# Creating a Simple C Compiler

Zi Lin (1500014129)

June 1, 2017

## 1 Problem Statement

This is project of course 02032730, Principles of Compilers, in which I create a simple C compiler with lex and yacc, but without semantic analyzer. In additional, I also make a program to visualize the parse tree, you can use it with my python script.

The project is fairly limited in functions - it does not support array, structure, union, file, set, switch statement, do statement and bit operation, etc. It only permits *int* and $float$ as data structures. And it will ignore the blank, tab and line break in your source code. However, it is able to completely compile in terms of lexical analysis and language syntax.

The part lexical analysis parts are done in *.lex* file and the essential parts of a compiler are integrated in *.yacc* file. The compilation steps are:

```
$ lex calculator.l //lexical analysis
$ yacc calculator.y //parser
$ parsing_tree.py //tree plotting
```

You can use the test code in *parsing.txt* to check the compiling process. And I will also represent some of my codes as well as necessary statement in next chapters.

## 2 Lexical Analysis & Parsing

### 2.1 Lex - Defining Terminals

In *calculator.l* we should define all the potential tokens in our source codes using regular expressions. The variable *number* here is initialized to 0 to mark the id of the token in order to record them when in parsing. In $printf$, the output of $'@'$ is to mark that it is the left part of the grammar, and the output of $'\$'$ is to show that it is a terminal.

Here are some examples:

```
//Defining integer
[0-9]+{
        yylval = number;
```

```
        number++;
        fprintf(yyout,"%dNUM@␣%d$%s\n",yylval, yylval, yytext);
        return NUM;
}

//Defining identifier
[a-zA-Z][a-zA-Z0-9]*
{
        yylval = number;
        number ++;
        fprintf(yyout,"%dID@␣%d$%s\n",yylval, yylval, yytext);
        return ID;
}

//Defining type
int{ yylval = number; number ++; return INT;}

//Defining while
while {yylval = number; number ++; return WHILE;}

//Defining compare
\=\={
        yylval = number;
        number ++;
        fprintf(yyout,"%dCOMPARE@␣%d$%s\n",yylval, yylval, yytext);
        return COMPARE;
}
```

### 2.2 Yacc - Input Grammar Rules

In calculator.y we should input all the potential grammar rules. These are the grammar given by the statement of assignment:

$Function \rightarrow Type\ identifier\ (\ ArgList\ )\ CompoundStmt$

$ArgList \rightarrow Arg \mid ArgList\ ,\ Arg \mid \epsilon$

$Declaration \rightarrow Type\ IdentList;$

$Type \rightarrow int \mid float$

$IdentList \rightarrow identifier\ ,\ IdentList \mid identifier$

$Stmt \rightarrow ForStmt \mid WhileStmt \mid Expr\ ; \mid IfStmt \mid CompoundStmt \mid Declaration \mid \epsilon$

$ForStmt \rightarrow for\ (\ Expr\ ;\ OptExpr\ ;\ OptExpr\ )\ Stmt$

$OptExpr \rightarrow Expr \mid \epsilon$

$WhileStmt \rightarrow while\ (\ Expr\ )\ Stmt$

$IfStmt \rightarrow if\ (\ Expr\ )\ Stmt\ ElsePart$

$ElsePart \rightarrow else\ Stmt \mid \epsilon$

$CompoundStmt \rightarrow StmtList$

$StmtList \rightarrow StmtList\ Stmt \mid \epsilon$

$Expr \rightarrow identifier\ =\ Expr \mid Rvalue$

$Compare \rightarrow == \mid < \mid > \mid <= \mid >= \mid !=$

$Mag \rightarrow Mag\ +\ Term \mid Mag\ -\ Term \mid Term$

$Term \rightarrow Term\ *\ Factor \mid Term\ /\ Factor \mid Factor$

$Factor \rightarrow (\ Expr\ ) \mid -\ Factor \mid +\ Factor \mid identifier \mid number$

First of all, we should make the definition of all the tokens and operator precedence.

%token NUM ID COMPARE INT FLOAT WHILE FOR IF ELSE

%left '+' '−'

%left '∗' '/'

Here are some samples of my codes. Notice that the variable *number* here is different from *number* in file *.lex*, which is initialized to 500 in order to distinguish it from the token. Thus the serial number will not be confusing. The value of the initialization depends on how many tokens in your source code. Since different terminals and non-terminals have their own serial numbers, and in the output we just put the number before them, it is clear where the non-terminal comes from and where it is reduced. The reduction procedure is printed to the file *parsing_process.txt*.

```
Function:
        Type ID '(' ArgList ')' CompoundStmt
        {
                $$ = num_y;
                fprintf
                (yyout,
                "%dFunction@␣%dType␣%dID␣%d$(␣%dArgList␣%d$)␣%dCompoundStmt\n",
                num_y, $1, $2,$3, $4, $5, $6);++num_y;
        };
```

```
Type:
        INT
        {
                $$ = num_y;
                fprintf(yyout,"%dType@␣%d$int\n", num_y, $1); ++num_y;
        }
        | FLOAT
        {
                $$ = num_y;
                fprintf(yyout,"%dType@␣%d$float\n", num_y, $1);++num_y;
        };

Factor:
        '(' Expr ')'
        {
                $$ = num_y;
                fprintf(yyout,"%dFactor@␣%d$(␣%dExpre␣%d$)\n",num_y,$1, $2, $3);
                ++num_y;
        }
        |'-' Factor
        {
                $$ = num_y;
                fprintf(yyout,"%dFactor@␣%d$-␣%dFactor\n",num_y,$1, $2);
                ++num_y;
        }
        |'+' Factor
        {
                $$ = num_y;
                fprintf(yyout,"%dFactor@␣%d$+␣%dFactor\n",num_y,$1, $2);++num_y;
        }
        |ID
        {
                $$ = num_y;
                fprintf(yyout,"%dFactor@␣%dID\n",num_y,$1);++num_y;
        }
        |NUM
        {
```

```
                $$ = num_y;
                fprintf(yyout,"%dFactor@␣%dNUM\n",num_y,$1);++num_y;
        };
```

In order to import source code from file *parsing.txt* and output the reduction procedure to file *parsing_process.txt*, we need to set the *yyin* and *yyout*. There is a risks of making mistakes in the filein and fileout, you should check it carefully.

```
//define it at the very beginning in .y
extern FILE* yyin;
extern FILE* yyout;

int main(void) {
        FILE* fp = fopen("parsing.txt","r");
        FILE* fileOut = fopen("parsing_process.txt","w");
        yyin = fp;
        yyout = fileOut;
        if(fp!=NULL && fileOut!=NULL){
                yyparse();
                //system("pause");
                printf("parsing␣success!");
        }
        else{
                printf("fail␣to␣open␣file!");
        }
        fclose(fp);
        fclose(fileOut);
}
```

### 2.3 Test - Output the procedure

Once you code the lex and yacc files correctly, you can rebuild the whole project to check the output of procedure(here in file *parsing_process.txt*).

I use the codes *fact.c* here to test my lex and yacc files. The source code is represented as follow:

```
int fact ()
{
  int n,f,i;
  n = 8;
```

```
  f = 1;
  for (i= 2; i<= n; i = i+1)
        f = f*i;
}
```

And the procedure is as follow(it is too long so I just post part of it). Notice that the numbers in the front represent the serial numbers, '@' represents the left part of the grammar and '$' is to show that it is a terminal.

500Type@ 0$int

1ID@ 1$fact

501ArgList@ 501$null

502StmtList@ 502$null

503Type@ 5$int

6ID@ 6$n

8ID@ 8$f

10ID@ 10$i

504IdentList@ 10ID

505IdentList@ 8ID 9$, 504Identlist

506IdentList@ 6ID 7$, 505Identlist

507Declarition@ 503Type 506IdentList 11$;

508Stmt@ 507Declarition

509StmtList@ 502StmtList 508Stmt

12ID@ 12$n

14NUM@ 14$8

.....

### 2.4 Test - Tree Plotting

In this section I will introduce the method of visualizing the parse tree using a python program with Graphviz. Graphviz is an open source graph visualization software for representing structural information as diagrams of abstract graphs and networks. I simply apply the package to my python script, which facilitates the creation and rendering of graph description in the DOT language of Graphviz drawing software from Python.

First of all, we should sort out the data of nodes and edges. As for node, you should input the IDs and names (the ID here is the serial number in lex and yacc). As for edge you should input the source ID and target ID. What we have to do is to extract information from the

output in section 2.3. The code is shown in *parsing_tree.py*. Remember that in the previous sections we use '@' to mark the left part, which means that it is the parent node of the tree After segmentation of the lines, the word without '@' is a child node, and in order to tell terminal from non-terminal, the IDs of terminals include '$'.

```
from graphviz import Digraph
import re #regular expression

#create a graph object
dot = Digraph(comment = "Parsing␣Tree")
#if you output dot, you will see "<graphviz.dot.Digraph object at 0x...>"
fileopen = open("parsing_process.txt")
for line in fileopen:
        wordlist = line.split()
        father = ''
        num1 = ''
        for word in wordlist:
    #add nodes and edges
                if word.find('@') != -1:   #find patent nodes
                        father = word.strip('@');
                        num = re.search('[0-9]+[$]*',father)
                        num1 = num.group(0)
                        dot.node(num1, father.replace(num1,''))
                else:      #find child nodes
                        child = word
                        num = re.search('[0-9]+[$]*',child)
                        num2 = num.group(0)
                        dot.node(num2, child.replace(num2,''))
                        dot.edge(num1, num2)
#save and render the source code, optionally view the result
dot.render('test-output/parse-tree.gv', view=True)
#you can check the source code by print(dot.source)
fileopen.close()
```

By the way, if you encounter the error as follow:

>>> RuntimeError: failed to execute['dot', '-Tpdf', '-0', '...'], make sure the Graphgviz executables are on your systems' path.

You should add path into your system variables. I run the program on Windows so it's

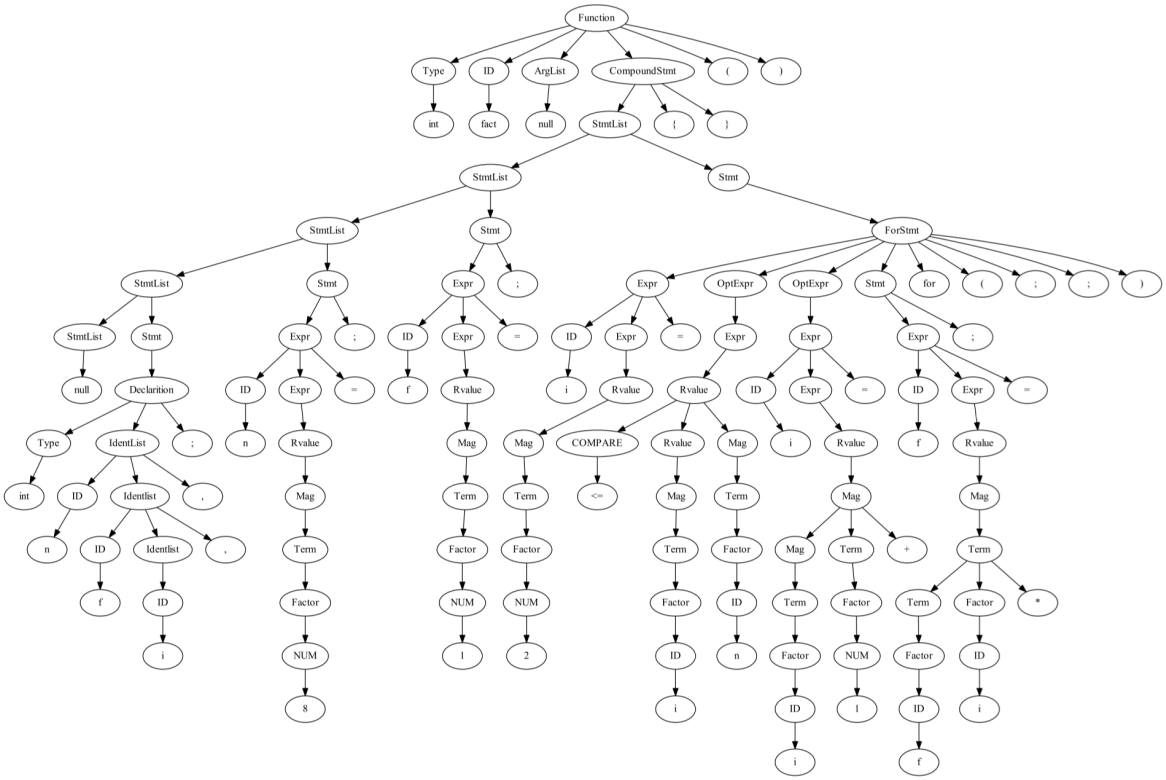
necessary to set it.

1. Add system environment variable, create variable $GRAPHVIZ\_DOT$ and set the value equals C:\Program Files (x86)\Graphviz-2.38\bin\dot.exe.

2. Set users' PATH environment, add variable $GRAPHVIZ\_INSTALL\_DIR$, set the value to be C:\Program Files (x86)\Graphviz-2.38.

3. Set system environment, add directory path *bin* (C:\Program Files (x86)\Graphviz-2.38\bin) to variable $PATH$

If you still have some problems, please turn to Google :) And then you can check the plot you create! Further more, you can try to set the order of the child nodes to promote the readability.




## 3 Project Reflection

I'd have to say, sometimes it feels quite annoyed when there are hundreds of errors shown in error list, especially when the error is in the generating code of lex and yacc - really hard to tell what goes wrong in those complicated codes. And it is useful to set some breakpoints or just run your program step by step (but still helps little when it comes to the generating program).

What I learn is, never try to rebuild the project after you have written a large number of codes, you just cannot figure out which part of it goes wrong, since the error could exist in any place. Even if the Visual Studio tells you that you have successfully rebuilt the project, chances

are that you can still be confronted with lots of weird bugs when running it. Just test it step by step, e.g. After writing the part of lex, you can try to input random digits or alphabets to check whether they can be recognized as numbers or identifiers. By the way, some times $system(\text{"}pause\text{"})$ helps.

In addition, pay attention to the $\epsilon$ and semicolons, it is very trivial but can make great difference. Also notice what can be output and what is missing, $printf$ helps you to position errors.

I made some mistakes when I try to import source code from file and output the result to file, because I have not programmed in C for long time (maybe one year...). And if you are not sure how to call a function or operation for some particular purposes, you'd better check the API.

Finding a proper package or software is a long process(maybe even longer than the time spent in realizing it). Here are some packages you can also try for making visualization - matplotlib, plotly, igraph etc. And you can also use *treeplot* in MATLAB (maybe should input the level of the tree so I did not choose it).

Finally, thanks for checking my project and reading my report. Please forgive me if I have any grammar mistakes. And I also welcome you to correct my program and give me some suggestions.