase: NAME
        ;

%%
//user subroutine section containing the main()function
extern FILE *yyin;

main()
{
        while(!feof(yyin)) {
                yyparse();
        }
}

yyerror(s)
char *m;
{
    fprintf(stderr, "%s\n", m);
}
```

The definition section in the above code contains a header file. The Lex tool generates the tokens, NAME, YOU, CURRENT, CONNECTION, and ADJ, which are used by the parser. The rules section in the code consists of the grammar and the associated actions. The grammar finds out whether or not the sentence is plain or complex based on the rule, which is specified as a sample statement in the program. The user subroutine section consists of the main() function that calls the YACC routines, yyparse() and yyerror(). Listing 1-6-7 shows the Lex specification provided as input to the Lex tool:

Listing 1-6-7: Lex Specification

```
%{
        #include "y.tab.h"      /* token codes from the parser */
        #define FIND 0 /* default - not a defined word type. */
        int condition;
%}

%%
\n      { condition = FIND; }
\.\n    {       condition = FIND;
                return 0;
        }
adjective{condition= ADJ;}
name{condition=NAME;}
pronoun{condition=YOU;}
conn{condition=CONNECTION;}
curr{condition=CURRENT;}
[a-zA-Z]+ {
             if(condition != FIND)
          {
                new_sent(condition, yytext);
             }
          else
          {
                switch(find_word(yytext))
                {
                case CURRENT:
                  return(CURRENT);
                case ADJ:
                  return(ADJ);
                case NAME:
                  return(NAME);
                case YOU:
                  return(YOU);
                case CONNECTION:
                  return(CONNECTION);
                default:
                  printf("%s:  nothing defined\n", yytext);
                }
        }
      }
.       ;

%%
//user subroutine section
// the structure word defines a linked list consisting of words
struct word
{
        char *word_name;
        int word_type;
        struct word *next;
};
// starting variable in the linked list of words
struct word *word_list;
extern void *malloc();
// adds a new word to the linked list
int new_sent(int type, char *word)
{
        struct word *pointer;
        if(lookup_word(word) != FIND)
    {
                printf("word %s exists \n", word);
                return 0;
    }
        // Word does not exist
               //add a new word to the linked list
        pointer = (struct word *) malloc(sizeof(struct word));
        pointer->next = word_list;
        pointer->word_name = (char *) malloc(strlen(word)+1);
        strcpy(pointer->word_name, word);
        pointer->word_type = type;
        word_list = pointer;
        return 1;       }
//searches for a word in the linked list
int find_word(char *word)
{
        struct word *pointer = word_list;
//finding the word in the list
        for(; pointer; pointer = pointer->next) {
                if(strcmp(pointer->word_name, word) == 0)
                        return pointer->word_type;
}

        return FIND;
}
```

The above code shows the Lex specifications provided as input to the Lex tool. The definition section in the code consists of declarations and a header file, y.yab.h. The rules section in the code consists of regular expressions and their associated actions. The tokens generated in the rules section are ADJ, NAME, YOU, CONNECTION, and CURRENT. The user subroutine section consists of a structure word and two functions, new_sent() and find_word(). The structure word defines a linked list consisting of words. The find_word function searches for a word in the linked list of words and the new_sent() function adds a new word to the linked list.

Related Topics

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

- *Writing Compilers Using C*
- Compiler Construction using Flex and Bison
- Parsing Techniques
- Understanding Lex