Compiler using Lex and Yacc

Introduction

A compiler is a **program that converts code in a programming language** (which is high-level) **into a machine-understandable format** (low-level). This compiler then generates an executable program that can be used to parse and execute input files in a particular language.

Creation of Compiler:

Part 1: Creating the Lexical Analyser

Part 2: Adding the Grammar Rules

Part 3: Creating the Symbol Table

Part 4: Adding the Syntax Tree

Part 5: Performing Semantic Analysis

Part 6: Intermediate Code Generation

What is Lex?

Lex is a tool used to create a lexical analyzer. So what is lexical analysis? It is the **process in which a stream of characters is converted into a sequence of tokens**. Such programs are called lexers or tokenizers. The file contains a set of regular expressions along with actions associated with each of them. The output is a table-driven scanner — which tells us what to do when we see a particular input character based on the state we are in. This output is saved in a file called lex.yy.c. All Lex files have a structure similar to the one given below.

```
{declarations}
%%
{rules}
%%
{subroutines}
```

The **declarations** are of two types, in C, and in Lex. All imports and global declarations are done in C and enclosed within `%{` and `%}`. In addition to this, the Lex file can contain definitions of regular expressions and symbols.

The **rules** consist of patterns followed by the actions in the same line.

Finally, the **subroutines** contain our own functions that we would like to write.

Yacc

Yacc (Yet Another Compiler Compiler) is a tool used to create a parser. It parses the stream of tokens from the Lex file and performs the semantic analysis. Yacc translates a given Context-Free Grammar (CFG) specifications into a C implementation y.tab.c. This C program when compiled, yields an executable parser. A Yacc file is in many ways, similar to the Lex file.

```
{declarations}
%%
{rules}
%%
{subroutines}
```

The **declarations** and **subroutines** are the same as those in Lex, but the rules are slightly different. Here, the **rules** are not regular expressions, rather they are grammar definitions in CFG. These rules, like in Lex, have two parts — productions and actions.

There is another file that is created — y.tab.h which we haven’t talked about yet. This is a file created when we compile the Yacc file. It tells our Lex file about all the valid token declarations that are defined in our Yacc program.

constructing our own C compiler.

What are the capabilities of our compiler?

Before we start coding, ***our compiler will accept***:

1. valid C statement such as declarations, initializations etc.
2. `if-else` statements

3. `for` loops
4. nested `for` and `if-else` statements

Implementation of front-end phase of the compiler:

1. generating the symbol table
2. creating the parse tree
3. performing semantic analysis
4. generating the intermediate code

Part 1: Creating the Lexical Analyzer

The Lex file, as we know by now has 3 parts.

1. Declarations

Let us first declare all the imports and global variables necessary for our compiler.

```
%{
     #include "y.tab.h"
     int countn=0;       /* for keeping track of the line number */
%}
```

We include `y.tab.h` and a counter to keep track of the line number we are on.

Regular definitions.

```
%option yylinenoalpha          [a-zA-Z]
digit          [0-9]
unary          "++"|"--"
```

`%option yylineno` creates a scanner that stores the line number.

2. Rules

This is one of the most important parts of our Lex file. We define all the rules necessary to tokenize the input stream we get from the C program. These tokens will then be used by our Yacc file.

First, let us define rules for our keywords. Here, if we come across `printf` or `scanf`, our lexer will identify it and return the token `PRINTFF` and `SCANFF` respectively.

```
"printf"    { strcpy(yylval.nd_obj.name,(yytext)); return PRINTFF;
}
"scanf"     { strcpy(yylval.nd_obj.name,(yytext)); return SCANFF; }
```

We similarly describe rules for other keywords such as `int`, `float`, `for`, `if`, `else` etc. In order to understand the other rules, being familiar with regex syntax is essential.

The above code is our ***final Lex program***.

3. Subroutines

Our subroutine is very simple, has only one function called `yywrap()`.

```
int yywrap() {
    return 1;
}
```

At this point, there may be some confusion, what are all these `yy` functions — `yywrap, yylval, yytext` etc.?

Compile our Lex file by using the following command

```
lex lexer.l
```

Part 2: Adding the Grammar Rules

In this section, our objective is to create our Yacc file with all the grammar rules essential for parsing the input C program.

Coding our YACC Program

As discussed earlier, the Yacc program consists of three sections. We will go through each part sequentially.

1. Declarations

In our declaration, we import the necessary header files and function declarations that will be used in the Yacc program.

```
%{
    #include<stdio.h>
    #include<string.h>
    #include<stdlib.h>
    #include<ctype.h>
    #include"lex.yy.c"

    void yyerror(const char *s);
    int yylex();
    int yywrap();
%}%token VOID CHARACTER PRINTFF SCANFF INT FLOAT CHAR FOR IF ELSE
```

```
TRUE FALSE NUMBER FLOAT_NUM ID LE GE EQ NE GT LT AND OR STR ADD
MULTIPLY DIVIDE SUBTRACT UNARY INCLUDE RETURN
```

The header files are straight from C, so we will skip over them. We have 3 functions that are defined — `yyerror(const char *s)`, `yylex()` and `yywrap()`. The latter 2 have been discussed in Part 1, coming to `yyerror()`, it is a library function that displays an error message.

Moving on, we see `%token` followed by a bunch of words. These words are actually tokens that the YACC file can accept. This is stored in `y.tab.h` which is used by the Lex program.

2. Rules

With our definitions done, it is time to get started with the rules of our parser. Before we begin, let us understand how rules are defined and there basic structure.

```
production-name: definition 1    { action }
| definition 2                   { more action }
|                                { some more action }
;
```

After `definition 2`, the next production we see is a null production. The actions corresponding to each definition are optional. They are executed on the basis of the definition which is satisfied for each production.

Let us take a simple declaration statement in C.

```
int x = 35;orfloat x;
```

We now define how the grammar will look above such declarative statements.

```
declaration: datatype ID '=' value
| datatype ID
;datatype: INT
| FLOAT
| CHAR
;value: NUMBER
| FLOAT_NUM
| CHARACTER
;
```

We will now systematically write all the grammar definitions that will be used to create our parser.

The first grammar will be for the whole C program structure. Each C program has a set of header files, the declaration of the `main` function, followed by the body and `return` statement. Our compiler assumes that there is only one function in our C code — the `main` function.

```
program: headers main '(' ')' '{' body return '}'
;
```

Next, we talk about the headers. `INCLUDE` is a token defined in our Lex program. The reason we have `headers headers` is to accommodate for multiple header files.

```
headers: headers headers
| INCLUDE
;
```

The `main` is another production which is defined below. We assume that no arguments are passed to this `main` function.

```
main: datatype ID
;
```

As you can see, datatype is not amongst the tokens defined above, but is another production.

```
datatype: INT
| FLOAT
| CHAR
| VOID
;
```

After the main, comes the body of our C code. The body can have a multitude of possible statements and loops. They are defined in the rules as follows.

```
body: FOR '(' statement ';' condition ';' statement ')' '{' body
'}'
| IF '(' condition ')' '{' body '}' else
| statement ';'
| body body
| PRINTFF '(' STR ')' ';'
| SCANFF '(' STR ',' '&' ID ')' ';'
;
```

Upon closer examination, we see that the code block within the `for` loop and `if` statement contains `body` once again. This allows our compiler to accept **nested statements** of varying complexities.

The `if-else` declaration contains the `else` production at the end. This allows us to have `if-else` as well as simple `if` statements. We define `else` as:

```
else: ELSE '{' body '}'
|
;
```

There are two more productions that are a part of the `body` that need to be discussed — `condition` and `statement`.

```
condition: value relop value
| TRUE
| FALSE
;statement: datatype ID init
| ID '=' expression
| ID relop expression
| ID UNARY
| UNARY ID;
```

Now that we know the production rules of `condition` and `statement`, we can proceed to look at the other productions being used in them.

`value` can be an integer, decimal value, character or a variable.

```
value: NUMBER
| FLOAT_NUM
| CHARACTER
| ID
;
```

`relop` is a production that defines all the possible relational operations that can be performed.

```
relop: LT
| GT
| LE
| GE
| EQ
| NE
;
```

Coming to `statement`, we can see that we have initializations, declarations as well as assignment operations. While declaring a variable in C, it is not necessary to initialize it to a specific value. That is why the `init` production is a **nullable** production.

```
init: '=' value
|
;
```

Moving on, we have `expression`.
```
expression: expression arithmetic expression
| value
;
```

These expressions can have arithmetic operations, so we have defined another production for the same. The arithmetic operations are defined as, but not limited to addition, subtraction, multiplication and division.
```
arithmetic: ADD
| SUBTRACT
| MULTIPLY
| DIVIDE
;
```

With all these productions completed, we are done with the `body` of our C program. The final part to the C code is the `return` statement. We define `return` as follows.
```
return: RETURN value ';'
|
;
```

`return` is also a nullable production.

With all these definitions done, we are done with the second part of our Yacc file. We can now move onto the final segment — subroutines.

3. Subroutines

Our subroutines consists of two functions — `main` and `yyerror`. They are defined as follows.

```
int main() {
    yyparse();
}void yyerror(const char* msg) {
    fprintf(stderr, "%s\n", msg);
}
```

The `main` function tells us to parse the input file while `yyerror` prints the errors that occur when we compile and execute our Yacc file.

The complete integrated code of all the productions we defined in the rules section, declarations and subroutines is given below.

To compile our Yacc program, we run the following command:

```
yacc -v -d parser1.y
```

To generate the executable, we run:

```
gcc -ll y.tab.c
```

To test our compiler, we execute the command given below. (Assuming you have a C program called `input1.c`.

```
./a.out<input1.c
```

With that, we are done with our Yacc file. The complete integrated code of all the productions we defined in the rules section, declarations and subroutines is given below.

At the moment, no actions have been defined. As a result, if the program is a valid C code, it will not show any `syntax error` while executing.

Part 3: Creating the Symbol Table

In this part, our aim is to construct the symbol table and store header files, variables, keywords and constants along with details such as line number, the type and data type. They will be discussed in further detail in this article. Our goal is to achieve the following symbol table.

The symbol table stores the identifier, the datatype (applicable to variables only), the type or category of the identifier and the line number.

With that said, let’s get started with adding the symbol table functionality to our previous code.

Adding the Symbol Table

In order to add our symbol table, we must first define the structure and details. Our symbol table will have details like the name of the symbol, the data type, the type of symbol (keyword, constant, variable etc.) and the line number.

```
struct dataType {
        char * id_name;
        char * data_type;
        char * type;
        int line_no;
} symbol_table[40];
```

NOTE: The line number proved to be a huge help in all the stages of the compiler as it would tell us which line the error was in whenever the program crashed.

Here, the `40` indicates the maximum number of entries in our symbol table, this can be increased or decreased as per our requirements.

The next part that needs to be addressed is the `insert_type()` function. This function is called whenever a function or variable is added to the symbol table. It copies the data type of the variable or function to be added to the character array called `type`.

```
void insert_type() {
    strcpy(type, yytext);
}
```

Now, we come to the main part of the symbol table feature — adding the symbols to our table. We use the `add` function to achieve this purpose.

```
void add(char c) {
  q=search(yytext);
  if(!q) {
    if(c == 'H') {
      symbol_table[count].id_name=strdup(yytext);
      symbol_table[count].data_type=strdup(type);
      symbol_table[count].line_no=countn;
      symbol_table[count].type=strdup("Header");
      count++;
    }
    else if(c == 'K') {
      symbol_table[count].id_name=strdup(yytext);
      symbol_table[count].data_type=strdup("N/A");
      symbol_table[count].line_no=countn;
      symbol_table[count].type=strdup("Keyword\t");
      count++;
    }  else if(c == 'V') {
      symbol_table[count].id_name=strdup(yytext);
      symbol_table[count].data_type=strdup(type);
      symbol_table[count].line_no=countn;
      symbol_table[count].type=strdup("Variable");
      count++;
    }  else if(c == 'C') {
      symbol_table[count].id_name=strdup(yytext);
      symbol_table[count].data_type=strdup("CONST");
      symbol_table[count].line_no=countn;
      symbol_table[count].type=strdup("Constant");
      count++;
    }  else if(c == 'F') {
      symbol_table[count].id_name=strdup(yytext);
      symbol_table[count].data_type=strdup(type);
      symbol_table[count].line_no=countn;
      symbol_table[count].type=strdup("Function");
      count++;
    }
}
```

In order to ensure that we do not have repeated occurrences of the same symbol in our table, we use the`search` function. It is defined as:

```
int search(char *type) {
    int i;
    for(i=count-1; i>=0; i--) {
        if(strcmp(symbol_table[i].id_name, type)==0) {
            return -1;
            break;
        }
    }
    return 0;
}
```

Once the functions are done and ready, all we have to do is start inserting the function calls at the relevant places. We have 5 types of symbols:

1. H — Headers: for all the header files
2. K — Keywords: for keywords like `for`, `if`, `else` etc.
3. V — Variables: called only during variable declarations
4. C — Constants: any assignment such as `9`, `'A'`, `-3.14` etc.

5. F — Functions: for now, only `main`

After adding the function calls at the right place, all we need to do is print the table in the `main` function of the Yacc file. The function calls to `add` and the updated `main` function can be seen below.

All we need to do now is compile and run our compiler. The commands are:

```
yacc -v -d parser2.y
lex lexer.l
gcc -ll y.tab.c
./a.out<input1.c
```

In order to get the output shown in Fig. 1, the corresponding C code is as follows.

We are finally done with creating our symbol table.

Part 4: Adding the Syntax Tree

Our parse tree will be a binary tree — it'll have two children. The root of the tree will be the start of the program. The structure is extremely simple and intuitive.

```
                      program
                         |
   __________________________________________
  |                                          |
headers                                   main
                                             |
                               __________________________
                              |                          |
                             body                      return
```

The body will consists of multiple statements, loops and if-else blocks. In order to construct a tree like this, we need to first create nodes. Once that is done, we write a function to add the elements to the tree and finally print the tree.

1. Node for Our Tree

Like any node for a binary tree, it'll have a left and right child along with the data about the node, here it is called `token`.

```
struct node {
  struct node *left;
  struct node *right;
  char *token;
};
```

We also need to redefine the types of the tokens and productions.

We call this new type `nd_obj`. It is defined as follows.

```
struct var_name {
   char name[100];
   struct node* nd;
} nd_obj;
```

In the later stages of our compiler, we will be adding different node types. They will be called `nd_obj2`, `nd_obj3` and so on.

In our definitions, we make a small modification by adding the type.

```
%token VOID%token <nd_obj> CHARACTER PRINTFF SCANFF INT FLOAT CHAR
FOR IF ELSE TRUE FALSE NUMBER FLOAT_NUM ID LE GE EQ NE GT LT AND OR
STR ADD MULTIPLY DIVIDE SUBTRACT UNARY INCLUDE RETURN

%type <nd_obj> headers main body return datatype expression
statement init value arithmetic relop program
```

At this stage, we must modify our Lex file as well. All the tokens are now of time `nd_obj` so we can now save the name of the token.

The updated Lex file is shown below.

As seen in Part 1 of this series, `yylval` is the value associated with the token. Using `yytext`, we assign the `name` to our tokens.

2. Making the Nodes and Adding them to the Parse Tree

We define the origin of the tree as `head`. It is of type `node`. This will serve as the entry point to access our parse tree. We define this in the first segment of the YACC file.

```
struct node *head;
```

The next step is to create a function that will add the nodes to the tree. This will be called `mknode()`. This function will take 3 arguments — the left child, right child and name of the node.

```
struct node* mknode(struct node *left, struct node *right, char
*token) {
  struct node *newnode = (struct node*) malloc(sizeof(struct
node));
  char *newstr = (char*) malloc(strlen(token)+1);
  strcpy(newstr, token);
  newnode->left = left;
  newnode->right = right;
  newnode->token = newstr;
  return(newnode);
}
```

As you can see, this function will help us make our tree. The next step is to call this function and pass the necessary parameters.

Before we get started with this part, it is imperative to understand how we access the productions and the elements in the grammar. Assume we have the following production:

```
print: PRINTFF { add('K'); } '(' STR ')' ';'
```

To access `PRINTFF` we use `$1`. To access `STR` we use `$4`. The elements are indexed from 1 and any actions that occur in between the `{}` count as one element. That is why `STR` is `$4` and not `$3`. We will be using this to call the `mknode()` function and pass the productions as children if necessary. In order to access the production ie. `print` we use `$$.nd`.

We start generating our tree from the `program` production. We will then move in order based on the productions in `parser2.y` which we had made in Part 3 of this series.

```
program: headers main '(' ')' '{' body return '}' {
    $2.nd = mknode($6.nd, $7.nd, "main");
    $$.nd = mknode($1.nd, $2.nd, "program");
    head = $$.nd;
}
;
```

Here, we assign `main` to have children `body` and `return` while the node is called “main”. `program` is called “program” and it’s children are `header` and `main`. Since this will be the entry point of our parse tree, we assign `program` to `head`.

We next move on to the `header` node.

```
headers: headers headers {
    $$.nd = mknode($1.nd, $2.nd, "headers"); }
| INCLUDE {
    add('H');
    $$.nd = mknode(NULL, NULL, $1.name);
}
;
```

In case of multiple headers, we create a new node having children `header` once again. If there is a single `header` we save the name as that of the header file and having no children.

We now move onto `body`. We skip `main` because it has been taken care of in the `program` production.

```
body: FOR { add('K'); } '(' statement ';' condition ';' statement
')' '{' body '}' {
    struct node *temp = mknode($6.nd, $8.nd, "CONDITION");
    struct node *temp2 = mknode($4.nd, temp, "CONDITION");
    $$.nd = mknode(temp2, $11.nd, $1.name);
}
| IF { add('K'); } '(' condition ')' '{' body '}' else {
    struct node *iff = mknode($4.nd, $8.nd, $1.name);
    $$.nd = mknode(iff, $11.nd, "if-else");
}
| statement ';' { $$.nd = $1.nd; }
| body body { $$.nd = mknode($1.nd, $2.nd, "statements"); }
| PRINTFF { add('K'); } '(' STR ')' ';' { $$.nd = mknode(NULL,
NULL, "printf"); }
| SCANFF { add('K'); } '(' STR ',' '&' ID ')' ';' {
    $$.nd = mknode(NULL, NULL, "scanf");
}
;
```

The `for` loop has a tricky structure, which will be explained below.

```
for
                          |
                  ________________________________________
                 |                                        |
             condition                                  body
                 |
     __________________________
    |                          |
   declaration              condition
                               |
                      ___________________________
```

```
                        |                            |
                      check                       iterator
```

That is why we have two temporary nodes in our code — to handle the check condition and the iteration while the other temporary node handles the declaration of the iterator as well as the above temporary node.

```
condition: value relop value {
    $$.nd = mknode($1.nd, $3.nd, $2.name);
}
| TRUE { add('K'); $$.nd = NULL; }
| FALSE { add('K'); $$.nd = NULL; }
| { $$.nd = NULL; };
```

The `if-else` block is implemented similarly, feel free to generate a tree like above to understand how it works.

```
else: ELSE { add('K'); } '{' body '}' {
    $$.nd = mknode(NULL, $4.nd, $1.name);
}
| { $$.nd = NULL; };
```

To keep things simple, we do not store the string that the `printf` functions have in our parse tree, for now they are leaf nodes. Feel free to add them to your code!

We next take a look at `statement`.

```
statement: datatype ID { add('V'); } init {
    $2.nd = mknode(NULL, NULL, $2.name);
    $$.nd = mknode($2.nd, $4.nd, "declaration");
}
| ID '=' expression {
    $1.nd = mknode(NULL, NULL, $1.name);
    $$.nd = mknode($1.nd, $3.nd, "=");
}
| ID relop expression {
    $1.nd = mknode(NULL, NULL, $1.name);
    $$.nd = mknode($1.nd, $3.nd, $2.name);
}
| ID UNARY {
    $1.nd = mknode(NULL, NULL, $1.name);
    $2.nd = mknode(NULL, NULL, $2.name);
    $$.nd = mknode($1.nd, $2.nd, "ITERATOR");
}
| UNARY ID {
    $1.nd = mknode(NULL, NULL, $1.name);
    $2.nd = mknode(NULL, NULL, $2.name);
    $$.nd = mknode($1.nd, $2.nd, "ITERATOR");
}
;
```

The logic for creating the nodes has already been covered, so we can just write down the codes directly.

Taking a look at `init`.

```
init: '=' value { $$.nd = $2.nd; }
| { $$.nd = mknode(NULL, NULL, "NULL"); }
;
```

We now take a look at `expression`.

```
expression: expression arithmetic expression {
    $$.nd = mknode($1.nd, $3.nd, $2.name);
}
| value { $$.nd = $1.nd; }
;
```

`value` is another production which will be added to the tree. However, it will be a leaf node always.

```
value: NUMBER { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| FLOAT_NUM { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| CHARACTER { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| ID { $$.nd = mknode(NULL, NULL, $1.name); }
;
```

We finally reach the last production that needs to be included in our parse tree, `return`.

```
return: RETURN { add('K'); } value ';' {
    $1.nd = mknode(NULL, NULL, "return");
    $$.nd = mknode($1.nd, $3.nd, "RETURN");
}
| { $$.nd = NULL; }
;
```

Our parse tree is finally constructed. All we need to do is an inorder traversal of the tree.

3. Inorder Traversal of the Abstract Syntax Tree

All we have to is an traverse the tree in an inorder fashion. The code for inorder traversal is as follows.

```
void printInorder(struct node *tree) {
    int i;
    if (tree->left) {
        printInorder(tree->left);
    }
    printf("%s, ", tree->token);
    if (tree->right) {
        printInorder(tree->right);
    }
}
```

The complete code is shown below.

All we need to do now is compile and run our compiler. The commands are:

```
yacc -v -d parser3.y
lex lexer.l
gcc -ll y.tab.c
./a.out<input1.c
```

We have successfully performed syntax analysis as well and generated the abstract syntax tree through our compiler.