词法分析&语法分析　实习指导

许畅 陈嘉 朱晓瑞

> Syntax and vocabulary are overwhelming constraints—the rules that run us. Language is using us to talk—we think we are using the language, but language is doing the thinking, we're its slavish agents.
>
> ——Harry Mathews

本次实习是所有四次实习中最简单的一次，没有之一。

为什么简单？因为词法分析和语法分析这两块可以说是整个编译器当中被自动化（automated）得最好的部分。也就是说，即使没有任何的理论基础，在掌握了工具的用法以后也可以在短的时间内做出功能很全很棒的词法分析器和语法分析器。当然，不要把这句话理解为词法和语法部分的理论基础不重要或者不需要掌握——恰恰相反，这一部分被认为是计算机理论在工程实践中最成功的应用之一，对于它的介绍也将是编译理论课中的绝对重点。只不过这篇指导内容的重点，不在于理论而在于工具的使用而已。

这篇指导分将词法分析工具 GNU Flex 和语法分析工具 GNU Bison 分开来进行介绍。如前文所述，能够完成这次实习并不需要太多的理论基础，因此你大可以不必等到理论课的第三章和第四章全部结束才着手开始完成实习任务——只要你看完并掌握了这篇文章中的绝大部分内容，完成本次实习应当是极其轻松的。

词法分析

词法分析器（Lexical analyzer，也称为 scanner）的主要任务是将输入文件中的字符流（character stream）组织成为词法单元流（token stream），在某些字符不符合程序设计语言的词法规范时它也要有能力报告错误。词法分析器是编译器的所有模块中唯一一个读入并处理了输入文件中每一个字符的模块，它使得后面的语法分析阶段能够在更高一级的抽象层次上去进行而不必纠结于字符串处理这样的细节问题。

高级程序设计语言大多采用英文作为输入方式，而英文有一个非常好的性质就是它比较容易断词：紧紧相邻的英文字母一定属于同一个词，而字母与字母之间插入任何非字母的字符（例如空格、运算符等）就可以将一个词断成两个词。判断一个词是否符合语言本身的词法规范也相对简单，一个最直接的办法是：我们可以事先开一张搜索表，将所有符合词法规范的字符串都存放在表里，每次我们从输入文件中断出一个词之后，通过查这张表就可以判断该词究竟合法还是不合法。

正因为词法分析任务的难度不高，在实用的编译器中它常常是手工写成而非使用工具生成的。例如，我们下面要介绍的这个工具 GNU Flex 原先就是为了为 GCC 进行词法分析而被开发出来的，但在 4.0 版本之后 GCC 的词法分析器已经一律改为手写了。不过，本次实习要求大家使用工具来做，而词法分析器生成工具所基于的理论基础，是计算理论里最入门的内容——正则表达式（regular expression）与有限状态自动机（finite state automata）。

一个正则表达式由特定字符串构成，或者由其他正则表达式通过以下三种运算得到：

- 并运算（union）：两个正则表达式 r 和 s 的并记作 r|s，意为 r 或 s 都可以被接受
- 连接运算（concatenation）：两个正则表达式 r 和 s 的连接记作 rs，意为 r 之后紧跟 s 才可以被接受
- Kleene 闭包（Kleene closure）：一个正则表达式 r 的 Kleene 闭包记作 r*，它表示 ε|r|rr|rrr|…

有关正则表达式的内容，大家在之前的课程中都已经接触过。正则表达式之所以被人们广泛应用，一方面是因为它在表达力足够强（基本上可以表示所有的词法规则）的同时还易于被人所书写和理解，另一方面也是因为判断一个字符串是否被一个特定的正则表达式识别可以做到非常高效（在线性时间内即可完成）：我们可以将一个正则表达式转化为一个 NFA，将这个 NFA 转化为一个 DFA，最后还可以对转化好的 DFA 进行化简，之后我们就可以通过直接模拟这个 DFA 的运行来对输入串进行判断了。具体 NFA 和 DFA 指的是什么，以及如何进行正则表达式、NFA 和 DFA 之间的转化等，请自行参考教材或者去上理论课。这里我们只需要知道，前面提到的所有转化以及识别工作，都是可以由工具自动完成的。我们所需要做的，仅仅是为工具提供作为词法规范的正则表达式——对，就这么简单。

说到这里，接下来就轮到我们的工具华丽登场了。

GNU Flex 介绍

Flex 的前身为 Lex。Lex 是 1975 年由 Mike Lesk 和当时还在 AT&T 做暑期实习的 Eric Schmidt 共同完成的一款基于 Unix 环境的词法分析器的生成工具。虽说 Lex 非常出名而且被广泛使用，但它的低效和诸多 bug 还是让人们用得很不爽（顺便八卦一下，负责了 Lex 大部分源码编写的原作者 Schmidt 后来在工业界混得风生水起，至本文截稿前他仍在担任 Google 的首席执行官）。后来伯克利实验室的 Vern Paxson 使用 C 语言重写 Lex 并将这个新程序命名为 Flex（意为 Fast Lexical Analyzer Generator）。无论在效率上还是在稳定性上，Flex 都远远好于它的前辈 Lex。我们在 Linux 下使用的是 Flex 在 GNU License 下的版本，称作 GNU Flex。

GNU Flex 在 Linux 下的安装非常简单。你可以去它的官方网站上下载安装包自行安装，不过在基于 Debian 的 Linux 系统下更简单的安装方法是直接在命令行敲入如下命令：

```
sudo apt-get install flex
```

虽说版本不一样，但 GNU Flex 的基本使用方法和教材上所介绍的 Lex 没有什么不同。首先，我们需要自行完成包括词法规则等在内的 Flex 代码。如何编写这份代码后面会提到，现在先假设这份写好的代码名 lexical.l；随后，我们使用 Flex 对这份代码进行编译：

```
flex lexical.l
```

编译好的结果会保存在当前目录下的 lex.yy.c 文件中。打开这个文件你会发现，该文件本质上就是一个 C 语言的源代码。事实上，这份源代码里目前对我们有用的函数只有一个，叫做 yylex()，该函数的作用就是读取输入文件中的一个词法单元。我们可以再为它编写一个 main 函数：

```
int main(int argc, char** argv)
{
    if (argc > 1)
    {
        if (!(yyin = fopen(argv[1], "r")))
        {
            perror(argv[1]);
            return 1;
        }
    }
    while (yylex() != 0) ;
    return 0;
}
```

这个 main 函数通过命令行读入若干个参数，并取第一个参数为其输入文件名尝试打开输入文件。如果打开文件失败则退出，而如果成功则调用 yylex()进行词法分析。其中，变量 yyin 是 Flex 内部使用的一个变量，代表输入文件的文件指针，如果我们不去设置它那么 Flex 会将它自动设置为 stdin（注意如果你将 main 函数独立成了一个文件，则需要声明 yyin 为外部变量：extern FILE* yyin）。

将这个 main 函数单独放到一个文件 main.c 中（你也可以直接放到 lexical.l 中的用户自定义代码部分，这样就可以不必声明 yyin；你甚至可以不写 main 函数，Flex 会自动给你配一个，但不推荐这么做），然后编译这两个 C 源文件，我们将输出程序命名为 scanner：

```
gcc main.c lex.yy.c -lfl -o scanner
```

注意编译命令中的“-lfl”参数一定不能少，否则 GCC 会因缺少库函数而报错。之后我们就可以使用这个 scanner 程序进行词法分析了。例如，想要对一个测试文件 test.cmm 进行词法分析，只需要在命令行输入：

```
./scanner test.cmm
```

就可以得到你想要的结果。

编写 Flex 源代码

上面介绍的是使用 Flex 创建词法分析器的基本步骤。在整个创建过程中，最重要的文件无疑你所编写的 Flex 源代码，它完全决定了你所生成的词法分析器的一切行为。接下来这段教程将指导你如何去编写 Flex 源代码。

Flex 输入文件包括 3 个部分，通过“%%”隔开，如下图所示：

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

第一部分为定义部分，实际上就是在给某些后面可能经常用到的正则表达式取一个别名，从而简化词法规则的书写。定义部分的格式一般为

```
name    definition
```

其中 name 是名字，definition 可以是任意的正则表达式（正则表达式该如何书写后面会专门提及）。例如，下面这段代码定义了两个名字：digit 和 letter，前者代表 0 到 9 中的任意一个数字字符，后者代表任意一个小写字母、大写字母或是下划线：

```
...
digit     [0-9]
letter    [_a-zA-Z]
%%
...
%%
...
```

Flex 源代码的第二部分为规则部分。它由正则表达式和相应的响应函数组成，格式为：

```
pattern    {action}
```

pattern 为正则表达式，其书写规则与声明部分的正则表达式相同。action 为将要进行的具体操作，这些操作可以用一段 C 代码表示。Flex 将按照这一部分给出的内容**依次**尝试每一个规则，尽可能匹配**最长**的输入串。如果有些内容不匹配任何规则，那么 Flex 默认只将其拷贝到标准输出，想要修改这个默认行为的话只需要在所有规则最后加上一条“.”（匹配任何输入）规则，然后在对应的 action 部分书写你想要的行为既可。

思考：这里稍微将话题岔开一下。我们在理论课上学过有关词法分析中的“最长输入串匹配”，这个匹配规则可以保证我们在读入类似“<=”的输入时能将其作为一个“小于或等于”词法单元，而不是一个“小于”后面接一个“等于”这样两个词法单元。但是，最长输入串匹配规则在某些情况下并不是完美的。也就是说，存在这样的输入串，该输入串完全符合词法规范，但却不能被采用最长输入串匹配规则的词法分析器（例如由 Flex 生成的词法分析器）所接受。你能想到一个这样的例子吗？

例如，下面这段代码在遇到输入文件中包含一串数字时，会将该数字串转化为整数值并印到屏幕上：

```
...
digit     [0-9]
%%
{digit}*     {
                printf("Integer value %d\n", atoi(yytext));
             }
...
%%
...
```

其中变量 yytext 的类型为 char*，它是 Flex 为我们提供的一个变量，里面保存了当前词法单元所对应的词素。而函

数 atoi()的作用把一个字符串表示的整数转化为 int 类型。

Flex 源代码的第三部分为用户自定义代码部分。这一部分代码会原封不动地拷贝到 lex.yy.c 中，方便用户自定义需要执行的函数（之前我们提到过的 main 函数也可以写在这里）。值得一提的是，如果用户想要对这一部分所用到的变量、函数或者头文件进行声明，可以在定义部分（也就是 Flex 源代码的第一部分）之前使用“%{”和“%}”符号将要声明的内容添加进去。被“%{”和“%}”所包围的内容也会一并拷贝到 lex.yy.c 的最前面。

下面通过一个简单的例子来说明 Flex 源代码如何书写[1]。我们知道 Unix/Linux 下有一个常用的文字统计工具 wc，它可以统计一个或者多个文件中的（英文）字符数、单词数和行数。使用 Flex 来可以让我们快速地写出一个类似的文字统计程序：

```
%{
    /* 此处省略#include 部分 */
    int chars = 0;
    int words = 0;
    int lines = 0;
%}
letter    [a-zA-Z]

%%

{letter}+    { words++; chars+= yyleng; }
\n           { chars++; lines++; }
.            { chars++; }

%%

int main(int argc, char** argv)
{
    if (argc > 1)
    {
        if (!(yyin = fopen(argv[1], “r”)))
        {
            perror(argv[1]);
            return 1;
        }
    }
    yylex();
    printf(“%8d%8d%8d\n”, lines, words, chars);
    return 0;
}
```

其中 yyleng 是 Flex 为我们提供的变量，你可以将其理解为 strlen(yytext)。我们用变量 chars 记录输入文件的字符数、words 记录单词数、lines 记录行数。上面这段程序应当很好理解：每遇到一个换行符就把行数加一，每识别出一个单词就把单词数加一（字符数同时加上单词长度），每读入一个字符就把字符数加一。最后在 main 函数中把 chars、words 和 lines 全部打印出来。需要注意的是，由于规则部分里我们没有让 yylex()返回某个值，因此在 main 函数中调用 yylex()时可以不套外层那个 while 循环。

真正的 wc 工具可以一次传入多个参数从而统计多个文件。为了能够让 Flex 源程序对多个文件进行统计，我们可以修改 main 函数的实现如下：

[1] 这个例子来源于 John Levine 的著作 *flex&bision*

```
int main(int argc, char** argv)
{
    int i, totchars = 0, totwords = 0, totlines = 0;
    if (argc < 2) {  /* just read stdin */
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
        return 0;
    }

    for (i=1;i<argc;i++) {
        FILE *f = fopen(argv[i], "r");
        if (!f) {
            perror(argv[i]);
            return 1;
        }
        yyrestart(f);
        yylex();
        fclose(f);
        printf("%8d%8d%8d %s\n", lines, words, chars, argv[i]);
        totchars += chars; chars = 0;
        totwords += words; words = 0;
        totlines += lines; lines = 0;
    }

    if (argc > 1)
        printf("%8d%8d%8d total\n", totlines, totwords, totchars);
    return 0;
}
```

其中 yyrestart(f)函数是 Flex 提供的库函数，它可以让 Flex 将其输入文件指针 yyin 设置为 f（你也可以像前面一样手动令 yyin = f）并重新初始化文件指针，令其指向输入文件的开头。

书写正则表达式

Flex 源代码中无论是定义部分还是规则部分，正则表达式无疑在其中扮演了相当重要的作用。那么，如何在 Flex 源代码中书写正则表达式呢？

- 符号“.”匹配换行符（\n）之外的任何一个字符
- 符号“[]”匹配一个字符类（character class），即方括号之内只要有一个字符被匹配上了那么我们就认为被方括号括起来的整个表达式都被匹配上了。例如，[0123456789]代表 0~9 中任意一个数字字符，[abcABC]代表 a、b、c 三个字母的小写或者大写。方括号中还可以使用连字符“-”代表一个范围，例如[0123456789]也可以直接写作[0-9]，而所有小写字母字符亦可直接写成[a-z][2]。如果方括号中的第一个字符是“^”则代表对该字符类取补，即方括号之内如果没有任何一个字符被匹配上了那么我们就认为被方括号括起来的整个表达式都被匹配上了。例如，[^_0-9a-zA-Z]代表所有的非字母、数字以及下划线的字符。
- 符号“^”用在方括号之外则会匹配一行的开头，符号“$”用于匹配一行的结尾，符号“<<EOF>>”用于匹配文件的结尾。

[2]注意“-”代表的范围基于字符集的编码顺序，例如在 ASCII 字符集下[A-z]会匹配所有的大小写字母，因为在 ASCII 字符集中字母的编号是连续的，而且小写字母排在大写字母之后。但是，如果你真的要匹配所有的大小写字母，考虑到代码可读性的因素，不建议写成[A-z]这种形式。

- 符号“{}”含义比较特殊。如果花括号之内包含了一个或者两个数字，则代表花括号之前的那个表达式需要出现的次数。例如，A{5}会匹配 AAAAA，A{1,3}则会匹配 A、AA 或者 AAA。如果花括号之内是一个在 Flex 源代码的定义部分定义过的名字,则代表那个名字对应的正则表达式。例如,假如在定义部分定义 letter 为[a-zA-Z]，则{letter}{1,3}代表连续的一至三个英文字母。
- 符号“*”代表 Kleene 闭包操作，匹配零个或者多个表达式。例如{letter}*代表没有或者多个英文字母。
- 符号“+”代表正闭包（positive closure）操作，匹配一个或者多个表达式。例如{letter}+代表一个或者多个英文字母
- 符号“?”匹配零个或者一个表达式。例如表达式 -?[0-9]+ 表示前面带一个可选的负号的数字串。无论是*、+还是?，它们都只对与其最邻近的那个字符生效。例如 abc+代表 ab 后面跟一个或多个 c 而不代表一个或者多个 abc。如果你要匹配后者，需要使用小括号“()”将几个字符括起来：(abc)+
- 符号“|”代表并操作，匹配其之前或者之后的任一表达式。例如，faith|hope|charity 代表这三个串中的任何一个
- 符号“\”用于输入各种转义字符，这与 C 语言字符串里“\”的用法类似。例如，“\n”代表换行，“\t”代表制表符，“*”代表星号，“\\”代表字符“\”等等。
- 符号“"”（英文引号）将逐字匹配被引起来的内容（无视各种特殊符号以及转义字符）。例如表达式"…"就代表三个点而不代表三个除换行以外的任意字符。
- 符号“/”会查看输入字符的上下文，x/y 会识别 x 仅当在输入文件中 x 之后紧跟 y。例如，0/1 可以匹配输入串 01 中的 0 但不匹配输入串 02 中的 0。
- 任何不属于上面介绍过的有特殊含义的字符在正则表达式中都仅匹配这个字符本身。

下面我们通过几个例子来演练一下 Flex 里正则表达式的书写：

- 带一个可选的正号或者负号的数字串可以这样写：[-+]?[0-9]+
- 带一个可选的正号或者负号以及一个可选的小数点的数字串则要困难一些，考虑下面几种写法：
  - [-+]?[0-9.]+ 会匹配太多额外的模式，像 1.2.3.4
  - [-+]?[0-9]+\.?[0-9]+ 会漏掉某些模式，像 12.或者.12
  - [-+]?[0-9]*\.?[0-9]+ 会漏掉 12.
  - [-+]?[0-9]+\.?[0-9]* 会漏掉.12
  - [-+]?[0-9]*\.?[0-9]* 会多匹配空串或者只有一个小数点的串

  正确的写法是 [-+]?([0-9]*\.?[0-9]+|[0-9]+\.)。注意不要把这个表达式直接抄到你的实习代码中——第一次实习对于浮点数的识别要求和这个例子是有区别的。
- 假设我们现在在做一个汇编器。目标机器的 CPU 中有 32 个寄存器，编号为 0…31。在汇编源码中可以使用 r 后面加一个或两个数字的方式来表示某一个寄存器，例如 r15 代表 15 号寄存器，r0 或 r00 代表第 0 号寄存器，r7 或者 r07 代表 7 号寄存器等等。现在我们希望把汇编源码中所有代表寄存器的词法单元都识别出来。考虑下面几种写法：
  - r[0-9]+ 的确会匹配 r0、r15，但同时它也会匹配 r99999——目前世界上还不存在哪个 CPU 能有一百万个寄存器。
  - r[0-9]{1,2} 同样会匹配一些额外的模式，例如 r32、r48 等。
  - r([0-2][0-9]?|[4-9]|(3(0|1)?)) 这种写法是正确的，但可读性就比较差了。
  - 正确性毋庸置疑而且可读性最好的写法应该是：r0|r00|r1|r01|r2|r02|r3|r03|r4|r04|r5|r05|r6|r06|r7|r07|r8|r08|r9|r09|r10|r11|r12|r13|r14|r15|r16|r17|r18|r19|r20|r21|r22|r23|r24|r25|r26|r27|r28|r29|r30|r31 ，但是这样写的话可扩展性又非常差——如果目标机器上有 128 甚至 256 个寄存器呢？

  思考：上面两种正确的写法里，哪一种写法生成的词法分析器时间效率更高？空间效率呢？

Flex 中的一些高级特性（选读）

读过前面的这些内容之后，你已经完全有能力使用 Flex 完成实习 1 的词法分析部分了。下面所列出的一些 Flex 里面的特性能让你在使用 Flex 的过程中感到更方便、更灵活，但其中的很多特性你可能这辈子都用不到。因此，如

果你对这部分内容不感兴趣，完全可以跳过这一节，这不会对你完成实习的必做部分产生任何负面的影响。

❖ **yylineno 选项**

我们在编写编译器的过程中，经常会需要记录行号以便在报错时提示用户输入文件的哪一行出现了问题。为了能够记录这个行号，我们当然可以自己定义某个变量，例如 int lines，来记录当前词法分析器读到了输入文件的哪一行。每当识别出模式“\n”，我们就让 lines = lines+1。

实际上，Flex 内部已经为我们提供了类似的变量，叫做 yylineno。我们不必去操心维护 yylineno 的值，它会在每行结束自动加一。不过，默认状态下它并不直接开放给用户使用。如果我们想要读取 yylineno 的值，需要在 Flex 源码的定义部分加入语句“%option yylineno”。

需要说明的是，虽然 yylineno 会自动增加，但当我们在词法分析过程中调用 yyrestart()函数读取另一个文件时它却不会重新被初始化，因此如果有需要用户需要自行添加初始化语句 yylineno = 1。

❖ **输入缓冲区**

我们在课本上所学到的词法分析器的工作原理都是在模拟一个 DFA 的运行。这个 DFA 每次读入一个字符，然后根据状态之间的转换关系决定下一步应该转换到哪个状态。事实上，实用的词法分析器很少会从输入文件里逐个字符进行读入，因为这样做需要进行大量磁盘操作，效率太低。更加高效的办法是一次读入一大段输入字符并且将其保存在程序专门为其在内存中申请的输入缓冲区中。

在 Flex 里，所有的输入缓冲区都有一个共同的类型，叫做 YY_BUFFER_STATE。你可以通过 yy_create_buffer()函数为一个特定的输入文件开辟一块输入缓冲区，例如：

```
YY_BUFFER_STATE bp;
FILE* f;

f = fopen(…, “r”);
bp = yy_create_buffer(f, YY_BUF_SIZE);
yy_switch_to_buffer(bp);
…
yy_flush_buffer(bp);
…
yy_delete_buffer(bp);
```

其中 YY_BUF_SIZE 是 Flex 内部的一个常数，如果你很好奇它是多少的话可以将其值打印出来。通过调用 yy_switch_to_buffer()函数可以让词法分析器到指定的缓冲区里读数据，调用 yy_flush_buffer()函数可以清空缓冲区中的内容，而调用 yy_delete_buffer()可以删除一个缓冲区。

如果你的词法分析器要支持文件与文件之间的相互引用（例如 C 语言中的#include），你可能会在词法分析的过程中频繁地使用 yyrestart()切换当前的输入文件。在切换到其他输入文件再切换回来之后，为了能继续之前的词法分析任务，你需要无损地保留原先的输入缓冲区内容，这就牵扯到使用一个栈来暂存当前输入文件的缓冲区了。虽然 Flex 有提供相关的函数帮助你做这件事情，但这些函数的功能相对比较羸弱，还不如我们自己手写的好。

实际上，原先我们的想法是将实现#include 以及#define 的相关功能作为选做内容加入到实习任务中。不过考虑到这两个功能实现起来并不轻松，于是后来就去掉了。有兴趣的同学可以自己思考一下应该如何实现这些功能。

❖ **一些比较好用的 Flex 库函数**

input()函数可以从当前的输入中读入一个字符，这有助于你不借助正则表达式来实现某些功能。例如，下面这段代码在输入文件中发现双斜线“//”后将从当前字符开始一直到行尾的所有字符全部丢弃掉：

```
%%
“//”    {
            char c = input();
            while (c != ‘\n’) c = input();
         }
```

注意，对于需要实现分组 1.3 的同学，这段代码可能不符合你想实现的功能，请仔细考虑。

unput(char c)函数可以将指定的字符放回输入缓冲区中。这对于宏定义等功能的实现来说是很方便的。例如，假设之前定义过一个宏#define BUFFER_LEN 1024，则当输入文件中遇到了字串 BUFFER_LEN 时，下面这段代码将该

宏所对应的内容放回输入缓冲：

```
char* p = macro_contents("BUFFER_LEN");          // p = "1024"
char* q = p + strlen(p);
while (q > p) unput(*--q);                  // push back right-to-left
```

yyless(int n)函数可以将刚从输入里读取的 yyleng–n 个字符放回到输入里，yymore()函数可以告诉 Flex 保留当前词素并在下一个词法单元被识别出来以后将下一个词素连接到当前词素的后面。配合使用 yyless()和 yymore()函数可以方便地处理那些边界难以界定的模式。例如，我们在书写字符串字面量的正则表达式时往往会写成由一对双引号引起来的所有内容\"[^\"]*\"，但有的时候被双引号引起来的内容里面也可能出现跟在转义符号之后的双引号，例如"This is an \"example\""。那么如何使用 Flex 处理这种情况呢？方法之一就是借助于 yyless 和 yymore：

```
%%
\"[^\"]*\"  {
                if (yytext[yyleng-2] == '\\') {
                    yyless(yyleng-1);
                    yymore();
                } else {
                    /* process the string literal */
                }
            }
```

REJECT 宏可以帮助我们识别那些互相重叠的模式。当我们执行 REJECT 之后，Flex 会进行一系列操作，这些操作的结果相当于将 yytext 放回输入之内，然后去试图匹配当前规则之后的那些规则。例如，下面这段代码：

```
%%
pink    { npink++; REJECT; }
ink     { nink++; REJECT; }
pin     { npin++; REJECT; }
```

会统计输入文件中所有的 pink、ink 和 pin 出现的个数，即使这三个单词之间互有重叠。

Flex 的特性远不止上面介绍的这些，要了解其更多的特性请参考 Flex 的在线文档。

完成本次实习——步骤 1

为了完成本次实习，首先你需要阅读 C--的语法规范（Grammar.pdf），以及对于这个语法的补充说明（附在实习要求 1 的最后）。该规范中与词法相关的内容都在 Tokens 那一节里，除了 INT、FLOAT 和 ID 三个词法单元需要你自行为其书写正则表达式之外，剩下的词法单元都没有任何的难度。

阅读完规范，对 C--的词法有一个大概了解以后，你就可以动手编写 Flex 源代码了。在敲入所有的词法之后，为了能检验你的词法分析器是否工作正常，你可以暂时向屏幕打印当前的词法单元的名称，例如：

```
%%
"+"     { printf("PLUS\n"); }
"-"     { printf("SUB\n"); }
"&&"    { printf("AND\n"); }
"||"    { printf("OR\n"); }
...
```

为了能够报告错误类型 A，你可以在所有规则的最后增加类似于这样的一条规则：

```
%%
...
.   {
        printf("Error type A at line %d: Mysterious character \'%s\'\n", yylineno, yytext);
    }
```

完成 Flex 源代码的编写之后，使用前面介绍过的方法将其编译出来，就可以自己书写一些小规模的测试数据来测试你的词法分析程序了。一定确保你的词法分析器的正确性！如果词法分析这里出了问题没有检查出来，到了后面语法分析发现了前面出了问题再回头调试，那会为原本简单的实习增加许多不必要的麻烦。为了使你在编写 Flex 源码时少走弯路，以下几条建议请仔细阅读：

- 留神空格和回车的使用。如果不注意，有时很容易让本应是空白符的空格或者回车变成正则表达式的一部分，有时又很容易让本应是正则表达式一部分的空格或回车变成 Flex 源码里的空白符。
- 正则表达式和其所对应的语义动作之间，永远不要插入空行。
- 如果对正则表达式中的运算符优先级有疑问，那就不要吝啬使用括号来确保正则表达式的优先级确实是你所想要的。
- 使用花括号括起每一段语义动作，即使该语义动作只包含有 1 行代码。
- 在 definition 部分我们可以为许多正则表达式取别名，这一点要好好利用。别名可以让后面的正则表达式更加容易阅读、扩展和调试。
- 在正则表达式中引用之前定义过的某个别名（例如 digit）时，时刻谨记该别名一定要用花括号{}括起来。

到这里为止，如果你认为自己的词法分析器没有问题了，那就请继续阅读下面的语法分析部分；如果你认为自己在某些地方还有点疑问，或者尚未完成词法分析部分，请重新阅读前文并抓紧时间完成。

> The whole problem with the world is that fools and fanatics are always so certain of themselves, and wiser people so full of doubts. ——Bertrand Russell

语法分析

词法分析的下一个阶段是语法分析。语法分析器（Syntax analyzor，也称为 parser）的主要任务是读入词法单元流、判断输入程序是否匹配程序设计语言的语法模型，并在匹配规范的情况下构建起输入程序的静态结构。语法分析使得编译器的后续阶段看到的输入程序不再是一串字符流或者单词流，而是一个结构整齐、处理方便的数据对象。

语法分析问题和词法分析问题有很多相似之处：它们的理论基础都是形式语言理论，它们都是计算机理论在工程实践中最成功的应用，它们都能被高效地（更具体地说，线性时间内）完成，它们的构建都可以被工具自动化地完成。不过，由于语法分析问题本身要比词法分析复杂得多，手写一个语法分析器的代价实在太大，故目前绝大多数实用的编译器在语法分析这里都是使用工具帮忙的。

正则表达式难以进行任意大的计数，故很多在程序设计语言中常见的结构（例如匹配的括号）根本没有办法使用正则文法进行表示。为了能够有效地对常见的语法结构进行表示，人们使用了比正则文法表达能力更强的上下文无关文法（Context Free Grammar, CFG）。然而，上下文无关文法虽然在表达能力上强于正则语言，但判断某个输入串是否属于特定的 CFG 的时间效率最好的算法也要 $O(n^3)$，这样的效率让人难以接受。因此，现代程序设计语言的语法大多都属于一般 CFG 的一个足够大的子集，比较常见的子集有 LL(k)文法以及 LR(k)文法，判断一个输入是否属于这两种文法都只需要线性时间。

上下文无关文法 G 在形式上是一个四元组：终结符号（也就是词法单元）集合 T、非终结符号集合 NT、初始符号 S 以及产生式集合 P。产生式集合 P 是一个文法的核心，它通过产生式定义了一系列推导的规则，从初始符号出发基于这些产生式经过不断地将非终结符替换为其他非终结符以及终结符，即可得到一串符合语法规约的词法单元。这个替换和推导的过程可以使用树形结构表示，称作语法树。事实上，语法分析的过程就是把词法单元流变成语法树的过程。尽管在之前曾经出现过各式各样的算法，但目前最常见的构建这棵语法树的技术只有两种：自顶向下方法和自底向上方法。我们下面将要介绍的工具 Bison 所生成的语法分析器就采用了自底向上的 LALR(1)分析技术（通过一定的设置还可以让 Bison 使用另一种被称为 GLR 的分析技术，不过对该技术的介绍已经超出了本课程的范围），而其他的某些语法分析工具，例如基于 Java 语言的 JTB，生成的语法分析器则是采用了自顶向下的 LL(1)分析技术。当然，具体的工具采用了哪一种技术这种细节对于工具的使用者来讲都是完全屏蔽的。和词法分析器的生成工具一样，工具的使用者所要做的仅仅是将输入程序的程序设计语言的语法告诉语法分析器生成工具，工具虽然不能显式地帮我们构造出一棵语法树，但我们可以通过在语法的产生式中插入语义动作这种更灵活的形式实现一些甚至比语法树生成更加复杂的功能。

GNU Bison 介绍

Bison 的前身为基于 Unix 的 Yacc。令人惊讶的是，Yacc 的发布时间甚至比 Lex 还要早。Yacc 所采用的 LR 分析技术的理论基础早在 50 年代就已经由 Knuth 逐步建立了起来，而 Yacc 本身则是贝尔实验室的 S.C.Johnson 基于这些理论在 75 年到 78 年写成的。到了 1985 年当时在 UC Berkeley 的一个研究生 Bob Corbett 在 BSD 下重写了 Yacc，后来 GNU project 接管了这个项目，为其增加了许多新的特性，于是就有了我们今天所用的 GNU Bison。

GNU Bison 在 Linux 下的安装非常简单。你可以去它的官方网站上下载安装包自行安装，基于 Debian 的 Linux 系统下更简单的方法同样是直接在命令行敲入如下命令：

```
sudo apt-get install bison
```

虽说版本不一样，但 GNU Bison 的基本使用方法和教材上所介绍的 Yacc 没有什么不同。首先，我们需要自行完成包括语法规则等在内的 Bison 代码。如何编写这份代码后面会提到，现在先假设这份写好的代码名 syntax.y；随后，我们使用 Bison 对这份代码进行编译：

```
bison syntax.y
```

编译好的结果会保存在当前目录下的 syntax.yy.c 文件中。打开这个文件你会发现，该文件本质上就是一个 C 语言的源代码。事实上，这份源代码里目前对我们有用的函数只有一个，叫做 yyparse()，该函数的作用就是对输入文件进行语法分析，如果分析成功没有错误则返回 0，否则返回非 0。不过，只有这个 yyparse()函数还不足以让我们的程序跑起来——前面说过，语法分析器的输入是一个个的词法单元，那么 Bison 通过什么方式来获得这些词法单元呢？事实上，Bison 在这里需要用户为它提供另外一个专门返回词法单元的函数，这个函数名叫 yylex()。

Flex 与 Bison 联合编译

yylex()相当于嵌在 Bison 里的词法分析器。这个函数可以由用户自行实现，但是因为我们之前已经使用 Flex 生成了一个 yylex()函数，能不能让 Bison 使用 Flex 生成过的 yylex()函数呢？答案是肯定的。

仍然以 Bison 源代码 syntax.y 为例。首先，为了能够使用 Flex 中的各种函数，需要在 Bison 源代码中引用 lex.yy.c：

```
#include “lex.yy.c”
```

随后在使用 Bison 编译这份代码时，我们需要加上“-d”参数：

```
bison -d syntax.y
```

这个参数的含义是，将编译的结果分拆成 syntax.tab.c 和 syntax.tab.h 两个文件，其中.h 文件里包含着一些词法单元的类型定义之类的内容。得到这个.h 文件以后，下一步是修改我们的 Flex 源代码 lexical.l，增加对 syntax.tab.h 的引用，并且让 Flex 源码中规则部分的每一条 action 都返回相应的词法单元，如下图所示：

```
%{
    #include “syntax.tab.h”
    …
%}
…
%%
“+”     { return PLUS; }
“-”     { return SUB; }
“&&”    { return AND; }
“||”    { return OR; }
…
```

其中，返回值 PLUS、SUB 等都是在 Bison 源代码中定义过的词法单元（如何定义它们后文会提到）。由于我们刚刚修改了 lexical.l，需要重新将它编译出来：

```
flex lexical.l
```

接下来是重写我们的 main 函数。由于 Bison 会在需要时自动调用 yylex()，我们在 main 函数中也就不需要调用它了。不过，Bison 是不会自己调用 yyparse()和 yyrestart()的，因此这两个函数仍需要我们在 main 函数中显式地进行调用：

```
int main(int argc, char** argv)
{
    if (argc <= 1) return 1;
    FILE* f = fopen(argv[1], "r");
    if (!f)
    {
        perror(argv[1]);
        return 1;
    }
    yyrestart(f);
    yyparse();
    return 0;
}
```

现在我们有了 3 个 C 语言源文件：main.c、lex.yy.c 以及 syntax.tab.c，其中 lex.yy.c 已经被 syntax.tab.c 引用了，因此我们最后要做的就是把 main.c 和 syntax.tab.c 放到一起进行编译：

```
gcc main.c syntax.tab.c -lfl -ly -o parser
```

其中“-lfl”不要省略，否则 GCC 会因缺少库函数而报错，但“-ly”这里一般情况下可以省略。现在我们就可以使用这个 parser 程序进行语法分析了。例如，想要对一个测试文件 test.cmm 进行语法分析，只需要在命令行输入：

```
./parser test.cmm
```

就可以得到你想要的结果。

编写 Bison 源代码

上面介绍的是使用 Flex 和 Bison 联合创建语法分析器的基本步骤。在整个创建过程中，最重要的文件无疑你所编写的 Flex 源代码和 Bison 源代码，它完全决定了你所生成的语法分析器的一切行为。Flex 源代码如何进行编写前面已经介绍过了，接下来这段教程将指导你如何去编写 Bison 源代码。

同 Flex 源码一样，Bison 源代码也分为三个部分，其作用与 Flex 代码大致相同：第一部分是声明部分，所有词法单元的定义都可以放到这里；第二部分是规则部分，其中包括具体的语法和相应的语义动作；第三部分是用户函数部分，这一部分代码会原封不动地拷贝到 syntax.tab.c 中，方便用户自定义需要执行的函数（main 函数也可以写在这里，不过同样不推荐这么做）。值得一提的是，如果用户想要对这一部分所用到的变量、函数或者头文件进行声明，可以在定义部分（也就是 Bison 源代码的第一部分）之前使用“%{”和“%}”符号将要声明的内容添加进去。被“%{”和“%}”所包围的内容也会一并拷贝到 syntax.tab.c 的最前面。

下面我们通过一个例子来对 Bison 源码的结构进行解释。一个在控制台运行的可以进行整数四则运算的小程序，其语法如下图所示（这里假设词法单元 INT 代表 Flex 识别出来的一个整数， ADD 代表加号+，SUB 代表减号－，MUL 代表乘号*，DIV 代表除号/）：

```
Calc    →    ε
        |    Exp
Exp     →    Factor
        |    Exp ADD Factor
        |    Exp SUB Factor
Factor  →    Term
        |    Factor MUL Term
        |    Factor DIV Term
Term    →    INT
```

这个程序完整的 Bison 代码为：

```
%{
    #include <stdio.h>
%}

/* declared tokens */
%token INT
%token ADD SUB MUL DIV

%%
Calc        :   /* empty */
            |   Exp                 { printf("= %d\n", $1); }
            ;
Exp         :   Factor
            |   Exp ADD Factor      { $$ = $1 + $3; }
            |   Exp SUB Factor  { $$ = $1 - $3; }
            ;
```

```
Factor      :   Term
            |   Factor MUL Term     { $$ = $1 * $3; }
            |   Factor DIV Term     { $$ = $1 / $3; }
            ;
Term        :   INT
            ;

%%
#include “lex.yy.c”
int main() {
    yyparse();
}

yyerror(char* msg) {
    fprintf(stderr, “error: %s\n”, msg);
}
```

这段 Bison 代码以%{…%}开头，被%{…%}包含的内容主要是对 stdio.h 的引用。接下来是一些以%token 开头的词法单元（终结符）定义，如果你需要采用 Flex 生成的 yylex()的话，那么在这里定义的词法单元都可以作为 Flex 源代码里的返回值。与终结符相对地，所有未被定义为%token 的符号都会被看作非终结符，这些非终结符要求必须在任意产生式的左边至少出现一次。

接下来的第二部分就是书写产生式的地方。第一个产生式左边的非终结符默认为初始符号（你也可以通过在定义部分添加%start X 来将另外的某个非终结符 X 指定为初始符号）。产生式中的箭头在这里用冒号（:）表示，一组产生式和另一组之间以分号（;）隔开。产生式中无论是终结符还是非终结符都各自对应一个属性值，产生式左边的非终结符对应的属性值用$$表示，右边的几个符号的属性值按从左到右的顺序依次表示为$1、$2、$3……每一条产生式的最后可以添加一组以花括号{}括起来的语义动作，这组语义动作会在整条产生式归约完成**之后**执行，如果不明确指定语义动作，那么 Bison 将采用默认的语义动作{ $$ = $1 }。语义动作也可以放在产生式的中间，例如 A→B {…} C，这样的写法等价于 A→BMC，M→ε{…}，其中 M 为额外引入的一个非终结符。需要注意的是，在产生式中间添加语义动作在某些情况下有可能会在原有语法中引入冲突，因此使用的时候要特别谨慎。

看到这里，不知道你的心里有没有产生一个疑问：每一个非终结符的属性值都可以通过它所产生的那些终结符或者非终结符的属性值计算出来，但是终结符本身的属性值如何得到呢？答案是在 yylex()函数中得到。因为我们的 yylex()函数是由 Flex 源代码生成的，因此要想让终结符带有属性值，就必须回头修改 Flex 源代码。假设在我们的 Flex 源代码中，INT 词法单元对应着一个数字串，那么我们可以将 Flex 源码修改为：

```
…
digit     [0-9]
%%
{digit}*     {
                yylval = atoi(yytext);
                return INT;
             }
…
%%
…
```

变量 yylval 是 Flex 的内部变量，意为当前词法单元所对应的属性值。我们只需要将这个变量赋成 atoi(yytext)就可以将词法单元 INT 的属性值设置为它所对应的整数值了。

回到之前的 Bison 代码中。代码的用户自定义函数部分我们写了两个函数：一个很简单的只调用了 yyparse()的 main 函数以及另一个没有返回类型并带有一个字符串参数的 yyerror()的函数。yyerror()函数是 Bison 提供的库函数，它会在你的语法分析器每发现一个语法错误时被调用，默认参数为”syntax error”。默认情况下 yyerror()只会将传入

的字符串参数打印到标准错误输出上，而你可以自己重新定义这个函数从而使它打印一些其他的东西，例如上例中我们就在参数前面多打印了“error: ”字样。

现在，编译并执行这个程序，然后在控制台输入 10-2+3，然后输入回车，最后输入 Ctrl+D 结束，你会看到屏幕上打印出了计算结果 11。

属性值的类型

不知道你发现了没有，在上面的例子中，每一个终结符以及非终结符的属性值都是 int 类型。但在我们构建语法树的过程中，我们非常希望不同的符号对应的属性值能有不同的类型，而且最好能对应任意类型而不仅仅是 int 型。这一节内容将会指导你如何在 Bison 中解决上述问题。

第一种方法是对宏 YYSTYPE 进行重定义。Bison 里会默认所有属性值的类型以及变量 yylval 的类型都是 YYSTYPE，默认情况下 YYSTYPE 被定义为 int。如果你在你的 Bison 代码的%{…%}部分加入例如这样一句话#define YYSTYPE float，那么所有属性值就都成为 float 型了。那么如何使得不同的符号对应不同的类型呢？你可以将 YYSTYPE 定义成一个联合体（union）类型，这样你就可以根据符号的不同来访问联合体中不同的域，从而实现多种类型的效果。

上面这种方法虽然可行，但在实际操作中还是稍显麻烦，因为你每次对属性值的访问都要自行指定哪个符号对应哪一个域。实际上，在 Bison 中已经内置了其他的机制来方便你对属性值类型的处理，一般而言我们还是更推荐使用这种方法而不是上面介绍的那种。

仍然还是以前面四则运算的小程序为例说明 Bison 中的属性值类型机制是如何工作的。原先这个四则运算程序只能计算整数值，现在我们加入浮点数运算的功能。修改后的语法如下图所示：

Calc	→	ε
	|	Exp
Exp	→	Factor
	|	Exp ADD Factor
	|	Exp SUB Factor
Factor	→	Term
	|	Factor MUL Term
	|	Factor DIV Term
Term	→	INT
	|	FLOAT

在这一份语法中，我们希望词法单元 INT 能有整型属性值，FLOAT 能有 float 型属性值，其他的非终结符为了简单起见我们让它们都具有 double 型的属性值。这份语法以及类型方案对应的 Bison 源代码如下：

```
%{
    #include <stdio.h>
%}

/* declared types */
%union {
    int type_int;
    float type_float;
    double type_double;
}

/* declared tokens */
%token <type_int> INT
%token <type_float> FLOAT
%token ADD SUB MUL DIV
```

```
/* declared non-terminals */
%type <type_double> Exp Factor Term

%%
Calc        :   /* empty */
            |   Exp                 { printf("= %lf\n, $1); }
            ;
Exp         :   Factor
            |   Exp ADD Factor      { $$ = $1 + $3; }
            |   Exp SUB Factor      { $$ = $1 - $3; }
            ;
Factor      :   Term
            |   Factor MUL Term     { $$ = $1 * $3; }
            |   Factor DIV Term     { $$ = $1 / $3; }
            ;
Term        :   INT                 { $$ = $1; }
            |   FLOAT               { $$ = $1; }
            ;

%%
…
```

首先，我们在定义部分的开头使用%union{…}将所有可能的类型都包含进去。接下来，在%token 部分里我们使用一对尖括号<>把需要确定属性值类型的每个词法单元所对应的类型括起来。对于那些需要指定其属性值类型的非终结符而言，我们使用%type 加上尖括号的办法确定它们的类型。当所有需要确定类型的符号的类型都被定下来之后，规则部分里的$$、$1 等就自动地带有了相应的类型，不再需要我们显示地为其指定类型了。

语法单元的位置

实习要求中需要你输出每一个语法单元出现的位置。你当然可以自己在 Flex 中定义每个行号和列号、在每一个语义动作中维护这个行号和这个列号并将它们作为属性值的一部分返回给语法单元。这种做法需要我们额外编写一些维护性的代码，让人感觉挺不方便。Bison 有没有内置的位置信息供我们使用呢？答案是肯定的。

前面介绍过 Bison 中的每一个语法单元都对应了一个属性值，在语义动作中这些属性值可以使用$$、$1、$2 等进行引用。实际上除了属性值以外，每一个语法单元还对应了一个位置信息，在语义动作中这些位置信息同样可以使用@$、@1、@2 等进行引用。位置信息的数据类型是一个 YYLTYPE，其默认的定义是：

```
typedef struct YYLTYPE
{
    int first_line;
    int first_column;
    int last_line;
    int last_column;
}
```

其中 first_line 和 first_column 分别是该语法单元对应的第一个词素出现的行号和列号，而 last_line 和 last_column 分别是该语法单元对应的最后一个词素出现的行号和列号。有了这些内容，输出位置信息时我们就显得游刃有余了。看到这里我们假设你会高高兴兴地回去修改代码，引用@1、@2 等将每一个语法单元的 first_line 打印出来，结果发现打印出来的行号全都是 1……

为什么会出现这种问题？主要原因在于，Bison 并不会替我们维护这些位置信息，我们必须在 Flex 源文件中自

行维护。看到这里你又会说，如果要我们自己去维护，那不就相当于使用本节开头介绍的那个笨方法了吗？其实，只要稍加利用 Flex 中的某些机制，维护这些信息并不需要太多的代码。我们在 Flex 源文件开头部分定义变量 yycolumn 并添加这样宏 YY_USER_ACTION：

```
%{
/* 此处省略#include 部分 */
    int yycolumn = 1;

    #define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno; \
        yylloc.first_column = yycolumn; yylloc.last_column = yycolumn + yyleng - 1; \
        yycolumn += yyleng;
%}
```

其中 yylloc 是 Flex 内置变量，代表当前词法单元所对应的位置信息；YY_USER_ACTION 宏代表在执行每一个语义动作之前需要先被执行的一段代码，默认为空，而这里我们将其改成了对位置信息的维护代码。最后还要在 Flex 源文件中做的更改就是在发现换行符以后对变量 yycolumn 进行复位：

```
...
%%
...
\n   { yycolumn = 1; }
```

好了，现在回到 Bison 中再去打印位置信息，是不是已经正常了？

思考：如果仔细考察上面对 YY_USER_ACTION 的定义，你会发现 yylloc.first_line 和 yyloc.last_line 总是被赋成一样的值。但当我们在 Bison 中打印出每一个语法单元的 first_line 和 last_line 时很多语法单元的这两个值并不相同。这是怎么回事呢？

二义性与冲突处理

Bison 的一个非常好用同时也是一个非常恼人的特性，是即使对于一个有二义性的文法，它也会有自己的一套隐式的冲突解决方案（我们知道，一旦出现归约/归约冲突，Bison 总会选择靠前的产生式；而一旦出现移入/归约冲突，则 Bison 总会选择移入）从而生成相应的语法分析器，而这些冲突解决方案在某些场合有可能并不是我们所期望的。因此，我们建议大家在使用 Bison 编译代码时要留意它所给的提示信息，如果提示文法有冲突，那么请一定对源码进行修改，尽量把所有的冲突全部消解掉。

前面我们的那个四则运算的小程序，如果它的语法变成这样：

```
Calc    →    ε
        |    Exp
Exp     →    Factor
        |    Exp ADD Exp
        |    Exp SUB Exp
...
```

虽然看起来好像没什么变化（Exp→Exp ADD/SUB Factor 现在变成了 Exp→Exp ADD/SUB Exp），但实际上前文里之所以没有这样写是因为这样做会引入额外的二义性。例如，输入为 1-2+3，程序到底是先算 1-2 呢，还是 2+3 呢？恐怕是先算前者。此时语法分析器在读到 1-2 的时候可以归约也可以移入，但由于 Bison 默认移入优先于归约，因此语法分析器会继续读入+3 然后计算 2+3。为了解决这里出现的二义性问题，要么重写语法（Exp→Exp ADD/SUB Factor 相当于强制规定加减法为左结合），要么显示地指定算符的优先级与结合性。一般而言，重写语法总是一件比较麻烦的事情，而且会引入不少像 Exp→Term 这样除了增加可读性之外没什么实质用途的产生式。所以更好的解决办法还是考虑优先级与结合性。

在 Bison 源代码中，我们可以通过%left、%right 和%nonassoc 对终结符的结合性进行规定，其中%left 表示左结

合，%right 表示右结合，%nonassoc 表示不可结合。例如，下面这段结合性的声明代码主要针对四则运算、括号以及赋值号：

```
%right   ASSIGN
%left    ADD     SUB
%left    MUL     DIV
%left    LP   RP
```

其中 ASSIGN 代表赋值号，LP 代表左括号，RP 代表右括号。实际上，Bison 规定任何排在后面的算符其优先级都要高于排在前面的算符。因此，这段代码实际上还在规定括号优先级高于乘除、乘除高于加减、加减高于赋值号。在我们实习所使用的 C--语言里，表达式 Exp 的语法便是高度冲突的，你需要模仿前面介绍的方法，根据 C--语法补充说明中的内容为运算符规定优先级和结合性，从而解决掉这些冲突。

另外一个在程序设计语言中非常常见的冲突是嵌套 if-else 所出现的冲突（也被称为“悬空 else”）。考虑 C--语言的这段语法：

```
Stmt        →    IF LP Exp RP Stmt
            |    IF LP Exp RP Stmt ELSE Stmt
```

假设我们的输入是 if (x > 0) if (x == 0) y = 0; else y = 1; ，那么语句最后的这个 else 是属于前一个 if 还是后一个 if 呢？标准 C 语言规定在这种情况下 else 总是匹配距离它最近的那个 if，这与 Bison 的默认处理方式（移入/归约冲突时总是移入）是一样的，因此即使我们不在 Bison 源代码里对这个问题进行任何处理，最后生成的语法分析器的行为也是正确的。但如果不处理，Bison 总是会提示我们的语法中存在一个移入/归约冲突，让人很不爽。有没有办法把这个冲突去掉呢？

显式地解决悬空 else 问题可以借助于算符优先级。Bison 源代码中每一条产生式后面都可以紧跟一个%prec 标记，指明该产生式的优先级等同于一个终结符。下面这段代码通过定义一个比 ELSE 优先级更低的 LOWER_THAN_ELSE 算符，降低了归约相对于移入 ELSE 的优先级：

```
…
%nonassoc   LOWER_THAN_ELSE
%nonassoc   ELSE
…
%%
…
Stmt    :   IF LP Exp RP Stmt      %prec LOWER_THAN_ELSE
        |   IF LP Exp RP Stmt ELSE Stmt
```

这里 ELSE 和 LOWER_THAN_ELSE 的结合性其实并不重要，重要的是当语法分析器读到 IF LP Exp RP 时，如果它面临归约和移入 ELSE 这两种选择，它会根据优先级自动选择移入 ELSE。通过指定优先级的办法，我们可以避免 Bison 在这里报告冲突。

思考： 如果我们要做一个奇怪的程序设计语言，每一个 else 都匹配距离它最远的那个 if，应该如何修改 Bison 代码？仅仅把 ELSE 和 LOWER_THAN_ELSE 的优先级对调一下就行了吗？

前面我们通过优先级和结合性解决了表达式和 if-else 语句里可能出现的二义性问题。事实上，有了优先级和结合性的帮助，我们几乎可以消除语法中所有的二义性——但我们强烈**不建议**大家使用它们解决除了表达式和 if-else 之外的任何冲突。原因很简单：只要是 Bison 报告的冲突，都有可能成为语法中潜在的一个缺陷，这个缺陷的来源很可能是你所定义的程序设计语言里的一些连你自己都没有意识到的语法问题。表达式和二义性这里我们之所以敢使用优先级和结合性是因为我们对冲突的来源非常了解，除此之外，只要是 Bison 认为有二义性的语法，大部分情况下这个语法让人来看肯定也会看出二义性。此时你要做的不是如何掩盖这些语法上的问题，而是应该坐下来仔细地对语法进行修订，发现并解决语法本身的问题。

我们知道，在 C++语言中，当我们声明了一个类 class MyClass {…}之后，在使用这个类定义对象 x 时可以直接写成 MyClass x;。而如果你细心观察 C 语言或者 C--语言的语法，会发现当我们声明了一个结构体，例如 struct Complex {…}之后，在使用这个结构体定义变量 x 时必须写成 struct Complex x; 而不能直接写成 Complex x; ，感觉有些不方便。现在我们想尝试修改一下 C--的语法使得后一种写法合法化：把 Specifier 的产生式修改为

```
Specifier    →    TYPE
             |    ID
             |    StructSpecifier
```

思考：这样的修改可行吗？如果不可行，会出现什么样的问题？怎样做才能不出现问题？

Bison 源代码的调试（选读）

在使用 Bison 进行编译时，如果增加-v 参数，那么 Bison 会在生成.yy.c 文件的同时帮我们多生成一个.output 文件。例如，执行

```
bison -d -v syntax.y
```

命令后，会在当前目录下发现一个新文件 syntax.output，这个文件中包含 Bison 所生成的语法分析器对应的 LALR 状态机的一些详尽的描述。如果你在使用 Bison 编译的过程中发现自己的语法里存在冲突，但你看了半天也搞不清楚到底哪里出现了冲突时，就可以去阅读这个.output 文件，里面对于每一个状态所对应的产生式、该状态何时进行移入何时进行归约、你的语法有多少冲突以及这些冲突在哪里等等都有十分完整的描述。

例如，如果我们不处理前面提到的悬空 else 问题的话，.output 文件的第一句就会是：

```
state 112 conflicts: 1 shift/reduce
```

向下翻，找到状态 112，.output 文件对该状态的描述为：

```
State 112

    36 Stmt: IF LP Exp RP Stmt .
    37       | IF LP Exp RP Stmt . ELSE Stmt

     ELSE    shift, and go to state 114

     ELSE          [reduce using rule 36 (Stmt)]
     $default    reduce using rule 36 (Stmt)
```

这里我们发现，状态 112 在读到 ELSE 时既可以移入又可以归约，而 Bison 选择了前者，将后者用方括号括了起来。知道了是这里出现问题，我们就可以以此为线索修改 Bison 源码或者重新修订语法了。

对于一个有一定规模的语法规范（如 C--语法）而言，Bison 所产生的 LALR 语法分析器可以有一百甚至几百个状态。即使将它们都输出到了.output 文件里，在这些状态里逐个寻找潜在的问题也是挺费劲的。另外，有些其它的问题，例如语法分析器在运行时刻出现“Segmentation fault”等等，很难从对状态机的静态描述中发现，必须要在动态的、交互的环境下才容易看出问题。为了达到这种效果，在使用 Bison 进行编译的时候，可以通过附加-t 参数打开其诊断模式（或者在代码中#define YYDEBUG 1）：

```
bison -d -t syntax.y
```

在 main 函数调用 yyparse()之前我们加上一句 yydebug = 1; ，重新编译整个程序。之后运行这个程序你会发现，语法分析器现在正像一个自动机一样，一个一个状态地在进行转换，并将当前状态的信息打印到标准输出上，方便你检查自己的代码中哪里出现了问题。以之前的那个四则运算小程序为例，当打开诊断模式之后运行程序，屏幕上会出现如下字样：

```
Starting parse
Entering state 0
Reading a token:
```

如果我们输入 4，会明显地看到语法分析器出现了状态转换，并将当前栈里的内容列了出来：

```
Next token is token INT ()
Shifting token INT ()
Entering state 1
Reducing stack by rule 9 (line 29):
    $1 = token INT ()
-> $$ = nterm Term ()
Stack now 0
Entering state 6
Reducing stack by rule 6 (line 25):
    $1 = nterm Term ()
-> $$ = nterm Factor ()
Stack now 0
Entering state 5
Reading a token:
```

继续敲入其他内容可以看到更进一步的状态转换。

诊断模式会使得语法分析器的性能下降不少，建议在不使用时不要随便打开。

错误恢复

当输入文件中出现语法错误的时候，Bison 总是会让它生成的语法分析器尽早地报告错误。每当语法分析器从 yylex()得到了一个词法单元，但这个当前状态并没有针对这个词法单元的动作时，就认为输入文件出现了语法错误，此时它会默认进入下面这个错误恢复模式：

- 调用 yyerror("syntax error")，只要你没有重写 yyerror()，该函数默认会在屏幕上打印出 syntax error 字样
- 从栈顶弹出所有还没有处理完的规则，直到语法分析器回到了一个可以移入特殊符号 error 的状态
- 移入 error，然后对输入的词法单元进行丢弃，直到找到一个能够跟在 error 之后的符号为止（该步骤也被称为 resynchronization）
- 如果在 error 后能成功移入三个符号，则继续正常的语法分析；否则，返回前面的步骤 2

前面这个步骤看起来似乎很复杂，但实际上要我们做的事情只有一件：在语法里指定一下 error 符号放到哪里即可。不过，这个看似简单的工作其实蛮有讲究的：一方面，我们希望 error 后面跟的内容越多越好，这样 resynchronization 就会更容易成功，这提示我们应该把 error 尽量放在高层的产生式中；另一方面，我们又希望能够丢弃尽可能少的词法单元，这提示我们应该把 error 尽量放在底层的产生式中。在实际应用中，人们一般把 error 放在例如行尾、括号结尾等地方，本质上相当于让行结束符“;”以及括号“{}”“()”等作为错误恢复的同步符号：

```
Stmt    →    error SEMI
CompSt  →    error RC
Exp     →    error RP
```

当然，上面这几个产生式仅仅是几个示例，并不意味着把这几个产生式照搬到你的 Bison 源代码中就可以让语法分析器能够满足本次实习的实习要求。你需要自己思考如何书写包含 error 的产生式才能够检查出输入文件中存在的各种语法错误——注意这并不是一项简单工作，根据目前的反馈情况看，已经有不少同学在 error 产生式的书写上面遇到问题了。

完成本次实习——步骤 2

想要做好一个语法分析器，第一步还是要仔细阅读并理解 C--的语法规范。平心而论，C--的语法要比它的词法复杂很多。虽说就算不理解语法的含义直接将产生式全部敲到 Bison 源代码里也能完成实习任务，但如果缺乏对语法的理解，在调试和测试这个语法分析器时你将会感到无所适从。另外，如果连你自己都没弄懂 C--语法中每一条产生式背后的具体含义，你又如何在下一次语义分析的实习中写程序去分析这些产生式的语义呢？

接下来，我们建议你先写一个包含所有语法产生式但不包含任何语义动作的 Bison 源代码，然后将它和修改以后的 Flex 代码、main 函数等一块编译出来先看看效果——对于一个没有语法错误的输入文件而言，这个程序应该什么也不输出；对于一个包含语法错误的输入文件而言，这个程序应该输出 syntax error。如果你的程序能够成功地判别有无语法错误，再去考虑优先级与结合性如何设置、如何进行错误恢复等问题；如果你的程序输出的结果不对，或者说你的程序根本编译不出来，那你可能要重新阅读前文并仔细检查一下到底是哪里出了问题——好在此时代码并不算多，借助于 Bison 的.output 文件以及诊断模式等我们相信要查出错来并不是什么难事。

下一步需要考虑语法树的表示和构造。语法树是一棵树，而且是一棵多叉树，因此为了能够建立它需要你去动手实现多叉树这个数据结构。你需要专门写函数完成多叉树的创建和插入操作，然后在 Bison 源文件中修改属性值的类型为多叉树类型，并添加语义动作，将产生式右边的每一个符号所对应的树节点作为产生式左边的非终结符所对应的树节点的子节点逐个进行插入。具体这棵多叉树的数据结构怎么定义、插入操作怎么完成完全取决于你自己的设计，在设计过程中有一点需要你注意的地方：在下一次实习里我们还将会在这棵语法树上进行一些其他的操作，故现在的数据结构设计将对后面语义分析部分产生较大影响。

构造完这棵树之后下一步就是按照实习要求中提到的缩进格式将它打印出来，同时要求打印的还有行号以及一些其他的信息。为了能打印这些信息，你需要专门写代码去对你生成的那棵语法树进行遍历。由于还要求打印行号，故在之前生成语法树的时候你就应该将每一个节点第一次出现时的行号都记录下来（使用位置信息@n、使用变量 yylineno 或者自己维护这个行号均可）。因为这段负责打印的代码仅仅是为了本次实习而写，今后的实习将不会复用它们，所以我们建议你将这些代码组织到一起或者写成函数接口的形式，以便在下次实习的时候直接删掉，而不需要对原有代码做较大的修改。

当然，如果你有其他的方法能够完成本次实习而且你的方法也能得到正确的结果，你大可以采用你自己的方法而不必遵循上面所介绍的思路。另外，实习时你也难免会因为碰到了一些解决不了的问题、查不出来的错误而心生惧意。不要怕！这是你应当经历的训练的一部分，请一定坚持下来。

祝你好运！

A person who never made a mistake never tried anything new. ——Albert Einstein