实现词法分析器

杨俣哲 李煦阳
孙一丁 杨科迪

October 2020 to October 2021

目录

1 前言 **3**

2 实验描述 **4**
2.1 实验内容 4
2.2 实验效果示例 4
2.3 实验要求 5

3 参考流程 **6**
3.1 Flex 程序基础结构 6
3.1.1 定义部分 6
3.1.2 规则部分 7
3.1.3 用户子例程 8
3.2 C++ 版本 8
3.3 运行测试 9
3.4 输入输出流 9
3.4.1 C 语言版本 9
3.4.2 C++ 语言版本 10
3.4.3 命令行输入输出流重定向 10
3.5 其他特性 11
3.5.1 起始状态 11
3.5.2 行号使用 11
3.5.3 符号表 11

4 参考框架使用 **13**
4.1 框架目录结构 13
4.2 本次实验所需修改框架部分 13
4.3 一个略复杂测试样例可能的输出结果 15

1 前言

从本次实验开始，后续的几个实验会是相互关联的，同学们需要依次完成词法分析器、语法分析器、语义分析 (类型检查)、中间代码生成、ARM 目标代码生成五个部分，最终完成本学期编译器的大作业，使用 OJ 进行自动化评测。因此，在完成基本内容的基础上，你可以提前按照之前下发的“上机大作业要求”的进阶加分要求自行设计对应的程序了。

我们会在之后的指导书中提供**代码参考框架**。

对于参考框架，需要注意以下几点内容。

- **参考框架的使用不是必须的**，如果你觉得阅读参考框架的代码思路比较费时间，或是想按照自己的设计思路完成后续程序，我们**完全允许且鼓励**不使用给定的参考框架，自己实现。
- 参考框架只提供一定的思路，**主要的代码填充工作我们会以 “TODO” 字样表示此处需要你进行填充**，这部分填充的代码同学们自行完成，且填充代码是框架代码的主要部分。

对于后续实验，需要注意以下几点内容。

- 从词法分析开始的所有实验均为小组作业，直至最后一次作业完成汇编代码评测，共五次实验均需线下讲代码，且只需最后一次作业完整正式提交报告，内容是完整的你的编译器构建过程。
- 完成以上五个部分后，你需要在希冀在线平台 OJ[1] 上在线评测以验证正确性，评测结果是评价你编译器完成程度的最重要标准，即“上机大作业要求”文件中提到的“目标代码 (ARM) 生成、完成编译器构造部分”及“进阶加分功能实现”模块的最重要评价标准。
- 虽然需完成最后一次“ARM 目标代码生成”部分，才会使用 OJ 进行正确性评测，为保证之前模块正确性，请尽早使用所有公开测试样例进行本地测试，避免出现“在最后发现问题，推倒重来”的现象。
- 最后，一次完整的 OJ 评测过程需要近 30 分钟，由于评测机资源有限，且预计一个组完全通过最终的样例需要近 50 次提交的修改，若均堆积在最后提交，必然造成高并发导致的服务器宕机和拥塞。**因此，极其建议同学们本地测试通过后再提交，并提早计划，完成作业**。

[1] 默认用户名是自己的学号，默认密码是 2021compiler 学号后六位，如学号为 1910000，则用户名为 1910000，默认密码为 2021compiler910000。请同学们尽快登录修改默认密码，绑定邮箱，并建立自己的小组。之前已以学号注册过该平台的同学密码仍为原密码，不会覆盖。如忘记密码可用邮箱找回或联系助教重置。

2 实验描述

2.1 实验内容

本次实验，需要你根据你设计的编译器所支持的语言特性，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：本次实验中，你需要借助 Flex 完成这样一个程序，它的输入是一个 SysY 语言程序，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）。

本次实验是小组作业。

2.2 实验效果示例

效果如下例，以下是输入的 SysY 语言程序：

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

你本次实验构造的语法分析器读取上述输入后，一个可能的输出结果为

```
INT             int
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。

2.3 实验要求

基本要求

- 按照上述实验内容及实验效果示例，借助 Flex 工具实现词法分析器。
- 无需撰写完整研究报告，但需要在雨课堂上将本实验的所有程序源码（无需包含编译出的可执行文件）打包为 zip 压缩文件提交。
- 上机课时，以小组为单位，线下讲程序（主要流程是：本次实验内容结果演示、阐述小组详细分工、助教针对实验内容进行提问）

课外探索及思考

你能否设计实现一个 Flex 工具，或实现其流程中的主要算法？即完成如下步骤：

- 设计实现正则表达式到 NFA 的转换程序 (可借助 Bison 工具)；
- 设计实现 NFA 到 DFA 的转换程序；
- 设计实现 DFA 化简的程序
- 实现模拟 DFA 运转的程序 (将前三步转换的 DFA 与标准的模拟运行算法融合起来)。

注意，本次实验中该“课外探索及思考”部分不影响成绩，只用作给有余力的同学练习。

3 参考流程

3.1 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。

3.1.1 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。

3.1.2 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。

3.1.3 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。

3.2 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，但大同小异。

```
%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. */
```

3.3 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。

3.4 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。

3.4.1 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。

3.4.2 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;
}
```

3.4.3 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。

3.5 其他特性

3.5.1 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。

3.5.2 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。

3.5.3 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。

4 参考框架使用

事先再次强调，参考框架的使用不是强制的！

你完全可以按照自己的想法，设计实现自己更好更有创意的框架！

本次框架下载链接，链接的 Readme.md 会给出相对更详细的对框架的解释。

4.1 框架目录结构

我们本次没有给出后续的完整框架。

本次给出的 src 目录下，**lexer.l 是我们本次词法分析实验需要着重关注并修改的文件，即 flex 的输入文件**。

test 目录下主要存放测试样例，大家可以在支持了更多的语言特性后，从最终的测试样例仓库下载更多的测试样例，放在对应目录下。同样你也可以自定义测试样例，需要注意保证 SysY 语言文件的后缀名为.sy 即可。

4.2 本次实验所需修改框架部分

为了能让框架中的代码大部分重用，框架定义了一个宏 ONLY_FOR_LEX，在 lexer.l 的第 6 行，首先需要为其解注释。

TODO：为宏 ONLY_FOR_LEX 解注释。

```
%option noyywrap
%{
    /*
    * You will need to comment this line in lab5.
    */
    #define ONLY_FOR_LEX
```

这是为了能让被 #ifdef ONLY_FOR_LEX，#else 和 #endif 之间包围的代码块被条件编译。本次实验你主要需要在 #ifdef ONLY_FOR_LEX 和 #else 之间填充代码，而在 #else 和 #endif 之间的代码块在下次实验中有用，主要用于 lex 和 yacc 文件间的信息传递。

具体如何进行信息传递呢? 如果你还有印象，请回忆我们在上次实验中是如何识别标识符的呢。我们手动实现并定义了 yylex，在特定规则下读入，在特定规则下又要使用 ungetc 进行回退，最后识别出的结果给 yyparse 用于识别各词素。比如 yylex 中识别的是一个标识符 ID，我们上次实验中在识别后就会 return ID，ID 则用于给 yyparse 匹配 CFG 进行归约操作。

我们在 makefile 定义了 testlabfour 动作，在 makefile 所在路径下的终端输入 make testlabfour 并执行，你可以看到 test 路径下输出了对应样例的结果，但这个输出和我们指导书最开始给的输出格式不一样，比如 ID 行均没有输出，对应的属性给出的框架中也没有保存，这是因为我们给的框架里没有实现缺失部分的功能，这部分需要我们之后补充。

TODO：执行 make testlabfour，了解当前框架词法分析模块已完成和缺失的功能，执行后可使用 make cleanlabfour 对输出进行清除。

在第一个%% 之前，你会定义一些需要定义正则式。我们已经给出了一部分，当然有很多词素是缺失的，如八进制数字、十六进制数字、行注释的定义等我们都没有给出，缺失的部分需要你对照“上机大作业要求”中要求的语言特性自行补全。

TODO：补全缺失的正则定义。

```
%}

EOL (\r\n|\n|\r)
WHITE [\t ]

%x BLOCKCOMMENT

%%
```

在第一个%% 和第二个%% 之间，你需要定义识别到这些词素进行的语义动作。

以一个关键字 int 为例，我们一般比较希望特殊识别出关键词 (保留字)，将其与 id 相区分开。在本次实验中我们只需要输出所有单词、词素和属性的对应关系 (**输出不是目的，目的是证明你的程序捕获并保存到了这种关系**)。因此，我们这里给出最简单的实现方式，填上 DEBUG_FOR_LAB4("INT tint"), 这个函数只完成一个简单的输出工作。

```
%%

"int" {
    /*
    * Questions:
    *   Q1: Why we need to return INT in further labs?
    *   Q2: What is "INT" actually?
    */
    #ifdef ONLY_FOR_LEX
        DEBUG_FOR_LAB4("INT\tint");
    #else
        return INT;
    #endif
}
```

DEBUG_FOR_LAB4 将我们传递的字符串由 yyout 输出。但 yyout 究竟指向哪里呢? 查看 lexer.l 的最后，main 函数中的定义。yyin 和 yyout 分别由命令行传递的参数 (argv) 捕获输入文件和输出文件名称，这也解释了最开始如何进行批量测试的问题。

```
    if(!(yyin = fopen(argv[1], "r"))){
        fprintf(stderr, "No such file or directory: %s", argv[1]);
        exit(EXIT_FAILURE);
    }

    if(!(yyout = fopen(argv[3], "w"))){
```

```
        fprintf(stderr, "No such file or directory: %s", argv[3]);
        exit(EXIT_FAILURE);
    }
```

TODO：你可以参考“int”的规则，完成其他大部分终结符的规则定义。

有一些比较特殊的语义动作，不需要语法分析的协助，我们就可以在词法分析的规则定义中完成。如八进制、十六进制数字识别后，我们可以直接转换成十进制数字存储，这部分的完成可以直接调用 sprintf 函数，给以合适的参数完成。

TODO：完成八进制数字、十六进制数字的规则定义，将其保存为十进制输出。

再比如识别标识符后，我们通常需要区分在不同作用域内的相同标识符。但遗憾的是，我们若只通过普通的词法分析，甚至不能了解标识符是定义还是使用。

在上次的实验中，笔者希望同学们完成自己对符号表的构想，那么在本次实验中，你可以将你的构想化为现实。实现自己的 SymbolTable 类，识别到标识符将其存入符号表中，最后再一次性输出。只不过在本次实验中，可能你没办法在符号表中记录足够的信息，以搜索到定义该标识符的位置。但你在本次实验中，你完全可以区分该标识符出现时所在的作用域。可参考例子。

当然，若只为完成此次实验，你也可以通过添加某些属性 (如行号) 以区分相同标识符在不同作用域内的多次出现。

TODO：完成标识符相关规则定义，你至少应对相同标识符的多次出现存入不同的属性。

TODO(optional)：完成符号表。此时可能需要用 yacc 文件辅助完成完整的语法分析，这也是下次实验的主要内容。

我们还需要识别注释，行注释和块注释都是必要的。我们给出行注释的正则式及规则定义，块注释的定义请同学们自行完成。

```
...
LINECOMMENT \/\/[^\n]*
...
{LINECOMMENT}
...
```

TODO：完成注释的正则定义和规则定义。

若你正确完成了上述所有子功能，你可以尝试使用 make testlabfour 进行测试，它会自动测试所有 test/lab4 路径下以 sy 结尾的样例，对应输出。

TODO：对所有的测试样例，得到期望的输出格式。

4.3 一个略复杂测试样例可能的输出结果

以下给出仓库中样例 2.sy 一个可能的输出结果，**不一定要和此结果一样，输出必要的信息即可：单词、词素、行号**。

token	lexeme	lineno	offset	pointer_to_scope
INT	int	3	0	
ID	f	3	4	0x55e47145a290
LPAREN	(	3	5	
RPAREN	)	3	6	
LBRACE	{	3	7	
INT	int	4	1	
ID	a	4	5	0x55e47145e610
SEMICOLON	;	4	6	
ID	a	5	1	0x55e47145e610
ASSIGN	=	5	3	
NUM	0	5	5	
SEMICOLON	;	5	6	
WHILE	while	6	1	
LPAREN	(	6	6	
ID	a	6	7	0x55e47145e610
LT	<	6	8	
NUM	10	6	9	
RPAREN	)	6	11	
LBRACE	{	6	12	
ID	a	7	2	0x55e47145e940
ASSIGN	=	7	4	
ID	a	7	6	0x55e47145e940
MULT	*	7	8	
NUM	2345	7	10	
SEMICOLON	;	7	14	
RBRACE	}	8	1	
RETURN	return	9	1	
ID	a	9	8	0x55e47145e610
SEMICOLON	;	9	9	
RBRACE	}	10	0	
INT	int	12	0	
ID	main	12	4	0x55e47145a290
LPAREN	(	12	8	
RPAREN	)	12	9	
LBRACE	{	12	10	
INT	int	13	1	
ID	_testid	13	5	0x55e47145ea60
SEMICOLON	;	13	12	
ID	_testid	14	1	0x55e47145ea60
ASSIGN	=	14	9	

```
       NUM           0      14      11
 SEMICOLON           ;      14      12
        IF          if      15       1
    LPAREN           (      15       3
        ID     _testid      15       4    0x55e47145ea60
        EQ          ==      15      11
       NUM           0      15      13
    RPAREN           )      15      14
    LBRACE           {      15      15
       INT         int      16       2
        ID     _testid      16       6    0x55e47145eb60
 SEMICOLON           ;      16      13
       INT         int      17       2
        ID    _testid2      17       6    0x55e47145eb60
 SEMICOLON           ;      17      14
        ID    _testid1      18       2    0x55e47145eb60
    ASSIGN           =      18      10
        ID    _testid1      18      11    0x55e47145eb60
      PLUS           +      18      19
       NUM           1      18      20
 SEMICOLON           ;      18      21
    RBRACE           }      19       1
    LBRACE           {      23       1
       INT         int      24       2
        ID           a      24       6    0x55e47145ecd0
 SEMICOLON           ;      24       7
    RBRACE           }      25       1
    LBRACE           {      26       1
       INT         int      27       2
        ID           a      27       6    0x55e47145ed70
 SEMICOLON           ;      27       7
    RBRACE           }      28       1
       INT         int      29       1
        ID           b      29       5    0x55e47145ea60
 SEMICOLON           ;      29       6
    RETURN      return      30       1
       NUM           0      30       8
 SEMICOLON           ;      30       9
    RBRACE           }      31       0
```