编译原理实验报告：词法分析器

15331416 赵寒旭 数字媒体技术

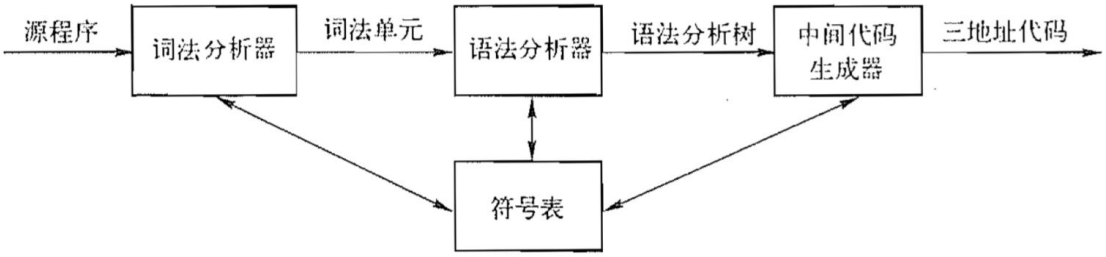
1. 实验目的

用 lex 实现一个词法分析器

2. 实验思路

2.1 词法分析器的作用

词法分析是编译的第一阶段，主要任务是读入源程序的输入字符，将它们组成词素，生成并输出一个词法单元序列，每个词法单元对应一个词素。本次实验中词法分析程序使用 lex 自动生成。



2.2 词法单元类型

引自 https://en.wikipedia.org/wiki/Lexical_analysis#Token

本文词法单元类型及内容参考下表：

Examples of token values

Token name	Sample token values
identifier	x, color, UP
keyword	if, while, return
separator	}, (, ;
operator	+, <, =
literal	true, 6.02e23, "music"
comment	// must be negative, /* Retrieves user data */

identifier（标识符）：由程序员选择的名称

keyword（关键字）：已由编程语言确定的名称

separator（分隔符）：标点符号和成对的分隔符号

operator（操作符）：操作参数产生结果

literal（文字量）：数字和文本

comment（注释）：包括行注释和块注释

2.3 词法分析器生成工具 Lex

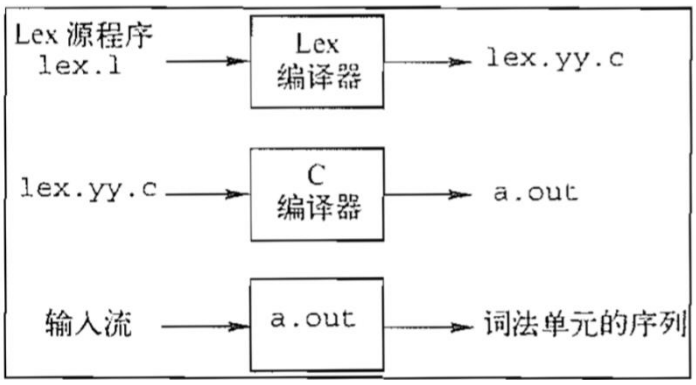
Lex 支持使用正则表达式来描述各个词法单元的模式，由此给出一个词法分析器的规约。

（1）用 Lex 创建一个词法分析器的过程

1）用 Lex 语言写一个输入文件 lex.l，描述将要生成的词法分析器。

2）Lex 编译器将 lex.l 转换成 c 语言程序 lex.yy.c

3）lex.yy.c 被 c 编译器便以为一个.exe 文件，得到一个读取输入字符流并生成词法单元流的可运行的词法分析器。



（2）Lex 程序的结构

声明部分

%%

转换规则

%%

辅助函数

1）声明部分

首先可有起始于"%{"符号，终止于"%}"符号包括 include 语句、声明语句在内的 C 语句。

包括变量和明示常量，可以声明词法单元的名字或正则定义。

2）转换规则

形式：模式 { 动作 }

每个模式是一个正则表达式，可以使用声明部分中定义的变量。

3）辅助函数

包含各个动作需要使用的所有辅助函数。

2.4 Lex 变量

```
yyin        FILE* 类型。 它指向 lexer 正在解析的当前文件。
yyout       FILE* 类型。 它指向记录 lexer 输出的位置。 缺省情况下，yyin 和 yyout 都指向标准输入和输出。
yytext      匹配模式的文本存储在这一变量中（char*）。
yyleng      给出匹配模式的长度。
yylineno    提供当前的行数信息。 （lexer不一定支持。）
```

实验中使用 yyin 指向待解析代码文件，用 yytext 获得模式匹配文本

2.5 Lex 函数

```
yylex()      这一函数开始分析。 它由 Lex 自动生成。
yywrap()     这一函数在文件（或输入）的末尾调用。 如果函数的返回值是1，就停止解析。
             因此它可以用来解析多个文件。 代码可以写在第三段，这就能够解析多个文件。
             方法是使用 yyin 文件指针（见上表）指向不同的文件，直到所有的文件都被解析。
             最后，yywrap() 可以返回 1 来表示解析的结束。
yyless(int n)   这一函数可以用来送回除了前 n 个字符外的所有读出标记。
yymore()     这一函数告诉 Lexer 将下一个标记附加到当前标记后。
```

实验中使用 yylex()函数分析代码文本，使 yywrap()返回值为 1 解析完毕停止解析。

2.6 正则表达式相关字符及其含义列表

A-Z, 0-9, a-z	构成了部分模式的字符和数字。
.	匹配任意字符，除了 \n。
-	用来指定范围。例如：A-Z 指从 A 到 Z 之间的所有字符。
[ ]	一个字符集合。匹配括号内的 任意 字符。如果第一个字符是 ^ 那么它表示否定模式。例如：[abC] 匹配 a, b, 和 C中的任何一个。
*	匹配 0个或者多个上述的模式。
+	匹配 1个或者多个上述模式。
?	匹配 0个或1个上述模式。
$	作为模式的最后一个字符匹配一行的结尾。
{ }	指出一个模式可能出现的次数。 例如：A{1,3} 表示 A 可能出现1次或3次。
\	用来转义元字符。同样用来覆盖字符在此表中定义的特殊意义，只取字符的本意。
^	否定。
|	表达式间的逻辑或。
"<一些符号>"	字符的字面含义。元字符具有。
/	向前匹配。如果在匹配的模版中的“/”后跟有后续表达式，只匹配模版中“/”前 面的部分。如：如果输入 A01，那么在模版 A0/1 中的 A0 是匹配的。
( )	将一系列常规表达式分组。

3. 实验步骤

3.1 定义需要匹配的具体词法单元对象

（1）标识符

```
/* -------------definition starting-------------*/
/* ------- identifier ------- */
ID [a-zA-Z_][a-zA-Z_0-9]*

```

（2）关键字

```
/* ------- keyword ------- */
TYPE int|float
STRUCT struct
RETURN return
IF if
ELSE else
WHILE while
BREAK break
```

（3）分隔符

```
/* ------- separator ------- */
SEMI_COLON [;]
COMMA [,]
LP \(
RP \)
LB \[
RB \]
LC \{
RC \}
```

（4）操作符

```
/* ------- operator ------- */
ASSIGNOP [=]
RELOP [>]|[<]|[>][=]|[<][=]|[=][=]|[!][=](^[=])
PLUS [+]
MINUS [-]
STAR [*]
DIV [/]
AND [&][&]
OR [|][|]
DOT [.]
NOT [!]
```

（5）文字量

```
/* ------- literal ------- */
INT_DEX [1-9][0-9]*|[0]
INT_HEX [0][Xx]([1-9][0-9]*|[0])
INT_OCT [0][0-7]
FLOAT [0-9]*[.][0-9]+([eE][+-]?[0-9]*|[0])?f?
TRUE true
```

（6）空白换行及制表符（直接跳过）

```

/* ------- drop this part ------- */
SPACE [ \n\t]

/* -------------definition ending-------------*/
```

3.2 匹配后操作的实现

（1）关键字

```
{TYPE} |
{STRUCT} |
{RETURN} |
{IF} |
{ELSE} |
{WHILE} |
{BREAK} {
  /* ------- keyword ------- */
    printf("<keyword, %s>\n", yytext);
}
```

（2）分隔符

```
{SEMI_COLON} |
{COMMA} |
{LP} |
{RP} |
{LB} |
{RB} |
{LC} |
{RC} {
  /* ------- separator ------- */
    printf("<separator, %s>\n", yytext);
}
```

（3）操作符

```
{ASSIGNOP} |
{RELOP} |
{PLUS} |
{MINUS} |
{STAR} |
{DIV} |
{AND} |
{OR} |
{DOT} |
{NOT} {
  /* ------- operator ------- */
    printf("<operator, %s>\n", yytext);
}
```

（4）文字量

```
{INT_DEX} |
{INT_HEX} |
{INT_OCT} |
{FLOAT} |
{TRUE} {
  /* ------- literal -------*/
    printf("<literal, %s>\n", yytext);
}


```

（5）标识符

```
{ID} {
  /* ------- identifier ------- */
    printf("<identifier, %s>\n", yytext);
}
```

（6）空白，换行及制表符

```
{SPACE} {
    /* ------- drop this part ------- */
}
```

3.3 c 语言辅助部分

（1）头文件包含

```
%{
#include "stdio.h"
#include "stdlib.h"

%}
```

（2）yywrap 返回值设置

当词法分析程序遇到文件结尾时，它调用例程 yywrap()来找出下一步要做什么，如果 yywrap()返回 0，则扫描程序就继续扫描，如果返回 1，则扫描程序就返回报告文件结尾的零标记。

lex 库中的 yywrap()的标准版本总是返回 1，但是可以用自己的值来替代它，如果 yywrap()返回指示有更多输入的 0，那么它首先需要调整指向新文件的 yyin，可能需要使用 fopen()。

本次实验设置返回值为 1，仅读取一个文件。

```
%%
int yywrap() {
  return 1;
}
```

（3）main 函数

```
int main(int argc, char** argv) {
   if (argc > 1) {
       if (!(yyin = fopen(argv[1], "r"))) {
           perror(argv[1]);
           return 1;
       }
   }

   yylex();
   return 0;
 }
```

yyin 指向待解析代码文件。

由 lex 创建的扫描程序有入口点 yylex()。调用 yylex()启动或重新开始扫描。如果 lex 动作执行将数值传递给调用程序的 return，那么 yylex()的下次调用就从它停止的地方继续。

4. 实验结果

4.1 运行步骤

```
Windows PowerShell
PS C:\github\lex\lexer> flex lexer.l
PS C:\github\lex\lexer> gcc -o lexer.exe lex.yy.c
PS C:\github\lex\lexer> .\lexer code.txt
```

进入根目录下运行文件，步骤如图。

4.2 运行结果

待分析代码文件 code.txt

```
1.  {
2.      int i; int j; float[100] a; float v; float x;
3.
4.      while ( true ) {
5.          do i = j+1; while ( a[i] < v);
6.          do j = j-1; while ( a[j] > v);
7.          if ( i >= j) break;
8.          x = a[i]; a[i] = a[j]; a[i] = x;
9.      }
10. }
```

1-4 行：

```
<separator, {>
<keyword, int>
<identifier, i>
<separator, ;>
<keyword, int>
<identifier, j>
<separator, ;>
<keyword, float>
<separator, [>
<literal, 100>
<separator, ]>
<identifier, a>
<separator, ;>
<keyword, float>
<identifier, v>
<separator, ;>
<keyword, float>
<identifier, x>
<separator, ;>
<keyword, while>
<separator, (>
<literal, true>
<separator, )>
<separator, {>
```

第 5 行 do i = j+1; while (a[i] < v);

```
<identifier, do>
<identifier, i>
<operator, =>
<identifier, j>
<operator, +>
<literal, 1>
<separator, ;>
<keyword, while>
<separator, (>
<identifier, a>
<separator, [>
<identifier, i>
<separator, ]>
<operator, <>
<identifier, v>
<separator, )>
<separator, ;>
```

第 6-10 行

```
<identifier, do>
<identifier, j>
<operator, =>
<identifier, j>
<operator, ->
<literal, 1>
<separator, ;>
<keyword, while>
<separator, (>
<identifier, a>
<separator, [>
<identifier, j>
<separator, ]>
<operator, >>
<identifier, v>
<separator, )>
<separator, ;>
<keyword, if>
<separator, (>
<identifier, i>
<operator, >=>
<identifier, j>
<separator, )>
<keyword, break>
<separator, ;>
<identifier, x>
<operator, =>
<identifier, a>
<separator, [>
<identifier, i>
<separator, ]>
<separator, ;>
<identifier, a>
<separator, [>
<identifier, i>
<separator, ]>
<operator, =>
<identifier, a>
<separator, [>
<identifier, j>
<separator, ]>
<separator, ;>
<identifier, a>
<separator, [>
<identifier, i>
<separator, ]>
<operator, =>
<identifier, x>
<separator, ;>
<separator, }>
<separator, }>
```

5. 遇到的问题及解决方案

5.1 未定义 yywrap( )函数返回值

```
PS C:\github\lex\lexer> flex lexer.l
PS C:\github\lex\lexer> gcc -o lexer.exe lex.yy.c
C:\Users\ZHAOHA~1\AppData\Local\Temp\cc4bU5oi.o:lex.yy.c:(.text+0x33a): undefined reference to `yywrap'
C:\Users\ZHAOHA~1\AppData\Local\Temp\cc4bU5oi.o:lex.yy.c:(.text+0xa38): undefined reference to `yywrap'
collect2.exe: error: ld returned 1 exit status
PS C:\github\lex\lexer>
```

在编译为 c 文件时报错，显示对'yywrap'未定义的引用。

添加 yywrap( )定义后解决。

5.2 Lex 中的冲突解决

```
53  %%
54
55  {ID} {
56    /* ------- identifier ------- */
57      printf("<identifier, %s>\n", yytext);
58  }
59
60  {TYPE} |
61  {STRUCT} |
62  {RETURN} |
63  {IF} |
64  {ELSE} |
65  {WHILE} |
66  {BREAK} {
67    /* ------- keyword ------- */
68      printf("<keyword, %s>\n", yytext);
69  }
```

```
Windows PowerShell
PS C:\github\lex\lexer> flex lexer.l
"lexer.l", line 60: warning, rule cannot be matched
"lexer.l", line 61: warning, rule cannot be matched
"lexer.l", line 62: warning, rule cannot be matched
"lexer.l", line 63: warning, rule cannot be matched
"lexer.l", line 64: warning, rule cannot be matched
"lexer.l", line 65: warning, rule cannot be matched
"lexer.l", line 66: warning, rule cannot be matched
"lexer.l", line 105: warning, rule cannot be matched
PS C:\github\lex\lexer>
```

在声明变量时，按先标识符后关键字的顺序，一开始在编写转换规则部分代码时，仍按照此顺序，把标识符先匹配，这样造成和后方关键字的冲突，使关键字部分无法匹配。

考虑到 Lex 中的冲突解决办法，当输入的多个前缀与一个或多个模式匹配时，Lex 用如下规则选择正确的词素：

1）总是选择最长的前缀。

2）如果最长的可能前缀与多个模式匹配，总是选择在 Lex 程序中先被列出的模式。

选择在 Lex 程序中先被列出的模式匹配，标识符的匹配应该放在关键字后面！

把 ID 的匹配移到最后，仅在 SPACE 前，再次运行无错误。，

```
Windows PowerShell
<separator, }>
PS C:\github\lex\lexer> flex lexer.l
"lexer.l", line 60: warning, rule cannot be matched
"lexer.l", line 61: warning, rule cannot be matched
"lexer.l", line 62: warning, rule cannot be matched
"lexer.l", line 63: warning, rule cannot be matched
"lexer.l", line 64: warning, rule cannot be matched
"lexer.l", line 65: warning, rule cannot be matched
"lexer.l", line 66: warning, rule cannot be matched
"lexer.l", line 105: warning, rule cannot be matched
PS C:\github\lex\lexer> flex lexer.l
PS C:\github\lex\lexer> gcc -o lexer.exe lex.yy.c
PS C:\github\lex\lexer> .\lexer code.txt
<separator, {>
<keyword, int>
<identifier, i>
<separator, ;>
<keyword, int>
<identifier, j>
<separator, ;>
```

注：由第一条规则可以知道，>=和>号将会自动区分，不必再考虑预读，Lex 会自动地向前读入一个字符并选择最长匹配。