编译原理实验报告

实验一：词法分析与语法分析

一、 实验目的

设计、编写一个词法和语法分析程序，加深对词法及语法分析的理解。

二、 实验要求

使用词法分析工具 GNU Flex 和语法分析工具 GNU Bison 实现一个语法和词法分析器，可以查出 C 语言源程序中的：

1)词法错误（错误类型 A：即出现 C 词法中未定义的字符以及任何不符合 C 词法单元定义的字符）；

2) 语法错误（错误类型 B）。

在输出错误提示信息时，需要输出具体的错误类型、出错的位置（源程序行号）以及相关的说明文字。

三、 实验结果

3.1 程序结构

程序的源文件由词法分析文件 lexical.l、语法分析文件 syntax.y 及程序入口 main.c 组成。使用 makefile 可以快速进行编译。经过编译后生成中间文件 lex.yy.c、syntax.tab.c、syntax.tab.h 及执行程序 complier。

测试时在根目录下使用 make 命令进行编译，使用./complier test1.cmm 来对测试文件进行测试。

3.2 程序功能

本程序实现了基本的词法与语法分析功能。当发现词法错误时，会输出“Error: type A at line …: Mysterious character “…””；当发现语法错误时，会输出“Error: type B at line …: ……（说明文字）”；当代码中不含有词法和语法错误时，则会输出语法树信息。

例：对于源代码：

```
int main()
{
    int i=1
    int j=~;
}
```

会输出错误信息：

```
Error: type A at line 3: syntax error: unexpected TYPE, expecting SEMI
Error: type A at line 4: Mysterious character “~”
```

对于正确的代码：

```
int main()
{
	int i=1;
	int j=2;

}
```

则会输出程序的语法树信息：

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: main
        LP
        RP
      CompSt (2)
        LC
        DefList (3)
          Def (3)
            Specifier (3)
              TYPE: int
            DecList (3)
              Dec (3)
                VarDec (3)
                  ID: i
                ASSIGNOP
                Exp (3)
                  INT: 1
            SEMI
          DefList (4)
            Def (4)
              Specifier (4)
                TYPE: int
              DecList (4)
                Dec (4)
                  VarDec (4)
                    ID: j
                  ASSIGNOP
                  Exp (4)
                    INT: 2
              SEMI
        RC
```

3.3 功能实现

3.3.1 词法分析

词法分析通过在 lexical.l 中书写正则表达式实现。由 int、float、if、else、while 等关键字，各种运算符，以及整形和浮点型的数字构成终结符的集合。

3.3.2 语法分析

语法分析通过在 syntax.y 中书写生成式实现。这部分与将实验指导附录中的 C 语言文法基本相同。

3.3.3 错误恢复

当出现语法错误时，除了报错外，还应进行错误恢复，将语法分析进行下去。Bison 中的错误恢复主要在通过文法中合理放置特殊标识符 error 实现。本程序一般将 error 标记放在逗号“,”分号“;”、括号“()”“{}”等前，即在新的语句开始时进行错误恢复，以使语法分析能够继续进行。

出现语法错误时，yyparse()会自动调用 yyerror()函数输出错误信息。为能输出符合格式的信息，对 yyerror()进行了重载。但是默认的错误信息为 syntax error，过于笼统，并非非常有用。为了输出更详细的错误信息，本程序使用了%define parse.error verbose 这个宏，这可以使错误信息更加详细，明确指出缺少了某个符号。

3.3.4 语法树的构建

语法树的构建是语法分析中的重要一环。语法树的每个叶节点对应一个终结符号，其他节点则对应一个语法单元。在 lexical.l 中定义了这个节点：

```
struct Node{
        int isToken;                //该节点是否为终结符即叶子节点
        int line;                   //该单元出现的行号
        int column;                 //列号
        char type[16];              //节点的类型，如 INT，FLOAT
        char text[32];              //该单元的字符内容，即变量名等
        struct Node *firstChild;    //该节点的首个子节点
        struct Node *nextSibling;   //该节点的临近的兄弟节点
    };
```

在 syntax.y 中，将所有的语法单元的属性值的类型均设为了 Node 型。

在每个词法单元被识别后，调用 setNode()函数对终结符所对应的节点设初值。在每个生成式被归约后，调用 newNode()函数将其添加进语法树中，所有被归约的语法单元互相为兄弟节点，在同一层中，它们为归约后的语法单元对应的节点的子节点。由于不同生成式中被归约的词法单元数量不定，newNode()使用了可变参数来处理，使用了 va_list 系列的宏。

3.3.5 语法树的输出

当源代码中不包含语法或词法错误时，则输出语法树。语法树的输出采用了前序遍历的方式，以递归函数实现。为了给不同层的节点设置不同的缩进格数，还定义了 depth 变量，每次向下寻找子节点时，depth++，最后根据 depth 的值设置对应的缩进。