编译原理实验一报告

一 程序功能

1.1：词法分析

词法分析过程主要是将程序的所有终结符全部识别出来，输出 token 序列，作为下一步的输入。其中要确定各个词素的种别码和属性值。我们只需要在（.l）文件中编写识别各个词法单元的正则表达式，并且在识别出特定词法单元之后设置相应的动作就可以完成词法分析。

为了解决将 int 或 float 识别为标识符的问题，可以将 int 和 float 的正则表达式（就是 int 和 float）放在标识符正则表达式的前面，这样当程序中出现 int 或者 float 时词法分析器就会先将它们识别成类型（type）词法单元。

```
21      int|float  {
22              create_lexical_unit("TYPE",yytext);
23              return TYPE;
24              }
```

```
53      {letter}({letter}|{digit}){0,31}    {
54              char* content = yytext;
55              create_lexical_unit("ID",content);
56              return ID;
57              }
```

图 1.1.1 类型的正则表达式需要在标识符正则表达式的前面

上图中词法分析器在识别出词法单元之后还要调用一个函数（create_lexical_unit），这是为了下一步做语法分析树用的。每个词法单元在识别出来之后都要调用这个函数，作为将来语法分析树的一个叶子结点。

动作的最后还包括返回一个词法符号，这个符号是在(.y)文件中定义的，它们是程序文法中的终结符。通过返回这个终结符符号，将词法分析的代码与语法分析联系到了一起。

1.2：语法分析

语法分析主要是要构造我们输入程序的语法数，那么就要定义树的数据结构，下面是书的一个节点的定义：

```
struct node{
    int is_null;  // 是否为空
    char* id;  //语(词)法单元名
    char* value; //值（如果是id或type）
    int line_no;  //第一个词素的行号
    struct node* child_nodes;  //指向第一个子结点的指针，所有子结点的链表
    struct node* right_brother;  //指向右兄弟的指针
};
```

图 1.2.1 语法分析树的节点数据结构定义

节点结构体第一个属性表示程序中是否有该节点生成的词素（如果有，那么非空，is_null=0，否则反之）。由于最后在打印语法树过程中要忽略不生成任何程序符号的节点，所以要设置这么一个属性来跳过某些文法符号的打印。第一个词素的行号也是需要维护的，在打印第一个词素行号时要用到。生成最后的多叉树我选择的是让一个节点分别连接它的第一个儿子，和右兄弟的办法。以上是对多叉树节点数据结构的介绍。

```
/* Expressions */
Exp : Exp ASSIGNOP Exp  {$$ = create_gram_unit("Exp",3,$1,$2,$3);}
  | Exp AND Exp  {$$ = create_gram_unit("Exp",3,$1,$2,$3);}
  | Exp OR Exp  {$$ = create_gram_unit("Exp",3,$1,$2,$3);}
  | Exp RELOP Exp  {$$ = create_gram_unit("Exp",3,$1,$2,$3);}
  | Exp PLUS Exp  {$$ = create_gram_unit("Exp",3,$1,$2,$3);}
```

图 1.2.2 在每个产生式规约过后需要调用创建子树的方法

如上图，当程序中不包含语法和词法错误时，每用到一个产生式做规约操作需要调用构造子树的方法（create_gram_unit），该函数以语法符号名，子节点个数，以及所有子节点作为参数传入，返回父结点指针（父结点就是这些子节点的公共父亲，也就是产生式左部）函数的功能主要是将该子树构造出来，通过子节点的属性来求出父结点的属性。这样在程序规约到最后，整个程序语法树就构造出来了。语法树构造出来之后，就对这棵语法树进行先序遍历，这就是程序没有错误的情况。

1.2.1：词法错误检测

词法错误检测十分简单，在词法分析中，如果一个单词没有匹配到所有的词法单元，而是默认匹配，那么就打印错误信息，并输出当前的行号即可。

```
. {
   has_error = 1;
   printf("Error type A at Line %d: Mysterious character \"%s\".\n",yylineno,yytext);
   fputs("Error type A at Line ", fp);
   fputs(num2str(yylineno), fp);
   fputs(": Mysterious character \"", fp);
   fputs(yytext, fp);
   fputs("\".\n", fp);
   }
```

图 1.2.1.1 词法错误的处理

1.2.2 语法错误检测及恢复

Bison 语法错误检测方法就是向产生式里添加 error 符号，当发生错误时，应该继续向下扫描，直到将所有的 token 序列扫描完毕，而不是当有错误就停滞不前。所以应将 error 放置在适当的位置。

我这里将 error 符号放置在所有的括号，分号之前。当检测到 error 时，填入这些符号，分析往往就能正常继续，因为这些符号后面通常是新的代码段。

同时，我通过设置运算符的结合性与优先级来消除了分析中的冲突：

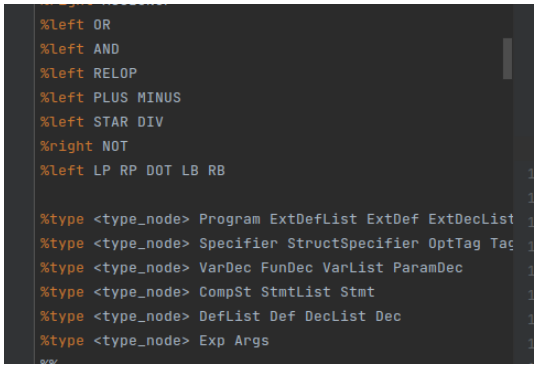

图 1.2.2.1 设置优先级和结合性

二 编译方式

我将所有的 C 语言代码全部写到了（.l）文件与（.y）文件中，所以编译方式与实验指导书上的编译方式相同：

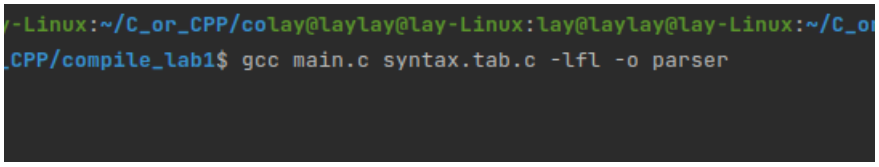

图 2.1 编译语句