实现词法分析器

杨侯哲 李煦阳

October 2020

目录

1 实验描述

实验内容 列出你的编译器支持的语言特性所涉及的单词，设计正规定义。你将利用 Flex 工具实现词法分析器，识别程序中所有单词，将其转化为单词流。也就是说：借助 Flex 完成这样一个程序，它的输入是一个程序，它的输出是每一个文法单元的类别、词素、以及必要的属性（比如，对于 NUMBER 会有属性它的“数值”属性；对于 ID 会有它在符号表的“序号”，有些标识符会有相同的“序号”。）

这需要设计符号表。当然目前符号表项还只是词素等简单内容，但符号表的数据结构，搜索算法，词素的保存，保留字的处理等问题都可以考虑了。

你需要验证你的程序：输入简单的源程序，输出单词流每个单词的词素内容、单词类别和属性（常数的属性可以是数值，标识符可以是指向符号表的指针）。

效果如下例：

```
main(){
    int a;
    if(a==0)
        a=a+1;
}
```

一个可能的输出结果为

```
ID              main    0
LPAREN          (
RPAREN          )
LBRACE          {
INT             int
ID              a       1
SEMICOLON       ;
IF              if
LPAREN          (
ID              a       1
EQ              ==
NUMBER          0       0
RPAREN          )
ID              a       1
ASSIGN          =
ID              a       1
PLUS            +
NUMBER          1       1
SEMICOLON       ;
RBRACE          }
```

其中每列分别为单词、词素、属性。

2 参考流程

2.1 Flex 程序基础结构

一个简单的 Flex 结构程序如下

```
%option noyywrap
%top{
#include<math.h>
}
%{
    int chars=0,words=0,lines=0;
%}

word    [a-zA-Z]+
line \n
char    .

%%

{word}   {words++;chars+=strlen(yytext);}
{line}  {lines++;}
{char}   {chars++;}

%%

int main(){
    yylex();
    fprintf(yyout,"%8d%8d%8d\n",lines,words,chars);
    return 0;
}
```

按照规范来说，Flex 程序分为定义部分、规则部分、用户子例程三个部分，每个部分之间用两个% 分隔。

2.1.1 定义部分

定义部分包含选项、文字块、开始条件、转换状态、规则等。

在上文给出的样例中**%option noyywrap** 即为一个选项，控制 flex 的一些功能，具体来说，这里的选项功能为去掉默认的 yywrap 函数调用，这是一个早期 lex 遗留的鸡肋，设计用来对应多文件输入的情况，在每次 yylex 结束后调用，但一般来说用户往往不会用到这个特性。

而用**%{%}** 包围起来的部分为文字块，可以看到块内可以直接书写 C 代码，Flex 会把文字块内的内容原封不动的复制到编译好的 C 文件中，而**%top{}** 块也为文字块，只是 Flex 会将这部分内容

放到编译文件的开头，一般用来引用额外的头文件，这里值得说明的是，如果观察 Flex 编译出的文件，可以发现它默认包含了以下内容

```
/* begin standard C headers. */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

/* end standard C headers. */
```

也就是说这部分文件其实不需要额外的声明就可以直接使用。

规则即为正规定义声明。Flex 除了支持我们学习的正则表达式的元字符包括 [] * + ? | () 以外，还支持像 {} / ^$ 等等元字符，可以指定“匹配除某个字符之外的字符”、“重复某个规则的若干次”，你可以在这里找到说明。

```
a{3,5} a{3,} a{3}
^"a*"$
[^\n]
[a-z]+ [a-zA-z0-9]
(ab|cd\*)?
0/1
```

除此以外 Flex 还支持一些其他的特殊元字符，我们在后面介绍特性时会介绍到。

2.1.2 规则部分

规则部分包含模式行与 C 代码，这里的写法很好理解，需要说明的是当存在二义性问题时，Flex 采用两个简单的原则来处理矛盾

1. 匹配输入时匹配尽可能多的字符串——最长前缀原则。
2. 如果两个模式都可以匹配的话，匹配在程序中更早出现的模式。

这里的更早出现，指的就是规则部分对于不同模式的书写先后顺序，例如：

```
    ...
    while while
    word [a-zA-Z]+
    line \n
    char .
    %%
    {while} {...}
    {word}  {...}
    {line}  {...}
```

```
    {char}  {...}
    ...
```

当输入为 while 时会匹配到 **while** 的模式中。

2.1.3 用户子例程

用户子例程的内容会被原样拷贝至 C 文件，通常包括规则中需要调用的函数。在主函数中通过调用 yylex 开始词法分析的过程，对于输入输出流的重定向我们会在之后提到。

2.2 C++ 版本

如果我们想要调用一些 C++ 中的标准库，或者说运用 C++ 的语法，对应的 Flex 程序结构需要做出一些调整

```
%option noyywrap
%top{
#include<map>
#include<iomanip>
}
%{
    int chars=0,words=0,lines=0;
%}

word    [a-zA-Z]+
line \n
char    .

%%
{word}   {words++;chars+=strlen(yytext);}
{line}  {lines++;}
{char}   {chars++;}
%%
int main(){
    yyFlexLexer lexer;
    lexer.yylex();
    std::cout<<std::setw(8)<<lines<<std::setw(8)<<words<<std::setw(8)<<chars<<std::endl;
    return 0;
}
```

可以看出，主要的差别在于用户子例程部分，我们需要按照 C++ 的风格创建词法分析器对象，而后调用对象的 yylex 函数。另外，C++ 版本默认引用的头文件也有所区别。

```
/* begin standard C++ headers. */
```

```
#include <iostream>
#include <errno.h>
#include <cstdlib>
#include <cstdio>
#include <cstring>
/* end standard C++ headers. */
```

2.3 运行测试

一个简单的测试 Makefile 如下

```
.PHONY:lc,lcc,clean
lc:
    flex sysy.l
    gcc lex.yy.c -o lc.out
    ./lc.out
lcc:
    flex -+ sysycc.l
    g++ lex.yy.cc -o lcc.out
    ./lcc.out
clean:
    rm *.out
```

当我们的词法分析器识别到文件结束符的时候，yylex 函数默认会结束，如果我们采用终端输入的方式，在 Windows 环境下敲 **ctrl+z** 表示文件结束符，而在 Mac 或 Linux 环境下可以通过 **ctrl+d** 表示文件结束。

2.4 输入输出流

显然，我们不希望每次执行翻译过程都要在终端中敲键盘输入、在终端中查看输出，那么对输入输出流的重定向就必不可少。假设我们希望读取目录下一个名为 **testin** 的文本，将输出写到 **testout** 中。

2.4.1 C 语言版本

在 Flex 程序中，我们可以便捷的通过预定义的全局变量 yyin 与 yyout 来进行 IO 重定向。

在介绍重定向的方式之前，需要说明的是，**在默认情况下 yyin 和 yyout 都是绑定为 stdin 和 stdout**。而为了统一我们的输出行为也应该使用 yyout，即如样例中所写的一样，这样做还有一些其他的好处，我们会在后面提到。

在此种情况下，我们只需要对用户例程进行一些简单的修改即可，

```
int main(int argc,char **argv){
    if(argc>1){
        yyin=fopen(argv[1],"r");
        if(argc>2){
```

```
            yyout=fopen(argv[2],"w");
        }
    }
    yylex();
    fprintf(yyout,"%8d%8d%8d%8d\n",lines,words,chars,spec);
    return 0;
}
```

这里主要的功能是两个 fopen 函数，我添加了一些额外的功能，通过这样的写法，我们可以直接把文件名通过命令行传入，即一行命令

```
./lc.out testin testout
```

即可，这样可以更加灵活的控制传入的文件名，方便测试。

2.4.2 C++ 语言版本

对于 C++ 版本，yyin 与 yyout 被定义在 yyFlexLexer 类作为 protected 成员，我们不能直接访问修改，但 yyFlexLexer 提供的初始化函数其实包含 istream 和 ostream 参数，同样**在默认情况下会绑定为标准输入输出流 cin 和 cout**。我们需要做的修改如：

```
%top{
#include<fstream>
}
...
%%
...
%%
int main(){
    std::ifstream input("./testin");
    std::ofstream output("./testout");
    yyFlexLexer lexer(&input);
    lexer.yylex();
    output<<std::setw(8)<<lines<<std::setw(8)<<words<<std::setw(8)<<chars<<std::endl;
    return 0;
}
```

2.4.3 命令行输入输出流重定向

如果你对命令行有足够的了解的话，实际上我们可以选择不用上文提到的方法，而是通过简单的命令行操作将**标准输入输出流**重定向。

```
./lc.out <testin >testout
```

其中 < 操作符将标准输入重定向，> 操作符将标准输出重定向，这里看起来与之前 C 语言版本所作的修改一致，但这样的调用并不需要对代码进行任何的改动，默认情况下即可生效。这种方法对 C 语言版本和 C++ 语言版本都有效。

2.5 其他特性

2.5.1 起始状态

在定义部分，我们可以声明一些起始状态，用来限制特定规则的作用范围。用它可以很方便地做一些事情，我们用识别注释段作为一个例子，因为在注释段中，同样会包含数字字母标识符等等元素，但我们不应将其作为正常的元素来识别，这时候通过声明额外的起始状态以及规则会很有帮助。

```
...
word    [a-zA-Z]+
line \n
char    .
commentbegin "/*"
commentelement .|\n
commentend "*/"
%x COMMENT
%%
{word}   {words++;chars+=strlen(yytext);}
{line}  {lines++;}
{char}   {chars++;}
{commentbegin} {BEGIN COMMENT;}
<COMMENT>{commentelement}   {}
<COMMENT>{commentend}   {BEGIN INITIAL;}
%%
...
```

在这之中，声明部分的**%x** 声明了一个新的起始状态，而在之后的规则使用中加入 **< 状态名 >** 的表明该规则只在当前状态下生效。而状态的切换可以看出通过在之后附加的语法块中通过定义好的宏 **BEGIN** 来切换，注意初始状态默认为 **INITIAL**，因此在结束该状态时我们实际写的是切换回初始状态。

还有额外的一点说明**%x** 声明的为独占的起始状态，当处在该状态时只有规则表明为该状态的才会生效，而**%s** 可以声明共享的起始状态，当处在共享的起始状态时，没有任何状态修饰的规则也会生效。

2.5.2 行号使用

如果你有需要了解当前处理到文件的第几行，通过添加**%option yylineno**，Flex 会定义全局变量 yylineno 来记录行号，遇到换行符后自动更新，但要注意 Flex 并不会帮你做初始化，需要自行初始化。

2.5.3 符号表

对于标识符 (ID)，相同的标识符可能在相同作用域而指向相同的内存，也可能因为重新声明或在不同作用域而指向不同内存。我们希望词法程序可以对这些情况做区分。

我们定义的编译器中一定是会有一些关键字的，我们可以对每个关键字进行声明，在规则中单独找出它们，另一种思路是将所有的关键字都视作普通的符号写入符号表，通过在符号表中提前定义好相关的关键字，可以减少定义与声明的内容。

2.5.4 一个略复杂的测试程序

同学们可以用这个程序进行结果测试，从这次实验开始后续的几个实验会是相互关联的了，同学们也可以参考大作业要求自行增加或减少特性。

```
/**
 I'm a function
 */
int f() {
    int a;
    a = 0;
    while(a < 10) {
        a *= 2;
    }
    return a;
}

int main(){
    int a;
    a = 0;
    if(a==0) {
        int a;
        a=a+1;
    }
    // Comment line
    return 0;
}
```