编译原理实验一 实验报告

151220152 殷乔逸

1. 程序功能

本次实验中实现了对 C 语言代码的词法分析、语法分析并且进行错误提示以及无错输出语法树结构等功能。代码中词法分析部分采用了 GNU Flex 方法，语法分析部分采用了 GNU Bison 方法并且自行设计了多叉树结构。具体的程序功能将分模块叙述如下：

a) 词法分析部分

在词法分析部分主要完成了对源文件代码的词法单元解析并且提取。通过正则表达式匹配不同的 Tokens 然后将这些 Tokens 作为后续语法分析的终结符号，以结点的形式存入多叉树结构之中。

i. 正则表达式

在正则表达式部分，比较难的是设计一个正则表达式能够比较好地匹配 C 语言风格的注释，并且对于 C 语法中不允许的嵌套注释有比较良好的辨识能力。其次较难的是对指数形式的 Float 型数据进行匹配。

1. 在设计匹配注释的正则表达式时，我分为两个情况考虑，一是行级注释，二是块级注释。行级注释比较简单，重点是**对块级注释的匹配**，首先想到是所有块级注释一定是/*……*/这样的形式，那么正则表达式的首尾比较容易确定，其次是对中间内容的判定，由于要识别出嵌套注释并且不允许其成为合法词法单元，想到的是块级注释的中间内容必然不能出现*/这样的符号串，我的写法是：**匹配没有星号的符号串，或者如果有星号则后续的一个符号必然没有反斜杠。**（亮点）从集合的角度来讲，上述匹配的符号串集合的补集恰好是中间内容出现了*/的符号串，所以完整的用于匹配注释的正则表达式如下：

```
51   COMMENTS \/\/.*|(\/\*(([^*]*)|(([^*]*)[*]([^/]*)))\*\/)
```

2. 其次是**对指数形式的 Float 型数据进行匹配**，由于在编译原理课后作业中完成过类似的任务，所以比较容易，主要就是小数点位置和指数部分按整型匹配，完整的用于匹配指数形式 Float 型数据的正则表达式如下：

```
49   FLOATEXP (([0-9]*\.[0-9]+)|([0-9]+\.))[eE][+-]?(([1-9][0-9]*)|0)
```

ii. 动作行为

动作行为这一块比较简单，就是将成功匹配到的正则表达式按词法单元名称和类型构建多叉树结点，每一个终结符号形成了一个多叉树的叶子结点，最终传递给语法分析部分。比较关键的动作行为是**数值转换函数**（亮点），由于实验要求最终输出的数值都是十进制形式，通过**自行实现的数值转换函数**将其转为十进制数值，然后**通过 sprintf()函数格式化写入字符数组**。例如十六进制转换：

```
165  {INTHEX} {
166      int value = Hex_2_Dec(yytext);
167      char content[64];
168      memset(content, '\0', sizeof(content));
169      sprintf(content, "%d", value);
170
171      yylval.tree_node = Node_Initializer("INT", content, false);
172      return INT;
173      }
```

然后是指数形式的浮点数转换，比较关键的是**整个匹配的符号串按照 e 或 E 为切割符号**，切割为小数部分和指数部分，然后指数部分按照 Int 型处理，小数

部分按照普通的浮点数处理，最终通过乘积得到实际的浮点数的值，同样通过sprintf()函数格式化写入字符数组。例如切割符号串得到小数和指数部分：

```
43      if (strchr(src, 'e')==NULL){
44          decimal_str = strtok(src, "E");
45          exp_str = strtok(NULL, "E");
46      }
47      else{
48          decimal_str = strtok(src, "e");
49          exp_str = strtok(NULL, "e");
50      }
```

b) **语法分析部分**

i. **规则部分（产生式及语义动作）**

这部分主要根据附录 A 给定的语法规则书写产生式规则，具体例子如下：

```
104   /* Declarators */
105   VarDec: ID {$$=Node_Initializer("VarDec", "", true); Insert_Child($$, $1);}
```

ii. **错误恢复**

错误恢复主要实现其实是 Bison 的封装内容，我主要是通过自己设置 error 的位置来识别一些主要的，明显的语法错误类型。比较关键的是**对分号的错误识别**，比如在 Stmt 产生式中加入 error：

```
| RETURN Exp error {$$=Node_Initializer("Stmt", "", true); char str[MAXLENGTH] = "Missing \";\""; syn_error=true; yyerror(str);}
```

或者是**对括号的识别，尤其是对类似 A[4,3]这样不符合 C 语言语法的数组访问的识别（亮点）：**

```
| VarDec LB INT error {$$=Node_Initializer("VarDec", "", true); char str[MAXLENGTH] = "Missing \"]\""; syn_error=true; yyerror(str);}
```

这两个是比较关键的，明显的语法错误，可以不使用系统默认的 syntax error。

c) **多叉树结构**

多叉树结构是整个程序的核心数据结构，通过多叉树结构可以有效存储合乎词法与语法的语法树。我在设计中使用了**二叉树结构**来存储多叉树，左子结点作为该结点的孩子也就是下层结点，右子结点作为该结点的兄弟结点也就是同层结点。

```
11  typedef struct Node {
12      /* Name */
13      char type[32];
14      /* Value */
15      char value[64];
16      /* Position */
17      int linenum;
18      /* Its children and its brothers */
19      struct Node* child;
20      struct Node* sibling;
21      /* Flag for non-ternimal,
22         false for ternimal, true for non-ternimal
23      */
24      bool is_non_ternmial;
25
26  } TreeNode;
```

其中 type[]数组用于存储词法单元名称，value[]用于存储词法单元的实际值，比如 INT 型数值或者 FLOAT 型数值（以字符串形式存储），或者就是字符串（标识符等）。is_non_ternmial 则用于标记该结点是否是非终结符号，用于在前序遍历输出的时候判断是否要输出具体值。

除此之外，实现了必要函数，比如创建结点，插入子结点和兄弟结点，前序遍历等：

```
30  TreeNode* Node_Initializer(char* node_name, char* value_str, bool flag);
31  TreeNode* Get_First_Child(TreeNode* parent);
32  TreeNode* Get_First_Sibling(TreeNode* brother);
33  bool Insert_Child(TreeNode* parent, TreeNode* child);
34  bool Insert_Sibling(TreeNode* brother, TreeNode* young);
35  bool Preorder_Traversal(TreeNode* cur_root, int layers);
```

2. 编译及运行方法（亮点）

首先压缩包解压后得到的文件夹 Lab1，目录结构如下：

```
.
├── Code
├── parser
├── README
├── results.txt
├── run.sh
└── Test
```

Test 文件夹中存储了 10 个测试用例，Code 文件夹存储了所有代码文件，results.txt 文件存储了 10 个测试用例的测试结果（包括错误提示的输出和无错语法树的输出）。

i. 如果想要直接编译运行，则在 shell 命令行执行`./run.sh`，则会直接编译并且对 Test 文件夹下的用例进行测试，输出的结果按序存储在 results.txt 中。

ii. 如果仅仅想要编译，则进入 Code 文件夹执行`make parser`，则会重新编译并在上层目录 Lab1 得到可执行文件 parser。

iii. 如果想要对某一个测试用例进行测试，则在 Lab1 目录下执行`./parser xxx`,其中 xxx 代表将要被测试的用例。

3. 测试结果

部分测试结果截图如下：

i. **错误提示的输出：**

```
1  1
2  Error type A at Line 4: Unrecognized Identifiers '~'
3  2
4  Error type B at line 5: Syntax Errors
5  Error type B at line 5: Missing "]"
6  Error type B at line 6: Missing ";"
```

ii. **正确的语法树输出：**

```
172  9
173  Program (1)
174    ExtDefList (1)
175      ExtDef (1)
176        Specifier (1)
177          TYPE
178        FunDec (1)
179          ID: main
180          LP
181          RP
182        CompSt (2)
183          LC
184          DefList (7)
185            Def (7)
186              Specifier (7)
187                TYPE
188              DecList (7)
189                Dec (7)
190                  VarDec (7)
191                    ID: i
192                  ASSIGNOP
193                  Exp (7)
194                    INT: 1
195              SEMI
196          RC
```