Compilers Project 1

INTRODUCTION

The project is focus on design a parser to parse .spl file into a parse tree. In this process, two steps are needed: one is to generate lexical analyzer, and another is to generate a syntax tree. Some little tricks are necessary to finish this project. As we haven't learn much on Flex and Bison, so a lot of time I'm trying to find manual.

The first part only takes about 3 hours to finish, and all the features implemented well. However, the second part takes me more than 20 hours, as there are so many difficulties in Bison, and the part B is much more complex than I thought.

The final result is that I realized all the basic and optional functions, but haven't implement bonus functions yet. But I found that the deadline of project is delayed, so I may have time to realize all the bonus part.

PART A: LEXICAL ANALYZER

Lexical analyzer is a simple Flex program to translate a source file to a simple list of tokens. Some tokens may have their names, such as token TYPE, ID, INT, FLOAT, CHAR, STRING, STRUCT, and so on. Other symbols are just simple token without name, so I can print them easily.

The realization is rather simple.

First of all, define all the regex rules to match tokens:

```
5   %}
6   NESTED_COMMENT \/\*([^\*]|(\*+[^\*\/]))*\/\*([^\*]|(\*+[^\*
7   COMMENT (\/\*([^*]|(\*+[^*\/]))*\*+\/)|(\/\/.*)
8   INT ([0-9]+)|(0x[0-9A-Fa-f]+)|(0[0-7]+)
9   FLOAT ([0-9]+\.[0-9]+)|([0-9]+(\.[0-9])?[fF])
10  CHAR ('.')|('\\x[0-9A-Fa-f]{2}')
11  STRUCT struct
12  STRING \"(\\.|[^\n"\\])*\"
13  TYPE (int)|(float)|(char)|(string)|(struct)
14  IF if
15  ELSE else
16  WHILE while
17  RETURN return
18  DOT \.
19  SEMI ;
20  COMMA ,
21  ASSIGN =
22  LT <
```

And some regex rules are omitted, which you can find in the tlex.l file.

By the way, there are two .l files in my source code, the tlex.l is a test file, which would only print the tokens with name out, and doesn't pass them to the Bison. Look README.md for more details (or use make test to generate test elf).

Back to the topic, the next step is to match regex to source file, some codes are shown here:

```
44  {NESTED_COMMENT} { printf("Error type A at Line %d: sy
45  {COMMENT} {}
46  {INT} { printf("INT %s\n", yytext); }
47  {FLOAT} { printf("FLOAT %s\n", yytext); }
48  {CHAR} { printf("CHAR %s\n", yytext); }
49  {STRUCT} { printf("STRUCT %s\n", yytext); }
50  {STRING} { printf("STRING %s\n", yytext); }
51  {TYPE} { printf("TYPE %s\n", yytext); }
52  {IF} { printf("IF\n"); }
53  {ELSE} { printf("ELSE\n"); }
54  {WHILE} { printf("WHILE\n"); }
55  {RETURN} { printf("RETURN\n"); }
56  {DOT} { printf("DOT\n"); }
57  {SEMI} { printf("SEMI\n"); }
58  {COMMA} { printf("COMMA\n"); }
59  {ASSIGN} { printf("ASSIGN\n"); }
60  {LT} { printf("LT\n"); }
61  {GT} { printf("GT\n"); }
62  {LE} { printf("LE\n"); }
63  {GE} { printf("GE\n"); }
64  {NE} { printf("NE\n"); }
65  {EQ} { printf("EQ\n"); }
```

I realized the nested multi-line comment check here, to match another /* */ pair in one /* */ pair. If nested multi-line comment is encounter, the program would exit immediately, and print out the line number of nested multi-line comment.

The whitespace check and unknown lexeme check are both here, as well.

Let's take a glance of the output generated by lexical analyzer:

```
Lexical Analysing file: test.spl
TYPE int
ID test
LP
TYPE int
ID a
COMMA
TYPE int
ID b
RP
LC
ID a
ASSIGN
INT 1
SEMI
RC
```

PART B: SYNTAX ANALYZER & SYNTAX TREE GENERATE

This part is the most difficult one, I have a lot of troubles when coding the Bison file, and the 26 pages project pdf isn't enough, so I have to read Flex/Bison manual to find other information.

I have read a lot of existing compilers to find design pattern, and finally find my own way: to generate tree nodes first, and then print it out. However, the code realization isn't beautiful, which is limited by my C programming skill.

All the reference compilers would be listed below, let's back to my design.

Main idea is to use CFG to generate syntax tree, and then print it out. I already know the second part is very easy, as a simple pre-order tree traversal. Therefore, the problem is the first part, this part is still too big, so I continue to divide it into several small tasks: tree structure design, new node function, union / token / type design, association order and priority, CFG without ambiguity and shift-reduce or reduce-reduce problem.

Yep, that's a lot of work to do. I firstly use a head file AST.h to put all the declarations here. AST.h includes tree structure, new node function, and print function. The design of tree structure is ugly, in which I used 7 child nodes to realize a 7-ary tree. However, when I finished my design, HHQ Huai told me that I can use a library called va list. I may use it in my next version, to realize a binary tree. Another little trick is to use extern errors variable, if the errors equals 0, then the tree would be printed, otherwise, nothing would happen after a series of error messages.

And then the AST.c file, I realized all the functions defined in AST.h. There is one thing to remember, non-terminal nodes should have their line number, so I keep a array of terminals, and check the type of token whether a terminal. If so, then skip the line number, simple print its name (or itself without name). A small thing to remember, the value in yytext would change, and I shouldn't use a char pointer, but copy the value in yytext, and store it in the tree node.

```
if (value != NULL) {
    char* buffer = (char*)malloc(sizeof(char)*strlen(yytext));
    strcpy(buffer, yytext);
    tree->value = buffer;
} else  tree->value = value;
```

The design of syntax.y, I defined a printerr function, so I won't use yyerror inside Bison. As in lex.l file, the return type are all nodes, so I declare all the tokens and types in syntax.y to be node type. The association rule and priority are as same to the regular C language.

By the way, for the bonus part, I have done them all. The hexadecimal integer match and the hexadecimal char match can be found in the regex rule in lex.l. And also does the comment and nested multi-line comment. Some screenshots are here:

```
/*something*/
int test(int a, int b) {
        a = 0x01;
}
```

Above is the source code, and then the output in syntax tree:

```
  Exp (3)
    INT: 0x01
SEMI
```

Some definitions are listed below:

```
16   %token <tr> INT FLOAT CHAR STRING STRUCT TYPE
17   %token <tr> IF ELSE WHILE
18   %token <tr> RETURN
19   %token <tr> ID
20   %token <tr> LT GT LE GE NE EQ PLUS MINUS MUL DIV AND OR N
21   %token <tr> DOT SEMI COMMA ASSIGN LP RP LB RB LC RC
22   %token <tr> ERR
23
24   %type <tr> Program ExtDefList ExtDef ExtDecList Specifier
25   %type <tr> VarList ParamDec CompSt StmtList Stmt Matched_
26   %type <tr> Def DecList Dec Exp Args;
27
28   %nonassoc LP RP LB RB LC RC
29   %nonassoc ELSE SEMI
30   %left DOT
31   %right ASSIGN
32   %left LT GT LE GE NE EQ
33   %left OR
34   %left AND
35   %left PLUS MINUS
36   %left MUL DIV
37   %right NOT
```

Then is the CFG definition, the basic idea of CFG definitions are the same as those in PDF. And two things I added.

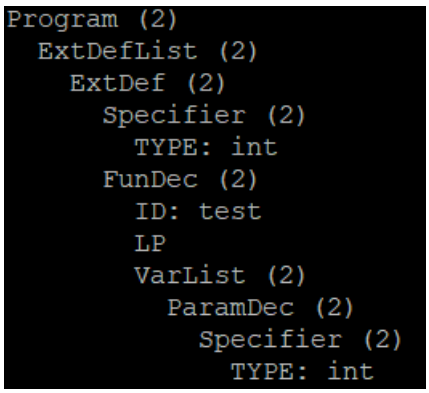
The first one is the redefinition of if statement, as I learned in lecture that simple if else statement may cause ambiguity, so I split Stmt into two parts: Matched_Stmt and Open_Stmt.

```
63   Stmt: Matched_Stmt { $$ = new_node("Stmt", NULL, $1->lineno, $1, NUL
64       | Open_Stmt { $$ = new_node("Stmt", NULL, $1->lineno, $1, NULL,
65   Matched_Stmt: IF LP Exp RP Matched_Stmt ELSE Matched_Stmt { $$ = new
66       | Exp SEMI { $$ = new_node("Matched_Stmt", NULL, $1->lineno, $1,
67       | CompSt { $$ = new_node("Matched_Stmt", NULL, $1->lineno, $1, N
68       | RETURN Exp SEMI { $$ = new_node("Matched_Stmt", NULL, $1->line
69       | WHILE LP Exp RP Stmt { $$ = new_node("Matched_Stmt", NULL, $1-
70       | Exp error { $$ = new_node("ERR", "semi error", $1->lineno, NUL
71       | RETURN Exp error { $$ = new_node("ERR", "semi error", $1->line
72       | IF LP Exp Matched_Stmt ELSE Matched_Stmt { $$ = new_node("ERR"
73       | WHILE LP Exp Stmt { $$ = new_node("ERR", "parenthesis error",
74   Open_Stmt: IF LP Exp RP Stmt { $$ = new_node("Open_Stmt", NULL, $1->
75       | IF LP Exp RP Matched_Stmt ELSE Open_Stmt { $$ = new_node("Open
76       | IF LP Exp Stmt { $$ = new_node("ERR", "parenthesis error", $1-
77       | IF LP Exp Matched_Stmt ELSE Open_Stmt { $$ = new_node("ERR", "
```

Another thing I worked on is to deal with type B error, so some error checks are added to the CFG, included semi check and parenthesis check.

Note that the line number in yylineno is the line number of finish match, to change it into start match, I use the line number of first token, and for terminals, the line number is defined in lex.l as the yylineno. So all the line number are start lines.

Let's see the output of syntax tree:



The following parts are omitted, you can find the whole version in the splc executable file.

REFERENCES

1. https://github.com/cjvalera/FlexBison
2. https://github.com/Jiantastic/c-to-mips-compiler
3. https://github.com/rabishah/Mini-C-Compiler-using-Flex-And-Yacc
4. https://github.com/stardust95/TinyCompiler
5. https://aquamentus.com/flex_bison.html