编译原理实验指导书

Compilers Principles Experiment Instruction Book

陈贺敏　王林

燕山大学软件工程系

实验 3 基于 LR(0)方法的语法分析

3.1 实验目的

（1）掌握 LR(0)分析表的构造方法。

（2）掌握设计、编制和调试典型的语法分析程序，进一步掌握常用的语法分析方法。

（3）理解语法分析在编译程序中的作用。

3.2 实验任务

编写一个程序对输入的源代码进行语法分析，并打印分析结果。（以下两个任务二选一）

1、借助语法分析工具 GNU Bison，编写一个对使用 C--语言书写的源代码进行语法分析（C--语言的文法参见附录 A），并使用 C 语言完成。

2、自己编写一个基于 LR(0)方法的语法分析程序。语言不限，文法不限。

此时可根据自己的实际情况，选择以下一项实现分析算法中分析表的构造：

（1）直接输入根据已知文法构造的 LR(0)分析表；

（2）输入已知文法的项目集规范族和转换函数，由程序自动生成 LR(0)分析表；

（3）输入已知文法，由程序自动生成 LR(0)分析表。

3.3 实验内容

3.3.1 实验要求

任务 1：你的程序要能够查出源代码中可能包含的语法错误：语法错误（**错误类型 B**）。

任务 2：你的程序应具有通用性，能够识别由词法分析得出的词法单元序列是否是给定文法的正确句子（程序），并能够输出分析过程和识别结果。

3.3.2 输入格式

1、如果你选择的是**任务 1**，那么你的程序的输入是一个包含 C--源代码的文本文件，程序需要能够接收一个输入文件名作为参数。例如，假设你的程序名为 cc、输入文件名为 test1、程序和输入文件都位于当前目录下，那么在 Linux 命令行下运行./cc test1 即可获得以 test1 作为输入文件的输出结果。

2、如果你选择的是**任务 2**，那么你的程序输入是一个包含待分析词法单元序列的文本文件，程序需要能够接收一个输入文件名作为参数，以获得相应的输出结果。

3.3.3 输出格式

实验三要求通过标准输出打印程序的运行结果。

任务 1：对于那些包含语法错误的输入文件，只要输出相关的语法有误的信息即可。在这种情况下，注意不要输出任何与语法树有关的内容。要求输出的信息包括错误类型、出错的行号以及说明文字，

其格式为：Error type [错误类型] at Line [行号]: [说明文字].

说明文字的内容没有具体要求，但是错误类型和出错的行号一定要正确，因为这是判断输出的错误提示信息是否正确的唯一标准。请严格遵守实验要求中给定的错误分类（**即语法错误为错误类型 B**），否则将影响你的实验评分。**注意，**输入文件中可能会包含一个或者多个语法错误（但输入文件的同一行中保证不出现多个错误），你的程序需要将这些错误全部报告出来，每一条错误提示信息在输出中单独占一行。

对于那些没有任何语法错误的输入文件，则需要输出相应的语法分析结果。

你的程序需要将构造好的语法树按照先序遍历的方式打印每一个结点的信息，这些信息包括：

1） 如果当前结点是一个语法单元并且该语法单元没有产生ε（即空串），则打印该语法单元的名称以及它在输入文件中的行号（行号被括号所包围，并且与语法单元名之间有一个空格）。所谓某个语法单元在输入文件中的行号是指该语法单元产生出的所有词素中的第一个在输入文件中出现的行号。

2） 如果当前结点是一个语法单元并且该语法单元产生了ε，则无需打印该语法单元的信息。

3） 如果当前结点是一个词法单元，则只要打印该词法单元的名称，而无需打印该词法单元的行号。

a) 如果当前结点是词法单元 ID，则要求额外打印该标识符所对应的词素；

b) 如果当前结点是词法单元 TYPE，则要求额外打印说明以该类型为 int 还是 float；

c) 如果当前结点是词法单元 INT 或者 FLOAT，则要求以十进制的形式额外打印该数 字所对应的数值；

d) 词法单元所额外打印的信息与词法单元名之间以一个冒号和一个空格隔开。

每一条词法或语法单元的信息单独占一行，而每个子结点的信息相对于其父结点的信息来说，在行首都要求缩进 2 个空格。具体输出格式可参见后续的样例。

任务 2：你的程序需要输出语法分析过程（包括 **LR(0)分析表**和**分析过程表**，并能够保存 LR(0)分析表）和相应的分析结果（即此串是否为 LR(0)文法的句子）。

3.3.4 测试环境

任务 1：你的程序将在如下环境中被编译并运行：

1) GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0-29；

2) GCC version 4.6.3；

3) GNU Flex version 2.5.35；

4) GNU Bison version 2.5.

一般而言，只要避免使用过于冷门的特性，使用其它版本的 Linux 或者 GCC 等，也基本上不会出现兼容性方面的问题。注意，实验三的检查过程中不会去安装或尝试引用各类方便编程的函数库（如 glib 等），因此请不要在你的程序中使用它们。

3.3.5 提交要求

1) 任务 1：以 Flex、Bison 以及 C 语言编写的可被正确编译运行的源程序。

任务 2：可以是 Java、C、Python 等任意语言编写的可被正确编译运行的源程序。

2）一份实验报告，内容包括：实验目的、实验任务、实验内容，算法描述，程序结构，主要变量说明，程序清单，调试情况及各种情况运行结果截图，心得体会。

3.3.6 样例

此部分样例是以**任务 1** 为例给出的，请仔细阅读样例，以加深对实验要求以及输出格式要求的理解。

说明：选择任务 2 的同学，可以参考教材 P125 页中表 6.1 和表 6.2 相关内容，自行设计测试样例，以反映出正确输出。

样例 1：

输入（行号是为标识需要，并非样例输入的一部分，后同）：

```
1  int main()
2  {
3      float a[10][2];
4      int i;
5      a[5,3] = 1.5;
6      if (a[1][2] == 0)   i = 1 else i = 0;
7  }
```

输出：

这个程序存在两处语法错误。其一，虽然我们的程序中允许出现方括号与逗号等字符，但二维数组正确的访问方式应该是 a[5][3]而非 a[5,3]。其二，第 6 行的 if-else 语句在 else 之前少了一个分号。因此你的程序可以输出如下的两行错误提示信息：

Error type B at Line 5: Missing "]".

Error type B at Line 6: Missing ";".

样例 2：

输入：

```
1  int inc()
2  {
3      int    i;
4      i = i + 1;
5  }
```

输出：

这个程序非常简单，也没有任何词法或语法错误，因此你的程序需要输出如下的语法树结点信息（左侧行号是为标识需要，并非程序输出的一部分）：

```
1  Program (1)
2    ExtDefList (1)
3      ExtDef (1)
4        Specifier (1)
5          TYPE: int
6        FunDec (1)
7          ID: inc
8          LP
9          RP
10       CompSt (2)
11         LC
12         DefList (3)
13           Def (3)
14             Specifier (3)
15               TYPE: int
16             DecList (3)
17               Dec (3)
18                 VarDec (3)
19                   ID: i
20             SEMI
21         StmtList (4)
22           Stmt (4)
23             Exp (4)
24               Exp (4)
25                 ID: i
26               ASSIGNOP
27               Exp (4)
28                 Exp (4)
```

```
29                      ID: i
30                  PLUS
31                  Exp (4)
32                      INT: 1
33              SEMI
34          RC
```

3.4 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。

本节指导内容将主要介绍语法分析工具 GNU Bison。如前所述，完成实验三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三。

3.4.1 语法分析概述

语法分析是词法分析的下一阶段。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。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(n3)，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 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。

3.4.2 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。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。

3.4.3 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：

```
%{
    #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();
```

```
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→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。

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。

3.4.4 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; }
```

```
33      | FLOAT { $$ = $1; }
34      ;
35
36  %%
37  …
```

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。

3.4.5 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：

```
%locations
…
%{
    /* 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 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：

```
…
%%
…
\n { yycolumn = 1; }
```

这样就可以实现在 Bison 中正常打印位置信息。

3.4.6 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。

3.4.7 Bison：源代码的调试

以下这部分内容是可选的，跳过不会对实验三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。

3.4.8 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三的要求。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。

3.4.9 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。

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。

构造完这棵树之后，下一步就是按照实验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三而写，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。