词法与语法分析 实验报告

151220151 殷瀚 411629507@qq.com

1.实现的功能

完成了所有必做及选做任务：

1. 检测词法错误（Error Type A）；
2. 识别八进制、十六进制数，识别指数形式浮点数；
3. 检测语法错误（Error Type B）；
4. 识别行注释（//）和块注释（/*...*/）；
5. 为没有错误的代码输出语法树。

2.实现方法及数据结构

2.1 使用 C++编程

本次试验中，我使用了 c++进行编程。事实上，与使用 c 相比，几乎不需要对 flex 和 bison 的文件进行什么修改，只要把编译时的 gcc 命令改为 g++，就可以在 flex 和 bison 中使用 c++。

值得一提的是，如果改写了 yyerror(char* msg)函数，在使用 g++编译时会报 warning，因为 c++不允许将字符串常量赋给 char*变量。此时只需在函数的参数前加上 const 即可：

```
void yyerror(const char* msg);
```

2.2 树结点

在 SyntaxTree.h 和 SyntaxTree.cpp 中实现了与语法树有关的类：Node。一个 Node 对象中包含结点类型、结点名字、所在行号和属性值。其中，属性值是一个 string 类型的变量，对于 int 和 float 类型的结点，也是将数字以字符串形式存储。

使用 stl 提供的 list，可以方便地管理一个结点的所有子节点；还有一个 father 指针指向该结点的父节点。

在此之上实现了与建树、删除、打印语法树有关的方法，详见代码注释。

2.3 词法分析

2.3.1 词法规则书写

参照 c--语法手册编写正则表达式即可，选做要求也比较简单。

2.3.2 非法数字识别

这里有一个问题，如果只有上述的正则表达式，当我们输入了一个非法的数字时，比如 0XG5，词法分析器只能报告“Mysterious character ‘G'”，而不能告诉我们是’0XG5’这个整型数不合法。更严重的是，如果输入一个非法浮点数 9.6E7.6，词法分析器会认为这是两个合法浮点数(9.6E7 和.6)，而不会报错。虽然对于第二种情况，我们在语法阶段依然能发现错误，但我们还是希望能在词法分析阶段就将其检测出来。

可以定义类似于 INT_ERROR 或 FLOAT_ERROR 的类型，来列举了一些非法数字的 pattern，

以将 9.6E7.6 这样的串整个进行匹配。但是这样明显不好，很难用一种 pattern 完全覆盖各种非法数字。

但这种思路给了我启发。基于 flex 处理冲突时的两条原则（匹配最长的串，长度相等时匹配最前面的规则），事实上，我们不需要写出错误类型的模式，只需要知道什么样的字符串“看起来像”是 int 或者 float 就行了。于是有如下两条新规则：

```
UnsignedIntegerConstantLike [0-9][0-9a-zA-Z]*
FloatingConstantLike [0-9][0-9a-zA-Z+-\.]*
```

以浮点数为例，一个数字开头的、只包含数字、字母、小数点和正负号的串我们认为可能是浮点数，只要我们将 FloatConstantLike 这条规则写在 Float 规则的后面，合法浮点数依然会被 Float 规则匹配，而所有看起来像浮点数但是不符合浮点数标准的串就会被 FloatConstantLike 规则匹配，就可以识别非法数字：

```
Error type A at line 3: Illegal float point number '9.6E7.6'
Error type B at Line 3:  syntax error
Error type A at line 4: Illegal integer '3eE4'
```

2.3.3 yylloc

尽管试验手册上没写，但是需要在 lexical.l 的开头加一句#include "syntax.tab.h"才能使用 yylloc.

2.4 语法分析

2.4.1 语法规则书写

同样，参照手册即可。同时创建结点，并插入子节点。

2.4.2 注释

识别注释虽说是语法分析的任务，但实际上是在 lexical.l 文件中完成的，定义两种注释的规则之后，对应的响应函数中什么都不做即可。

行注释没什么好说的，但是块注释比预想的要复杂一些，主要是要处理嵌套的情况。直接使用正则表达式”/*”(.|\n)*”*/”肯定是不行的。

在 StackOverflow 上关于这个的讨论有不少。难点在于 flex 没有提供非贪婪模式的匹配。直接的想法就是，在”/*”之后，遇到的第一个”*/”，这之间的部分当作一块注释，正则表达式很好写，”/*”(.|\n)*?”*/”，很遗憾，flex 并不支持非贪婪匹配。

我们也可以仅仅把”/*”作为一个模式，在响应函数中通过 c/c++代码逐个读取字符串，直至读到”*/”停止，这样肯定是没毛病的，就是比较麻烦，也没有利用好 flex 的特性。

比较好的办法是使用 start conditions，其实就是为词法分析的状态机手动写了一个状态，以及这个状态相关的转移动作。网上也有不少的实现方法，我就直接参考了 flex 作者在自己博客 (https://westes.github.io/flex/manual/How-can-I-match-C_002dstyle-comments_003f.html) 上给出的方法，先在第一部分的后面加一行%x IN_COMMENT，然后规则部分写：

```
<INITIAL>"/*"        BEGIN(IN_COMMENT);
<IN_COMMENT>"*/"     BEGIN(INITIAL);
<IN_COMMENT>[^*\n]+  // eat comment in chunks
<IN_COMMENT>"*"      // eat the lone star
<IN_COMMENT>\n
```

除此之外，直接为不能嵌套的块注释写一个正则表达式也不是不可以，University of Manchester 的 cs212 课件中有详解(http://www.cs.man.ac.uk/~pjj/cs212/ex2_str_comm.html)，我试了一下，用正则式"/*"([^*]|(*)*[^*/])*(*)*"*/"去匹配是可以的，但是为什么 StackOverflow 上的大佬们都几乎不提呢，我猜有两种原因：一是用 start conditions 可以让状

态机的状态少一些；二是为了可读性和易于修改，如果再引入更复杂的限制，正则表达式就很难写了（也很难看懂和证明），但是 start conditions 的方法还是相对容易一些。

2.4.3 冲突与错误恢复

添加以下结合性声明，并依照实验手册解决悬空 else 问题：

```
/* combination principles */
%right ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right NOT
%left DOT LP RP LB RB
```

3. 编译运行方法

直接在 Code 目录下 make 即可，将依次调用 flex 和 bison 生成.c 文件，再寻找 Code 目录下所有的.c 和.cpp 文件进行编译连接得到可执行文件 parser（生成的 parser 在**根目录**下）。

在 Code 目录下执行 make test 将调用项目**根目录**下的 parser，对 Test 目录下所有.cmm 文件进行分析。

在 Code 目录下执行 make clean 将 Code 目录下的临时文件清理，不会删除根目录下的可执行文件。

4. 试验总结

通过本次试验熟悉了一下现有的分析工具，第一次试验总的来说不算难，设计实现好语法树的数据结构后，照着语法手册把规则输入进去就行了。但愿现在写好的代码不会为后面的试验埋坑（flag）。