2.2 实验指导

词法分析和语法分析这两块，可以说是在整个编译器当中被自动化得最好的部分。也就是说，即使没有任何的理论基础，在掌握了工具的用法之后，也可以在短时间内做出功能很全很棒的词法分析程序和语法分析程序。当然这并不意味着，词法分析和语法分析部分的理论基础并不重要。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，对它的介绍也是编译理论课中的重点。但本节指导内容的重点不在于理论而在于工具的使用。

本节指导内容将分别介绍词法分析工具GNU Flex和语法分析工具GNU Bison。如前所述，完成实验一并不需要太多的理论基础，只要看完并掌握了本节的大部分内容即可完成实验一。

2.2.1 词法分析概述

词法分析程序的主要任务是将输入文件中的字符流组织成为词法单元流，在某些字符不符合程序设计语言词法规范时它也要有能力报告相应的错误。词法分析程序是编译器所有模块中唯一读入并处理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在更高抽象层次上运作，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，而英文有个非常好的性质，就是它比较容易断词：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（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中，每次我们从输入文件中断出一个词之后，通过查找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，而并非使用工具生成。例如，我们下面要介绍的这个工具GNU Flex原先就是为了帮助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，实验一要求使用工具来做，而词法分析程序生成工具所基于的理论基础，是计算理论中最入门的内容：**正则表达式（Regular Expression）**和**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，或者由其它正则表达式通过以下三种运算得到：

1) **并运算（Union）**：两个正则表达式r和s的并记作r丨s，意为r或s都可以被接受。

2) **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。

3) **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在课本的理论部分讨论过。正则表达式之所以被人们所广泛应用，一方面是因为它在表达能力足够强（基本上可以表示所有的词法规则）的同时还易于书写和理解；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接受可以做到非常高效（在线性时间内即可完成）。比如，我们可以将一个正则表达式转化为一个**NFA（即不确定的有限状态自动机）**，然后将这个NFA转化为一个**DFA（即确定的有限状态自动机）**，再将转化好的DFA进行化简，之后我们就可以通过模拟这个DFA的运行来对输入串进行识别了。具体的NFA和DFA的含义，以及如何进行正则表达式、NFA及DFA之间的转化等，请参考课本的理论部分。这里我们仅需要知道，前面所述的所有转化和识别工作，都可以由工具自动完成。而我们所需要做的，仅仅是为工具提供作为词法规范的正则表达式。

2.2.2 GNU Flex介绍

Flex的前身是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环境的词法分析程序生成工具。虽然Lex很出名并被广泛使用，但它的低效和诸多问题也使其颇受诟病。后来伯克利实验室的Vern Paxson使用C语言重写Lex，并将这个新的程序命名为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
```

虽然版本不一样，但GNU Flex的使用方法与课本上介绍的Lex基本相同。首先，我们需要自行完成包括词法规则等在内的Flex代码。至于如何编写这部分代码我们在后面会提到，现在先假设这部分写好的代码名为lexical.l。随后，我们使用Flex对该代码进行编译：

```
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。打开这个文件你就会发现，该文件本质上就是一份C语言的源代码。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：

```
1  int main(int argc, char** argv) {
2    if (argc > 1) {
3      if (!(yyin = fopen(argv[1], "r"))) {
4        perror(argv[1]);
5        return 1;
6      }
7    }
8    while (yylex() != 0);
9    return 0;
10 }
```

这个main函数通过命令行读入若干个参数，取第一个参数为其输入文件名并尝试打开该输入文件。如果文件打开失败则退出，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，表示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（即**标准输入**，通常连接到键盘）。注意，如果你将main函数独立设为一个文件，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（你也可以直接放入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，因为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”参数不可缺少，否则GCC会因为缺少库函数而报错。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了。

2.2.3 Flex：编写源代码

以上介绍的是使用Flex创建词法分析程序的基本步骤。在整个创建过程中，最重要的文件无疑是你所编写的Flex源代码，它完全决定了你的词法分析程序的一切行为。接下来我们介绍如何编写Flex源代码。

Flex源代码文件包括三个部分，由“%%”隔开，如下所示：

```
1  {definitions}
2  %%
3  {rules}
4  %%
5  {user subroutines}
```

第一部分为**定义部分**，实际上就是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是任意的正则表达式（正则表达式该如何书写后面会介绍）。例如，下面的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，后者则代表任意一个小写字母、大写字母或下划线：

```
1  …
2  digit  [0-9]
3  letter  [_a-zA-Z]
4  %%
5  …
6  %%
7  …
```

Flex源代码文件的第二部分为**规则部分**，它由正则表达式和相应的响应函数组成，其格式为：

```
pattern  {action}
```

其中pattern为正则表达式，其书写规则与前面的定义部分的正则表达式相同。而action则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，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的最后加上一条“.”（即匹配任何输入）规则，然后在其对应的action部分书写你想要的行为即可。

例如，下面这段Flex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打印到屏幕上：

```
1  …
2  digit  [0-9]
3  %%
4  {digit}+  { printf("Integer value %d\n", atoi(yytext)); }
5  …
6  %%
7  …
```

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。函数atoi()的作用是把一个字符串表示的整数转化为int类型。

Flex源代码文件的第三部分为**用户自定义代码部分**。这部分代码会被原封不动地拷贝到lex.yy.c中，以方便用户自定义所需要执行的函数（之前我们提到过的main函数也可以写在这里）。值得一提的是，如果用户想要对这部分所用到的变量、函数或者头文件进行声明，可以在前面的定义部分（即Flex源代码文件的第一部分）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源代码该如何书写[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。利用Flex我们可以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记录输入文件中的字符数、words记录单词数、lines记录行数。上面这段程序很好理解：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，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的值全部打印出来。需要注意的是，由于在规则部分里我们没有让yylex()返回任何值，因此在main函数中调用yylex()时可以不套外层的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。为了能够让这个Flex程序对多个文件进行统计，我们可以修改main函数如下：

```
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();
```

[1] 该例子来源于：《flex & bision》，John Levine著，陆军译，东南大学出版社，第29页，2011年。

```
16      fclose(f);
17      printf("%8d%8d%8d %s\n", lines, words, chars, argv[i]);
18      totchars += chars; chars = 0;
19      totwords += words; words = 0;
20      totlines += lines; lines = 0;
21    }
22    if (argc > 1)
23      printf("%8d%8d%8d total\n", totlines, totwords, totchars);
24    return 0;
25  }
```

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

2.2.4 Flex：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，正则表达式都在其中扮演了重要的作用。那么，如何在Flex源代码中书写正则表达式呢？我们下面介绍一些规则：

1) 符号“.”匹配除换行符“\n”之外的任何一个字符。

2) 符号“[”和“]”共同匹配一个字符类，即方括号之内只要有一个字符被匹配上了，那么被方括号括起来的整个表达式都被匹配上了。例如，[0123456789]表示0~9中任意一个数字字符，[abcABC]表示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“-”表示一个范围，例如[0123456789]也可以直接写作[0-9]，而所有小写字母字符也可直接写成[a-z] 。如果方括号中的第一个字符是“^”，则表示对这个字符类取补，即方括号之内如果没有任何一个字符被匹配上，那么被方括号括起来的整个表达式就认为被匹配上了。例如，[^_0-9a-zA-Z]表示所有的非字母、数字以及下划线的字符。

3) 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。

4) 符号“{”和“}”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，则表示那个名字对应的正则表达式。例如，在定义部分如果定义了letter为[a-zA-Z]，则{letter}{1,3}表示连续的一至三个英文字母。

5) 符号“*”为**Kleene闭包**操作，匹配零个或者多个表达式。例如{letter}*表示零个或者多个英文字母。

6) 符号“+”为**正闭包**操作，匹配一个或者多个表达式。例如{letter}+表示一个或者多个

英文字母。

7) 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?，它们都只对其最邻近的那个字符生效。例如abc+表示ab后面跟一个或多个c，而不表示一个或者多个abc。如果你要匹配后者，则需要使用小括号“(”和“)”将这几个字符括起来：(abc)+。

8) 符号“|”为**选择**操作，匹配其之前或之后的任一表达式。例如，faith | hope | charity表示这三个串中的任何一个。

9) 符号“\”用于表示各种转义字符，与C语言字符串里“\”的用法类似。例如，“\n”表示换行，“\t”表示制表符，“*”表示星号，“\\”代表字符“\”等。

10) 符号“"”（英文引号）将逐字匹配被引起来的内容（即无视各种特殊符号及转义字符）。例如，表达式"..."就表示三个点而不表示三个除换行符以外的任意字符。

11) 符号“/”会查看输入字符的上下文，例如，x/y识别x仅当在输入文件中x之后紧跟着y，0/1可以匹配输入串01中的0但不匹配输入串02中的0。

12) 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该字符本身。

下面我们通过几个例子来练习一下Flex源代码里正则表达式的书写：

1) 带一个可选的正号或者负号的数字串，可以这样写：[+–]?[0-9]+。

2) 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，表示起来要困难一些，可以考虑下面几种写法：

a) [+–]?[0-9.]+会匹配太多额外的模式，像1.2.3.4；

b) [+–]?[0-9]+\.?[0-9]+会漏掉某些模式，像12.或者.12；

c) [+–]?[0-9]*\.?[0-9]+会漏掉12.；

d) [+–]?[0-9]+\.?[0-9]*会漏掉.12；

e) [+–]?[0-9]*\.?[0-9]*会多匹配空串或者只有一个小数点的串；

f) 正确的写法是：[+–]?([0-9]*\.?[0-9]+|[0-9]+\.)。

3) 假设我们现在做一个汇编器，目标机器的CPU中有32个寄存器，编号为0…31。汇编源代码可以使用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表示第15号寄存器，r0或r00表示第0号寄存器，r7或者r07表示第7号寄存器等。现在我们希望识别汇编源代码中所有可能的寄存器表示，可以考虑下面几种写法：

g) r[0-9]+可以匹配r0和r15，但它也会匹配r99999，目前世界上还不存在CPU能拥有一百万个寄存器。

h) r[0-9]{1,2}同样会匹配一些额外的表示，例如r32和r48等。

i) r([0-2][0-9]?|[4-9]|(3(0|1)?))是正确的写法，但其可读性比较差。

j) 正确性毋庸置疑并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？

2.2.5 Flex：高级特性

前面介绍的Flex内容已足够帮助完成实验一的词法分析部分。下面介绍一些Flex里面的高级特性，能让你在使用Flex的过程中感到更方便和灵活。这部分内容是可选的，跳过也不会对实验一的完成产生负面的影响。

yylineno选项：

在写编译器程序的过程中，经常会需要记录行号，以便在报错时提示输入文件的哪一行出现了问题。为了能记录这个行号，我们可以自己定义某个变量，例如lines，来记录词法分析程序当前读到了输入文件的哪一行。每当识别出“\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，默认状态下它并不开放给用户使用。如果我们想要读取yylineno的值，则需要在Flex源代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()函数读取另一个输入文件时它却不会重新被初始化，因此我们需要自行添加初始化语句yylineno = 1。

输入缓冲区：

课本上介绍的词法分析程序其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，实用的词法分析程序很少会从输入文件中逐个读入字符，因为这样需要进行大量的磁盘操作，效率较低。更加高效的办法是一次读入一大段输入字符，并将其保存在专门的输入缓冲区中。

在Flex中，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：

```
1  YY_BUFFER_STATE bp;
2  FILE* f;
3  f = fopen(…, "r");
4  bp = yy_create_buffer(f, YY_BUF_SIZE);
5  yy_switch_to_buffer(bp);
6  …
7  yy_flush_buffer(bp);
8  …
9  yy_delete_buffer(bp);
```

其中YY_BUF_SIZE是Flex内部的一个常数，通过调用yy_switch_to_buffer()函数可以让词法分析程序到指定的输入缓冲区中读字符，调用yy_flush_buffer()函数可以清空缓冲区中的内容，而调用yy_delete_buffer()则可以删除一个缓冲区。

如果你的词法分析程序要支持文件与文件之间的相互引用（例如C语言中的#include），你可能需要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输入缓冲区的内容，这就需要使用一个栈来暂存当前输入文件的缓冲区。虽然Flex也有提供相关的函数来帮助你做这件事情，但这些函数的功能比较弱，建议自己手写更好。

Flex库函数input：

Flex库函数input()可以从当前的输入文件中读入一个字符，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：

```
1  %%
2  "//"  {
3     char c = input();
4     while (c != '\n') c = input();
5    }
```

Flex库函数unput：

Flex库函数unput(char c)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是很方便的。例如，假设之前定义过一个宏#define BUFFER_LEN 1024，当在输入文件中遇到字符串BUFFER_LEN时，下面这段代码将该宏所对应的内容放回输入缓冲区：

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

Flex库函数yyless和yymore：

Flex库函数yyless(int n)可以将刚从输入缓冲区中读取的yyleng–n个字符放回到输入缓冲区中，而函数yymore()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之后将下

一个词素连接到当前词素的后面。配合使用yyless()和yymore()可以方便地处理那些边界难以界定的模式。例如，我们在为字符串常量书写正则表达式时，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：

```
1  %%
2  \"[^\"]*\"  {
3      if (yytext[yyleng - 2] == '\\') {
4        yyless(yyleng - 1);
5        yymore();
6       } else {
7        /* process the string literal */
8       }
9    }
```

Flex宏REJECT：

Flex宏REJECT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会进行一系列的操作，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，考虑下面这段Flex源代码：

```
1  %%
2  pink  { npink++; REJECT; }
3  ink  { nink++; REJECT; }
4  pin  { npin++; REJECT; }
```

这段代码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还有更多的特性，感兴趣的读者可以参考其用户手册。

2.2.6 词法分析提示

为了完成实验一，首先需要阅读C--语言文法（附录A），包括其文法定义和补充说明。除了INT、FLOAT和ID这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难度。

阅读完C--语言文法，对C--的词法有大概的了解之后，就可以开始编写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程序是否工作正常，你可以暂时向屏幕打印当前的词法单元的名称，例如：

```
1  %%
2  "+"  { printf("PLUS\n"); }
3  "-"  { printf("SUB\n"); }
4  "&&"  { printf("AND\n"); }
5  "||"  { printf("OR\n"); }
6  …
```

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 characters \'%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输入文件来测试你的词法分析程序了。一定要确保你的词法分析程序的正确性！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的问题再回头调试，那将增加许多不必要的麻烦。为了使你在编写Flex源代码时少走弯路，以下是几条建议：

1) 留意空格和回车的使用。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源代码里的空白符。

2) 正则表达式和其所对应的动作之间，永远不要插入空行。

3) 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想要的。

4) 使用花括号括起每一段动作，即使该动作只包含有一行代码。

5) 在定义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。

6) 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“{”和“}”括起来。

2.2.7 语法分析概述

词法分析的下一阶段是语法分析。语法分析程序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规范，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。

语法分析与词法分析有很多相似之处：它们的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，它们都能被高效地完成，它们的构建都可以被工具自动化完成。不过，由于语法分析本身要比词法分析复杂得多，手写一个语法分析程序的代价太大，所以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助完成的。

正则表达式难以进行任意大的计数，所以很多在程序设计语言中常见的结构（例如匹配的

括号）无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或CFG）**。然而，虽然上下文无关文法在表达能力上要强于正则语言，但在判断某个输入串是否属于特定CFG的问题上，时间效率最好的算法也要$O(n^3)$[1]，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的推导规则，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它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所生成的语法分析程序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我们讨论的范围）。而其它的某些语法分析工具，例如基于Java语言的JTB[2]生成的语法分析程序，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。和词法分析程序的生成工具一样，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程序生成工具，虽然工具本身不能帮助直接构造出语法树，但我们可以通过在语法产生式中插入语义动作这种更加灵活的形式，来实现一些甚至比构造语法树更加复杂的功能。

2.2.8 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。

---

[1] 《自动机理论、语言和计算导论》，John E. Hopcroft等著，刘田、姜晖和王捍贫译，机械工业出版社，中信出版社，第209页，2004年。

[2] http://compilers.cs.ucla.edu/jtb/.

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的基本使用方法和课本上所介绍的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源代码。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。打开这个文件你就会发现，该文件本质上就是一份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，语法分析程序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，这个函数的名字正是yylex()。

函数yylex()相当于嵌在Bison里的词法分析程序。这个函数可以由用户自行实现，但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生成的yylex()函数呢？答案是肯定的。

仍以Bison源代码文件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编译这份源代码时，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文件之后，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源代码中规则部分的每一条action都返回相应的词法单元，如下所示：

```
1  %{
2    #include "syntax.tab.h"
3    …
4  %}
5  …
6  %%
7  "+"  { return PLUS; }
8  "-"  { return SUB; }
```

```
 9  "&&"  { return AND; }
10  "||"  { return OR; }
11  …
```

其中，返回值PLUS和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：

```
 1  int main(int argc, char** argv)
 2  {
 3    if (argc <= 1) return 1;
 4    FILE* f = fopen(argv[1], "r");
 5    if (!f)
 6    {
 7      perror(argv[1]);
 8      return 1;
 9    }
10    yyrestart(f);
11    yyparse();
12    return 0;
13  }
```

现在我们有了三个C语言源代码文件：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。如果加上“-ly”参数，编译报错，可以安装libbison-dev依赖库：

```
sudo apt-get install libbison-dev
```

现在我们可以使用这个parser程序进行语法分析了。例如，想要对一个输入文件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。

2.2.9 Bison：编写源代码

我们前面介绍了使用Flex和Bison联合创建语法分析程序的基本步骤。在整个创建过程中，最重要的文件无疑是你所编写的Flex源代码和Bison源代码文件，它们完全决定了所生成

的语法分析程序的一切行为。Flex源代码如何进行编写前面已经介绍过了，接下来我们介绍如何编写Bison源代码。

同Flex源代码类似，Bison源代码也分为三个部分，其作用与Flex源代码大致相同。第一部分是**定义部分**，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**，这部分的源代码会被原封不动地拷贝到syntax.tab.c中，以方便用户自定义所需要的函数（main函数也可以写在这里，不过不推荐这么做）。值得一提的是，如果用户想要对这部分所用到的变量、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“%}”所包围的内容也会被一并拷贝到syntax.tab.c的最前面。

下面我们通过一个例子来对Bison源代码的结构进行解释。一个在控制台运行可以进行整数四则运算的小程序，其语法如下所示（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
```

这个小程序的Bison源代码为：

```
 1  %{
 2    #include <stdio.h>
 3  %}
 4
 5  /* declared tokens */
 6  %token INT
 7  %token ADD SUB MUL DIV
 8
 9  %%
10  Calc :  /* empty */
11    | Exp  { printf("= %d\n", $1); }
12    ;
13  Exp : Factor
14    | Exp ADD Factor  { $$ = $1 + $3; }
15    | Exp SUB Factor  { $$ = $1 - $3; }
16    ;
17  Factor : Term
18    | Factor MUL Term  { $$ = $1 * $3; }
19    | Factor DIV Term  { $$ = $1 / $3; }
20    ;
21  Term : INT
22    ;
23  %%
24  #include "lex.yy.c"
25  int main() {
26    yyparse();
```

```
27  }
28  yyerror(char* msg) {
29    fprintf(stderr, "error: %s\n", msg);
30  }
```

这段Bison源代码以“%{”和“%}”开头，被“%{”和“%}”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。与终结符相对的，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。

第二部分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）。产生式里的箭头在这里用冒号“:”表示，一组产生式与另一组之间以分号“;”隔开。产生式里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$$表示，右边的几个符号的属性值按从左到右的顺序依次对应为$1、$2、$3等。每条产生式的最后可以添加一组以花括号“{”和“}”括起来的语义动作，这组语义动作会在整条产生式归约完成之后执行，如果不明确指定语义动作，那么Bison将采用默认的语义动作{ $$ = $1 }。语义动作也可以放在产生式的中间，例如$A \to B\ \{\ldots\}\ C$，这样的写法等价于$A \to BMC$，$M \to \varepsilon\ \{\ldots\}$，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。

在这里你可能有疑问：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该如何得到呢？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源代码修改为：

```
1  …
2  digit  [0-9]
3  %%
4  {digit}*  {
5      yylval = atoi(yytext);
6      return INT;
7    }
8  …
9  %%
10 …
```

其中yylval是Flex的内部变量，表示当前词法单元所对应的属性值。我们只需将该变量的值赋成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源代码中。在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()函数会在你的语法分析程序每发现一个语法错误时被调用，其默认参数为“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，而你也可以自己重新定义这个函数，从而使它打印一些别的内容，例如上例中我们就在该参数前面多打印了“error: ”的字样。

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。

2.2.10 Bison：属性值的类型

在上面的例子中，每个终结符或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的类型而不仅仅是int类型。下面我们介绍如何在Bison中解决这个问题。

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Bison源代码的“%{”和“%}”之间加入#define YYSTYPE float，那么所有的属性值就都成为了float类型。那么如何使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。

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个域。实际上，在Bison中已经内置了其它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。

我们仍然还是以前面的四则运算小程序为例，来说明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
```

在这份语法中，我们希望词法单元INT能有整型属性值，而FLOAT能有浮点型属性值，其他的非终结符为了简单起见，我们让它们都具有双精度型的属性值。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。

2.2.11 Bison：语法单元的位置

实验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以及在每个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，非常不方便。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中每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、$2等进行引用。实际上除了属性值之外，每个语法单元还对应了一个位置信息，在语义动作中这些位置信息同样可以使用@$、@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：

```
1  typedef struct YYLTYPE {
2    int first_line;
3    int first_column;
4    int last_line;
5    int last_column;
6  }
```

其中的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，输出所需的位置信息就比较方便了。但注意，如果直接引用@1、@2等将每个语法单元的first_line打印出来，你会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并不会主动替我们维护这些位置信息，我们需要在Flex源代码文件中自行维护。不过只要稍加利用Flex中的某些机制，维护这些信息并不需要太多的代码量。我们可以在Flex源文件的开头部分定义变量yycolumn，并添加如下的宏定义YY_USER_ACTION：

```
1  %locations
2  …
3  %{
4    /* 此处省略#include部分 */
5    int yycolumn = 1;
6    #define YY_USER_ACTION \
7      yylloc.first_line = yylloc.last_line = yylineno; \
8      yylloc.first_column = yycolumn; \
9      yylloc.last_column = yycolumn + yyleng - 1; \
10     yycolumn += yyleng;
11 %}
```

其中yylloc是Flex的内置变量，表示当前词法单元所对应的位置信息；YY_USER_ACTION宏表示在执行每一个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。除此之外，最后还要在Flex源代码文件中做的更改，就是在发现了换行符之后对变量yycolumn进行复位：

```
1  …
2  %%
3  …
4  \n  { yycolumn = 1; }
```

这样就可以实现在Bison中正常打印位置信息。

2.2.12 Bison：二义性与冲突处理

Bison有一个非常好用但也很恼人的特性：对于一个有二义性的文法，它有自己的一套隐式的冲突解决方案（一旦出现归约/归约冲突，Bison总会选择靠前的产生式；一旦出现移入/归约冲突，Bison总会选择移入）从而生成相应的语法分析程序，而这些冲突解决方案在某些场合可能并不是我们所期望的。因此，我们建议在使用Bison编译源代码时要留意它所给的提示信息，如果提示文法有冲突，那么请一定对源代码进行修改，尽量把所有冲突全部消解掉。

前面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 Factor | Exp SUB Factor变成了Exp → Exp ADD Exp | Exp SUB Exp），但实际上前面之所以没有这样写，是因为这样做会引入二义性。例如，如果输入为1 – 2 + 3，语法分析程序将无法确定先算1 – 2还是2 + 3。语法分析程序在读到1 – 2的时候可以归约（即先算1 – 2）也可以移入（即先算2 + 3），但由于Bison默认移入优先于归约，语法分析程序会继续读入+ 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 Factor | Exp SUB Factor相当于规定加减法为左结合），要么显式地指定算符的优先级与结合性。一般而言，重写语法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任何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，而“%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：

```
1  %right ASSIGN
2  %left ADD SUB
3  %left MUL DIV
4  %left LP RP
```

其中ASSIGN表示赋值号，LP表示左括号，RP表示右括号。此外，Bison也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。在实验一所使用的C--语言里，表达式Exp的语法便

是冲突的，你需要模仿前面介绍的方法，根据C--语言的文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很常见的冲突就是嵌套if-else所出现的冲突（也被称为悬空else问题）。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）是一致的。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程序的行为也是正确的。但如果不处理，Bison总是会提示我们该语法中存在一个移入/归约冲突。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的优先级。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：

```
1  …
2  %nonassoc LOWER_THAN_ELSE
3  %nonassoc ELSE
4  …
5  %%
6  …
7  Stmt : IF LP Exp RP Stmt  %prec LOWER_THAN_ELSE
8    |  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程序读到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。

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，但我们不建议使用它们解决除了表达式和if-else之外的任何其它冲突。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的方法来解决，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也能看出二义性。此时你要做的不是掩盖这些语法上的问题，而是仔细对语法进行修订，发现并解决语法本身的问题。

2.2.13 Bison：源代码的调试

以下这部分内容是可选的，跳过不会对实验一的完成产生负面影响。

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，你会在当前目录下发现一个新文件syntax.output，这个文件中包含Bison所生成的语法分析程序对应的LALR状态机的一些详尽描述。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，但无法确定在何处，就可以阅读这个.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，.output文件的第一句就会是：

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

继续向下翻，找到状态112，.output文件对该状态的描述为：

```
1  State 112
2
3  36 Stmt : IF LP Exp RP Stmt .
4  37   | IF LP Exp RP Stmt . ELSE Stmt
5
6  ELSE shift, and go to state 114
7
8  ELSE [reduce using rule 36 (Stmt)]
9  $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。知道是这里出现了问题，我们就可以以此为线索修改Bison源代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−−语言）而言，Bison所产生的LALR语法分析程序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，有些问题，例如语法分析程序在运行时刻出现“Segmentation fault”等，很难从对状态机的静态描述中发现，必须要在动态、交互的环境下才容易看出问题所在。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（或者在代码中加上#define YYDEBUG 1）：

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

在main函数调用yyparse()之前我们加一句：yydebug = 1;，然后重新编译整个程序。之后运行这个程序你就会发现，语法分析程序现在正像一个自动机，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，以方便你检查自己代码中哪里出现了问题。以前面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：

```
1  Starting parse
2  Entering state 0
3  Reading a token:
```

如果我们输入4，你会明显地看到语法分析程序出现了状态转换，并将当前栈里的内容打印出来：

```
 1  Next token is token INT ()
 2  Shifting token INT ()
 3  Entering state 1
 4  Reducing stack by rule 9 (line 29):
 5    $1 = token INT ()
 6  -> $$ = nterm Term ()
 7  Stack now 0
 8  Entering state 6
 9  Reducing stack by rule 6 (line 25):
10    $1 = nterm Term ()
11  -> $$ = nterm Factor ()
12  Stack now 0
13  Entering state 5
14  Reading a token:
```

继续输入其他内容，我们可以看到更进一步的状态转换。

注意，诊断模式会使语法分析程序的性能下降不少，建议在不使用时不要随便打开。

2.2.14 Bison：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程序尽早地报告错误。每当语法分析程序从yylex()得到了一个词法单元，如果当前状态并没有针对这个词法单元的动作，那就会认为输入文件里出现了语法错误，此时它默认进入下面这个错误恢复模式：

1) 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的字样。

2) 从栈顶弹出所有还没有处理完的规则，直到语法分析程序回到了一个可以移入特殊符号error的状态。

3) 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**再同步**）。

4) 如果在error之后能成功移入三个符号，则继续正常的语法分析；否则，返回前面的步骤二。

这些步骤看起来似乎很复杂，但实际上需要我们做的事情只有一件，即在语法里指定error符号应该放到哪里。不过，需谨慎考虑放置error符号的位置：一方面，我们希望error后

面跟的内容越多越好，这样再同步就会更容易成功，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
```

以上几个产生式仅仅是示例，并不意味着把它们照搬到你的Bison源代码中就可以让语法分析程序能够满足实验一的要求。你需要进一步思考如何书写包含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。

2.2.15 语法分析提示

想要做好一个语法分析程序，第一步要仔细阅读并理解C--语法规范。C--的语法要比它的词法复杂很多，如果缺乏对语法的理解，在调试和测试语法分析程序时你将感到无所适从。另外，如果没弄懂C--语法中每条产生式背后的具体含义，则无法在后面的实验二中去分析这些产生式的语义。

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源代码、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该如何设置以及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无法编译，那你需要重新阅读前文并仔细检查哪里出了问题。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帮助，要查出错误并不是太难的事情。

再下一步需要考虑语法树的表示和构造。语法树是一棵多叉树，因此为了能够建立它你需要实现多叉树的数据结构。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源代码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结点作为产生式左边的非终结符所对应的树结点的子结点逐个进行插入。具体这棵多叉树的数据结构怎么定义、插入操作怎么完成等完全取决于你的设计，在设计过程中有一点需

要注意：在实验二中我们还会在这棵语法树上进行一些其它的操作，所以现在的数据结构设计会对后面的语义分析产生一定的影响。

构造完这棵树之后，下一步就是按照实验一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相关信息。为了能打印这些信息，你需要写专门的代码对生成的语法树进行遍历。由于还要求打印行号，所以在之前生成语法树的时候你就需要将每个结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以）。这段负责打印的代码仅是为了实验一而写，后面的实验不会再用，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，以方便后面的实验对代码的调整。