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
华志远 李帅东 李君龙
唐显达

2020 年 10 月—2025 年 10 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、属性、行号、列号。其中类别指的就是该文法单元的类型，比如关键字、标识符、常量等。词素指的是该文法单元在源程序中出现的字符串，属性指的是该文法单元的附加信息，比如整型常量的属性为其数值，标识符的属性为符号表项指针等。行号、列号指的是该文法单元在源程序中出现的位置。在本次实验中，暂时不要求创建符号表项，你可以直接输出标识符的词素作为其属性。

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、代码优化、ARM/RISC-V 目标代码生成六个部分，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 INT_MIN = -2147483648;

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：

```
Token           Lexeme          Property        Line    Column
INT             int                             1       0
IDENT           INT_MIN         INT_MIN         1       4
ASSIGN          =                               1       12
MINUS           -                               1       14
LL_CONST        2147483648      2147483648      1       15
SEMICOLON       ;                               1       25
INT             int                             3       0
IDENT           main            main            3       4
LPAREN          (                               3       8
```

```
RPAREN          )                       3       9
LBRACE          {                       4       0
INT             int                     5       4
IDENT           a               a       5       8
SEMICOLON       ;                       5       9
IDENT           a               a       6       4
ASSIGN          =                       6       6
INT_CONST       1               1       6       8
PLUS            +                       6       10
INT_CONST       2               2       6       12
SEMICOLON       ;                       6       13
IF              if                      7       4
LPAREN          (                       7       6
IDENT           a               a       7       7
LT              <                       7       9
INT_CONST       5               5       7       11
RPAREN          )                       7       12
RETURN          return                  8       8
INT_CONST       1               1       8       15
SEMICOLON       ;                       8       16
RETURN          return                  9       4
INT_CONST       0               0       9       11
SEMICOLON       ;                       9       12
RBRACE          }                       10      0
```

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（gitlog 需要能体现出小组分工以及实验完成过程）；
- 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。此处的 yyFlexLexer 实际上是一个宏定义，它会被替换为 `prefix + class`，

因此只要我们对这一宏重定义即可实现词法分析器类名的修改，避免与其他词法分析器冲突。

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;
}
```

又或者，yyFlexLexer 基类提供了 `switch_streams` 成员函数，你也可以在创建对象后调用该函数来切换输入输出流：

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。但你仍然需要获取框架中的 testcase/ 目录与 test.py 测试脚本，在中间代码生成及后续实验中你需要保证你的代码能通过测试脚本的测试。
- 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六次实验均为小组作业，均需线下讲解代码。
- 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。同时在课程结束前也不要将项目在其他网站公开。如果你的公开仓库被发现，我们会扣除你当次实验的部分分数，并且如果有两组之间抄袭被发现，无论是谁抄袭谁，都会按 0 分处理，请同学们注意保管好自己的代码。
- 完成最后一次实验“ARM/RISC-V 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/RISC-V）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**。

[1]用户名是自己的学号，默认密码是 0x656d757374614e，如学号为 2345678，则用户名为 2345678，默认密码为 0x656d757374614e。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。

3.2 代码框架

今年提供的实验代码框架位于 github，不根据后端选择的不同产生差异，使用统一的代码框架。你可以使用以下命令获取实验框架，**请注意，在实验开始前，你需要先仔细阅读框架中的 README.md 文档，框架如何使用等信息均写在了文档中**

```
git clone https://github.com/CentaureaHO/NKU-Compiler2025.git
cd NKU-Compiler2025
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。如果框架仓库有更新，会在课程群里进行通知，请大家注意课程群内信息并及时 merge 代码。

Makefile 中已经配置好了各文件的依赖关系，你只需要简单的使用 make 指令就可以完成编译和链接工作。当你修改了 Flex / Bison 文件后，使用 make 时也会自动调用 Flex / Bison 重新生成对应的 C++ 源码。在第一次实验中，你主要需要关心下面这些文件：

```
framework
├── main.cpp
├── frontend
│   └── parser
│       ├── lexer.l // 在此处为 token 定义词法规则
│       ├── parser.y // 在此处定义你需要识别的 token
│       ├── parser.{h,cpp} // 接口类，此处做 flex/bison 的 token 信息转换，了解即可
│       └── scanner.h
└── interfaces
    └── frontend
        ├── iparser.h
        └── token.h // 在此声明你的 token 类
```

这里面实际上做了两类 token 的定义，一类是 flex / bison 运行过程中使用到的 token，另一类就是框架自行定义的 token 类。我们并不需要 bison 生成的 token 类中的所有信息，所以在中间做一层转换，拿到满足本次实验要求的 token 信息即可。假设你成功编译后想要对项目根目录下的 test.sy 进行测试，使用如下命令：

```
bin/compiler -lexer -o test.out test.sy
# 查看 test.out 来检验你的词法分析实现
# 在中间代码生成之前，框架不会提供测试脚本，需要同学们自行检验实现的正确性
```

3.3 任务

1. 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；
2. 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；
3. 完成单行注释和多行注释的词法分析；

4. 完成其他终结符的词法分析；
5. 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；
5. 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；
6. 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；
7. 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 * d / e // equivalent to a = b + c * d / e
```

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。

3.5 注意事项

1. 首先再次提醒一遍学术诚信问题，无论你是参考助教提供的代码，还是参考往年学长学姐的代码，必须在代码中加上参考代码的链接以及参考了哪一部分，否则会按抄袭进行处理。
2. 虽然我们提供了代码框架，只需要在指定的部分填充代码即可，但是框架已有的代码也是要求读懂的。不要按照框架给的示例然后生搬硬套就完成了实验，这样你会在后续的实验中感到明显困难。同时这样做，在线下检查时也很难回答出助教的问题。
3. 尽早开始你的工作，词法分析和语法分析的代码量均不会超过 500 行，但是类型检查和中间代码生成单次实验的代码量就有 2000-3000 行，同时你要学习较多的算法以及阅读较多的框架内容。所以不要因为 ddl 还很久就一直拖延，后续我们会提前讲解后面的实验内容，卡着 ddl 完成你可能在临近期末时面临非常大的代码压力。
4. 词法分析和语法分析没有自动测试脚本，所以你需要自己进行一些完善的测试来确保你程序的正确性，防止你在中间代码生成中调试很长时间结果最后发现是词法分析的错误。一些基本的情况你可以查看 testcase/lexer/目录下的测试用例与期望输出。

3.6 线下检查提问示例

1. 你是如何解析-2147483648 这个数字的，是解析为一个负号和一个整型常量，还是直接解析为一个整型常量，说明理由。
2. 你在词法分析中是否使用到了 yyleng, yytext 等变量，说明它们的作用。
3. 你在词法分析中 return 的枚举类型是在哪被定义的，这些 return 的返回值在哪里被使用到了。
4. 以 0x.AP-3 为例描述十六进制浮点数格式，并说明你如何处理该浮点常量的。
5. 关键字和标识符的处理可不可以在 frontend/parser/lexer.l 文件中交换书写顺序，说明理由。
6. 框架或者你是如何将输入文本传给 flex 进行处理的，用到了什么变量或者函数。