作者:

张驰一 19307130043

宋宇霖 19307130139

报告目录

flex的用法
工作原理
如何识别不同token及其类型
声明
KEYWORD
OPERATOR
DELIMITER
数字
ID
STRING
COMMENT
规则
简单的情况
相对复杂的情况
ID
INTEGER
更为复杂的情况
STRING
COMMENT
判断token行列号
输出及报错
贡献

> 撰写项目报告，说明flex的用法，识别不同token所使用的正则表达式及其原理，如何判断token的行列号及类型，如何实现报错功能等等，在结尾标明分工及贡献百分比

flex的用法

flex可以快速将正则表达式的语法分析转化为C语言程序，具体规则以[flex官方文档]为例，

第一个`%{`和`%}`内，是c程序，可以在这里写对于库的引入，和一些常量的定义。

然后是声明部分，

形如`DIGIT    [0-9]`，可以视为是一种alias。

以上两部分都是定义区。`%%`标识着规则区的开始，

从上到下，每次遇到匹配的文本，就会执行右边的程序。flex提供了一些可以使用的参数，例如常用的是`yytext`对应当前匹配到的原字符串，`yymore`告诉扫描器把匹配到的下一个token加到yytext后面，而不是覆盖现有的yytext。

在匹配规则方面，flex会进行贪心匹配，直至匹配到有规则对应的最长字符串。如果最长字符串仍对应多个规则，那么将执行最靠前的规则。因此规则的顺序十分重要。

第二个`%%`可以定义一些函数，以方便实现执行程序。

```
/* scanner for a toy Pascal-like language */

%{
/* need this for the call to atof() below */
#include <math.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+    {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }

"+"|"-"|"*"|"/"   printf( "An operator: %s\n", yytext );

.           printf( "Unrecognized character: %s\n", yytext );
...
%%

main( argc, argv )
int argc;
char **argv;
    {
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;
    yylex();
    }
```

工作原理

flex从头开始扫描输入程序，尽可能多地进行贪心匹配，直至匹配到有规则对应的最长字符串。如果最长字符串仍对应多个规则，那么将执行最靠前的规则。执行完规则之后flex将从上次匹配到的串终点开始继续扫描和匹配，直至扫描完一遍输入程序。flex只会扫描一遍程序，因此规则的次序，相互之前的包含关系需要仔细斟酌。

如何识别不同token及其类型

声明

主要有keyword, string, operator, delimiter, ID, WS, comment和一般数字(包含integer和real)

KEYWORD

```
KEYWORD
("AND"|"ARRAY"|"BEGIN"|"BY"|"DIV"|"DO"|"ELSE"|"ELSIF"|"END"|"EXIT"|"FOR"|"IF"
|"IN"|"IS"|"LOOP"|"MOD"|"NOT"|"OF"|"OR"|"OUT"|"PROCEDURE"|"PROGRAM"|"READ"|"R
ECORD"|"RETURN"|"THEN"|"TO"|"TYPE"|"VAR"|"WHILE"|"WRITE")
```

根据PCAT Programming Language Reference Manual，一个个读入所有的关键词。

OPERATOR

```
OPERATOR    (":="|"+"|"-"|"*"|"/"|"<"|"<="|">"|">="|"="|"<>")
```

逐一枚举。

DELIMITER

```
DELIMITER   (":"|";"|","|"."|"("|")"|"["|"]"|"{"|"}"|"[<"|">]"|'\')
```

逐一枚举。

数字

```
DIGIT       [0-9]
INTEGER     {DIGIT}+
REAL        {DIGIT}+"."{DIGIT}*
```

首先确定DIGIT范围，在基于DIGIT确定INT和REAL的范围

注意一个细节：real的小数点前必须有数字，小数点后可以没有数字。

ID

```
ID          [a-zA-Z][a-zA-Z0-9]*
```

首字符为英语字母，后跟任意数量的英语字母和数字

STRING

```
STRING      \"[^\"^\n]*\"
UNSTRING    \"[^\"^\n]*\n
```

在manual中的定义为： `STRING = '"' (NOT('"'))* '"'` ，即两边是双引号，中间是除了双引号以外的符号。但是事实上STRING仅接受可打印的ASCII字符，因此上述定义并不准确。又由于case11中要求对错误STRING进行辨别，所以我定义了两类STRING: 正常匹配的和非正常终止的。

具体逻辑将在规则部分详细介绍。

COMMENT

本次project中comment并没有用正则表达式直接给出匹配规则，这是因为正则表达式不能够比较好地处理各类COMMENT错误。因此我将匹配过程分解为多步，利用了排他性的开始条件与其他token“隔离”开处理。具体代码如下：

```
<COMMENT>.              {col ++; yymore();}
<COMMENT>\n             {col = 1; row++; yymore();}
<COMMENT>"*)"           {
                                BEGIN INITIAL;
                                col += 2;
                                CommentOutput(c_ori_row, c_ori_col,
"Comment", yytext, "");
                        }
<COMMENT><<EOF>>        {
                                CommentOutput(c_ori_row, c_ori_col,
"Invalid", yytext, "ERROR: Unterminated comment");
                                BEGIN INITIAL;
                                cout << "Number of tokens: " << TokensNumber
<< endl;
                                return C_EOF;
                        }
```

具体逻辑将在规则一节中详述.

规则

简单的情况

对于简单的TOKEN匹配情况，仅用正则表达式即可很好地分析，只需要根据情况实时更新字符所在的行和列的位置，并输出字符的类别和内容。对于它们，编译程序设计的关键在于不同匹配式的优先级。

首先是KETWORD，关键词优先应优先提取。其余各类字符的顺序可以根据正则表达式对应的集合是否有包含关系，进行逐一分析。字符串和注释由于有比较特殊的开始符，因此不会和其他token所冲突，放在靠前靠后的位置都可以。

最终我们的匹配顺序为`KEYWORD>STRING&UNSTRING>OPERATOR>DELIMITER>ID>INTEGER>REAL>COMMENT类`，可行的顺序事实上并不唯一。

下面是简单情况的处理规则。

```
<INITIAL><<EOF>> {
                        cout << endl << "Number of tokens: " << TokensNumber
<< endl;
                        return T_EOF;
                 }

{WS}            col += strlen(yytext);

{KEYWORD}       TokenOutput(row, col, "Keyword  ", yytext, "");
```

```
{OPERATOR}      TokenOutput(row, col, "Operator ", yytext, "");
{DELIMITER}     TokenOutput(row, col, "Delimiter", yytext, "");
{REAL}          TokenOutput(row, col, "Real", yytext, "");
\n              {
                row++; col = 1;
                }

```

对于没有匹配到的符号，如果是空格，则col++；否则应该是不正确的字符，需要输出错误信息。

```
.               {
                        if(!strcmp(yytext, " ")){
                                col++;
                        }
                        else{
                                TokenOutput(row, col, "Invalid", yytext,
"ERROR: Bad character");
                        }
                }
```

相对复杂的情况

对于可能出错的字符，我们在匹配时需要对匹配到的串进行对应错误情况的分析。事实上，错误的可能性多种多样，难以完善地分析，因此本次实验仅针对具体的错误案例：ID过长和整数过大进行处理。代码如下：

ID

```
{ID}            {
                        if (strlen(yytext) > 255){
                                char msg[] = "ERROR: Identifier is longer
than 255";
                                TokenOutput(row, col, "Invalid", yytext,
msg);
                                return 1;
                        }
                        TokenOutput(row, col, "Identifer", yytext, "");
                }


```

INTEGER

```
{INTEGER}       {
                        if(strlen(yytext) > 9 && atoi(yytext) == -1)
                        {
                                char msg[] = "ERROR: Integer out of range";
                                TokenOutput(row, col, "Invalid", yytext,
msg);
                                return 1;
                        }
                        TokenOutput(row, col, "Integer", yytext, "");
                }

```

更为复杂的情况

STRING

String的错误可能有：过长、包含不该有tab或是回车，未终止等。前两者好办，只需要检查这些不该有的特性。对于未终止的情况，如果我们用正则表达式来匹配未终止的字符串，难免会匹配到其他token，从而无法正确识别出全部读token。因此我引入了unstring这一类token，表示未终结的字符串，在字符串出错时就即使结束匹配并报错。在匹配时，匹配器在遇到双印号第一个回车的时候，就会将其匹配为unstring，从而报错。

```
STRING      \"[^\"^\n]*\"
UNSTRING    \"[^\"^\n]*\n
```

代码如下：

```
{UNSTRING}      {

                        col = 1;
                        TokenOutput(row, col, "Invalid", yytext, "ERROR:
Unterminated string");
                        row++;
                }

{STRING}        {
                        if (strlen(yytext) > 257){
                                char msg[] = "ERROR: String is longer than
255";
                                TokenOutput(row, col, "Invalid", yytext,
msg);
                                return 1;
                        }
                        for(int cnt=0; cnt<strlen(yytext); cnt++){
                                if(yytext[cnt] == '\t'){
                                        char msg[] = "ERROR: String contains
tab";
                                        TokenOutput(row, col, "Invalid",
yytext, msg);
                                        return 1;
```

```
                                }
                        }
                        TokenOutput(row, col, "String   ", yytext, "");
                }
```

COMMENT

comment的主要难点在于其可以容纳各种各样的符号，包括tab和换行符，需要能正确统计行列号。此外如果遇到未终止的情况也需要处理。鉴于情况复杂，且与之前我们定义的规则有包含关系，难以通过调换顺序的方式来协调，因此我们采用排他性前提条件COMMENT来进行匹配。该条件的原理是：如果该条件成立，则其余其他条件下的规则均被忽视，只考虑当前条件下的规则。默认是INITIAL条件。条件的启用和切换可以通过BEGIN xxx_condition命令进行。注意：<<EOF>>需要显式指明INITIAL，才会是仅在INITIAL条件下启用。

我设计的匹配逻辑是：从匹配到(*开始进入COMMENT条件，接下来适用的规则仅有如下四条：

```
"(*"                    {
                                c_ori_col = col;
                                c_ori_row = row;
                                col += 2;
                                BEGIN COMMENT;
                                yymore();
                        }
<COMMENT>.              {col ++; yymore();}

<COMMENT>\n             {col = 1; row++; yymore();}

<COMMENT>"*)"           {
                                BEGIN INITIAL;
                                col += 2;
                                CommentOutput(c_ori_row, c_ori_col,
"Comment", yytext, "");
                        }



<COMMENT><<EOF>>        {
                                CommentOutput(c_ori_row, c_ori_col,
"Invalid", yytext, "ERROR: Unterminated comment");//c_ori_row, c_ori_col分别保留的是注释开头的行号与列号
                                BEGIN INITIAL;
                                cout << "Number of tokens: " << TokensNumber
<< endl;
                                return C_EOF;
                        }
```

1. 如果扫描到\n，那么就让行数++
2. 如果扫描到*)，那么注释匹配正常结束，回到默认的INITIAL状态，并返回注释信息
3. 如果匹配到<EOF>，说明注释意外终止，回到默认的INITIAL状态，并返回错误信息和注释信息
4. 如果匹配到其他，那么就继续匹配

有必要特别说明的是其中用到了yymore()函数，这个函数告诉扫描器：当匹配到下一个token时，把它加到现在的yytext后面，而不是覆盖现在的 yytext。

判断token行列号

ROW的识别可以通过在每次识别到换行符`\n`时，使全局变量`row`加1. 指的一提的是comment中可能出现多个换行符，因此对于每个\n都需要在对应处理中使row++

COL的识别可以通过每次识别符号时，使全局变量`col`增加`strlen(yytext)`，并在换行时重置为1.

(对于制表符可以跳至下一个mod4余1的位，当然这个例子里`\t`只会出现在第一行，所以可以用`+4`替代)

输出及报错

在lexer.lex文件中新建辅助函数，我们可以输出词法分析的结果。

针对每个错误，我们可以在执行匹配token操作的程序前，先进行异常判断和处理。

对于各类token，我们设计了以下通用函数输出其信息和错误情况：

```
void TokenOutput(int& row, int& col, const char* type, char* text, char*
msg){

        cout << left << setw(8) << row << setw(8) << col;
        if(msg != "")
                cout << setw(20) << type << msg << ": " << text << endl;
        else
                cout << setw(20) << type << text << endl;
        col += strlen(text);
        TokensNumber++;

        return;
}

```

其中msg是报错信息，如果没有出错，则msg为空。

对于comment，我们用另一个类似的函数输出：

```
void CommentOutput(int& row, int& col, const char* type, char* text, char*
msg){
        cout << left << setw(8) << row << setw(8) << col;
        if(msg != "")
                cout << setw(20) << type << msg << ": " << text << endl;
        else
                cout << setw(20) << type << text << endl;
        return;
}
```

贡献

两位成员贡献比均为50%，具体如下：

张驰一：

- 设计KEYWORD, OPERATOR, DELIMITER，ID，数字类型的正则表达式。
- 对匹配顺序进行思考与设计
- 实现基本的统计token行列号方法
- 实现识别字符的格式化输出
- 完成50%报告撰写

宋宇霖：

- 修改STRING, COMMENT，ID，INTEGER的正则表达式以应对case11中的各类错误
- 设计不同的匹配规则，实现各种错误的准确识别和报错输出
- 修改token行列号计算方式以确保代码出错的情况下依然可以统计行列。
- 对代码输出格式，规则架构等细节进行修改
- 完成50%报告撰写

两人在充分沟通，双方都完全理解的情况下共同完成全部11个case。