编译原理实验报告

实验一：词法分析和语法分析

191220008 陈南疃

924690736@qq.com

一、运行方式

```
$ make
$ ../parser testfile
$ make clean
```

二、实现功能

任务编号：11

选做部分：1.1

（一）词法分析

1、正则表达式的别名复用

在正则表达式的书写过程中，利用了别名复用的机制，先定义了 `DIGIT`，`DIGITS` 和 `LETTER`：

```
DIGIT           [0-9]
DIGITS          {DIGIT}+
LETTER_         [a-zA-Z_]
```

然后在后续的的浮点数、整数和变量名的正则表达式书写中进行复用，简化了表达上的繁琐，方便后续的调试。

2、空白符的识别

常见的空白符有空格，换行符（\n），回车符（\r）和制表符（\t）。

但经过查阅，发现仍有一些其他非常见空白符，如：垂直制表符（\v），换页符（\f）。

因此将空白符的正则表达式书写为：

```
DELIM           [ \n\r\t\v\f]
WS              {DELIM}+
```

3、识别优先级的考虑

在模式匹配的过程中可能存在冲突，结合 Flex 的冲突解决机制：① 多个前缀可能匹配时，选择最长的前缀；② 某个前缀和多个模式匹配时，选择列在前面的模式。对模式匹配的顺序加以如下限制：

- 先匹配 RELOP，后匹配其他操作符 （如：`==` 和 `=`）
- 先匹配浮点数，再匹配八进制数和十六进制数，最后匹配十进制数
- 先匹配保留字，后匹配变量名

4、不合法的八进制数，十六进制数和变量名的识别

在编写八进制数，十六进制数和变量名的正则表达式后，经过简单的测试发现，虽然能够识别正确的八进制数，十六进制数和变量名，但对于不合法的八进制数，十六进制数和变量名，则会发生误判（将其中的一部分识别为正确格式，或者匹配到其他模式）。

因此，为了正确识别不合法的八进制数，十六进制数和变量名，我对其进行了单独的匹配（并依然为其建立了树结点，以便对后续的代码进行分析），处理好与其他模式的优先级关系，并输出报错信息：

```
OCTERROR        0[0-7]*[8-9]+[0-9]*
HEXERROR        0[xX][0-9a-fA-F]*[g-zG-Z]+[0-9a-zA-Z]*|0[xX]
IDERROR         {DIGIT}+{LETTER_}({LETTER_}|{DIGIT})*
```

（二）语法分析

1、减号和负号的区分

由于减号和负号的表达方式是相同的，均会被识别为 `MINUS`，因此借鉴 if-else 语句的二义性问题的解决办法，在结合性和优先级定义处添加 `NEG`，使其优先级与 `NOT` 相同，均为右结合，然后用 `%prec` 将含义为负号的 `MINUS` 的优先级修改为正确的优先级：

```
%left PLUS MINUS
%right NOT NEG
...
| MINUS Exp %prec NEG
```

2、不同类型终结符的值存储方式

在终结符中有多种不同的值类型，如整型、浮点型、字符串等。因此，在同一结构体类型中对于不同类型值在相同空间中的存储，显然可以考虑利用联合类型来保存：

```
union {
    unsigned val_int;
    float val_float;
    char val_content[33];
} val;
```

对于 `INT` 的值，用 `unsigned val_int` 来保存；对于 `FLOAT` 的值，用 `float val_float` 来保存；对于保留字、操作符、属性类型、变量名，则用 `char val_content[33]` 来保存。

3、利用枚举类型对终结符进行分类

在建立树结点和打印树结点时，若存储的是终结符，则需要根据终结符的类型，来判断使用 联合类型中的哪一个变量进行存储或输出，由于终结符模式数量较多，我定义了一个枚举类型对其进行分类，并在树结点结构体中添加该类型：

```
typedef enum { TOKEN_TYPE, TOKEN_FLOAT, TOKEN_OCT, TOKEN_HEX, TOKEN_INT,
TOKEN_ID, TOKEN_OTHER, NON_TERMINAL, NON_TERMINAL_NULL } Type;
```

4、抽象语法树的数据结构选择

在词法分析和语法分析的同时，我们需要基于语法单元建立抽象语法树，根据产生式的形式，可以判断这是一颗多子女且子女数目不定的多叉树，因此我采用了“子女-兄弟”的表示方法进行建树：

```
typedef struct Node {
    ...
    struct Node* firstChild;
    struct Node* nextSibling;
} Node;
```

5、抽象语法树建立中可变参数 va_list 的使用

由于每个树结点的子女数目的不定的，那么函数的参数数目节难以确定，经查阅资料，可以利用可变参数进行传参，只需要在函数参数中加一个子女数目 `int childNum` 的参数，即可完成在函数中对子女结点的提取：

```
void addChild(Node* parent, int childNum, ...) {
    va_list childList;
    va_start(childList, childNum);
    Node* tmp = (Node*)va_arg(childList, Node*);
    ...
    va_end(childList);
}
```

```
ExtDef : Specifier ExtDecList SEMI        { ... addChild($$, 3, $1, $2, $3); }
```

6、编写 printError 函数代替 yyerror 函数

由于 yyerror 函数会在检测到语法错误时自动被调用，其默认参数并不是我每次都期望得到的，所以我删除在 yyerror 函数中的打印语句，重写了一个功能类似的 printError 函数，增加了行号参数，在每个错误产生式后手动调用，传入合适的行号（不一定每次都是 yyerror 函数中的 yylineno），并根据错误的类型针对性地输出期望的报错信息，以便调试和查看。

7、错误恢复

错误恢复部分的逻辑相对复杂，需要考虑在上层还是下层进行错误恢复，以及错误产生式也可能出现冲突，因而可能出现同一处错误产生了多次报错，为了解决这个问题，我定义了全局变量 lastErrorLine，与当前的行号进行比较，进而判断是否是重复输出。

此外，由于仍可能有少数情况导致 printError 函数未被调用，所以我还定义了全局变量 missLine 来辅助判断是否有漏掉的错误恢复没输出，如有漏掉的错误恢复则立即输出报错语句。

三、一些建议

实验手册在错误恢复的部分讲述得比较抽象，不太容易理解清楚，再加上做到这一部分的时候可能上课还没讲到，导致我做这一部分的时候思维有些混乱，没太能理清楚，希望可以在手册上举一个具体的例子辅助理解。此外，语法分析部分提到的 %locations 需要放在 syntax.y 中，而不是 Flex 源文件 lexical.l 中，可以稍加说明。