实现词法分析器

杨俣哲 李煦阳
孙一丁 杨科迪
时浩铭
杨科迪 韩佳迅

2020 年 10 月—2023 年 9 月

目录

1 实验描述

1.1 实验内容

本次实验，需要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 语言源程序，它的输出是每一个文法单元的类别、词素、行号、列号，以及必要的属性。比如，对于 `DECIMAL` 会有属于它的“数值”属性，对于 `ID` 会有它的符号表项。

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原理大作业，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。

1.2 实验效果示例

以下是输入的 SysY 语言程序：

```
int a;

int main()
{
    int a;
    a = 1 + 2;
    if(a < 5)
        return 1;
    return 0;
}
```

你本次实验构造的词法分析器读取上述输入后，一个可能的输出结果为：

```
INT         int         0    0
ID          a           0    4    0x55e47145a290
SEMICOLON   ;           0    5
INT         int         2    0
ID          main        2    4    0x55e47145e610
LPAREN      (           2    8
RPAREN      )           2    9
LBRACE      {           3    0
INT         int         4    4
ID          a           4    8    0x55e47145e940
SEMICOLON   ;           4    9
ID          a           5    4    0x55e47145e940
```

```
ASSIGN      =           5    6
DECIMAL     1           5    8    1
ADD         +           5    10
DECIMAL     2           5    12   2
SEMICOLON   ;           5    13
IF          if          6    4
LPAREN      (           6    6
ID          a           6    7    0x55e47145e940
LESS        <           6    9
DECIMAL     5           6    11   5
RPAREN      )           6    12
RETURN      return      7    8
DECIMAL     1           7    15   1
SEMICOLON   ;           7    16
RETURN      return      8    4
DECIMAL     0           8    11   0
SEMICOLON   ;           8    12
RBRACE      }           9    0
```

其中每列分别为单词、词素、行号、列号、属性（DECIMAL 的属性为数值，ID 的属性为符号表项指针）。

1.3 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，但需要在雨课堂上提交 GitLab 链接或者程序源码打包后的 zip 压缩文件（无需包含编译出的可执行文件）；
- 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。

2 Flex 编程简介

2.1 Flex 程序基础结构

一个简单的 Flex 程序结构如下：

```
%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%% 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. 匹配尽可能长的字符串——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，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. */
```

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;
}
```

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输入输出的文件，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下：

```
%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。

3 实验流程

3.1 前言

我们会提供实验的**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，自己完成本学期的编译实验。
- 参考框架只提供一定的思路，我们会以注释的形式给出主要的代码填充提示，同学们需要自行完成这部分代码。

对于后续实验，需要注意以下几点内容：

- 从实现词法分析器直至最后目标代码生成共五次实验均为小组作业，均需线下讲代码，且只需最后一次作业提交正式报告，内容是你的编译器完整的构建过程。
- 由于本学期的实验为小组作业，请使用希冀平台 gitlab进行版本控制与协作开发，并注意不要将代码仓库公开，我们会通过提交记录评判同学们的分工与时间分配情况。
- 完成最后一次实验“ARM 目标代码生成”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。
- 虽然在完成最后一次实验之后，才会使用 OJ 进行正确性评测，为保证之前各个实验正确性，请尽早使用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，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**。

3.2 代码框架

本次实验框架代码的目录结构如下：

```
./
├── include
│   └── common.h
└── src
```

[1]用户名是自己的学号，默认密码是 2023compiler 学号，如学号为 2110000，则用户名为 2110000，默认密码为 2023compiler2110000。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。

```
    ├── lexer.l
    └── main.cpp
├── sysyruntimelibrary
├── test
├── .gitignore
├── example.sy
├── Makefile
└── README.md
```

src 目录下的 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 目录中包含所有的测试用例，sysyruntimelibrary 目录下的文件为 SysY 语言的运行时库，Makefile 中包含编译、测试本次实验的命令，README.md 会对其用法进行详细的介绍。

你可以通过以下命令获取框架代码：

```
git clone https://github.com/shm0214/2023NKUCS-Compilers-Lab.git
git switch lab3
git remote rename origin framework
git remote add origin <url>
```

其中，url 为你们小组代码仓库的地址。我们会随时更新框架代码，你应该注意使用如下命令获取更新并合并到代码中：

```
git pull framework lab3
```

3.3 任务

1. 实现符号表。对于标识符 (ID)，它的属性为符号表项 (Symbol Table Entry)，同名标识符在相同作用域可能指向相同的符号表项，也可能因为在不同作用域的重新声明而指向不同符号表项。我们希望词法程序可以对这些情况做区分，这需要设计符号表 (Symbol Table)，虽然目前符号表项还只是词素、作用域等简单内容，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；
2. 完成整形常量的词法分析。你需要定义八进制和十六进制的规则，将其保存为十进制输出；
3. 完成浮点型常量的词法分析。你需要定义浮点型常量的规则，将其保存为 float 类型进行输出；
4. 完成单行注释和多行注释的词法分析；
5. 完成其他终结符的词法分析；
6. 对所有的测试用例，输出其中每个单词的类别、词素、行号、列号和属性，且均能得到期望的输出格式。

3.4 提示

1. 如果在 Flex 编程的过程中遇到问题，你应该注意查阅Flex 手册；

2. 你需要参考SysY 语言定义设计各个终结符的模式；
3. 你应该注意 Flex 模式匹配的顺序和最长前缀原则（2.1.2）；
4. 对于符号表，你可以定义 SymbolTable 类，通过 map 实现标识符到符号表项的映射，符号表应具有插入与查询的功能：当声明标识符时，向符号表中插入其对应的符号表项；当使用标识符时，在符号表中查找其符号表项；
5. 在词法分析阶段，我们不容易区分识别出的标识符是定义还是使用，你可以通过全局变量设置上下文对此进行区分；
6. 为了支持同名标识符在不同作用域的多次声明，你可以通过栈来改进你的符号表，栈顶为当前作用域的符号表，进入新的作用域时入栈，退出当前作用域时出栈；
7. 我们定义的编译器中一定是会有一些关键字的（如 SysY 语言运行时库），我们可以对每个关键字定义模式，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，在符号表中提前定义好关键字对应的符号表项；
8. 对于整型常量，你可以参考 ISO/IEC 9899 中整型常量的定义，在此基础上忽略所有后缀；
9. 对于浮点型常量，你可以参考 ISO/IEC 9899 中浮点型常量的定义，在此基础上忽略所有后缀；
10. 对于单行注释，SysY 语言的定义为：以序列‘//’开始，直到换行符结束，不包括换行符。你可以很容易地设计模式对单行注释进行匹配；
11. 对于多行注释，SysY 语言的定义为：以序列‘/*’开始，直到**第一次**出现‘*/’时结束，包括结束处。检查你设计的模式能否正确处理以下情况：

```
// */                          // comment, not syntax error
f = g/**//h;                   // equivalent to f = g / h;
/*//*/ l();                    // equivalent to l();
m = n//**/o
+ p;                           // equivalent to m = n + p;
/* comment */ a = b + c */ // equivalent to a = b + c */
```

如果你觉得正确实现关于注释的词法分析存在困难，你可以参考2.5.1和 Flex 手册中的开始条件。