中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

编译器构造实验

Lab I – 词法分析与语法分析

2023年秋季学期

Teaching Team: 王焱林、伍嘉栋、黄炎贤、王岩立

课程信息

- 课程类别：专业选修课
- 课程学分：1分
- 开课单位：软件工程学院
- 主要教材
  - 《编译原理实践与指导教程》（非必须）
- 成绩构成
  - 平时40%（4次Lab各10%）+ 大作业60%
- 助教：伍嘉栋 (Lab1, Lab3)、黄炎贤(Lab2, Lab4)

特别说明

- 本课程需要和理论课《编译原理》配套学习
- 作业迟交：每天扣10%的作业分
- 作业抄袭：0分

1.1 实验内容

1.1.1 实验要求

- 编写一个程序对使用C--语言书写的源代码进行词法分析和语法分析（C--语言的文法参见附录A），并打印分析结果。
- 能够查出C--源代码中可能包含的下述几类错误：
  - 词法错误（错误类型A）：即出现C--词法中未定义的字符以及任何不符合C--词法单元定义的字符；
  - 语法错误（错误类型B）

1.1.1 实验要求

- 你的程序需要完成以下的要求：
  - 要求1：识别八进制数和十六进制数。
    - 若输入文件中包含符合词法定义的八进制数（如0123）和十六进制数（如0x3F），你的程序需要得出相应的词法单元；
    - 若输入文件中包含不符合词法定义的八进制数（如09）和十六进制数（如0x1G），你的程序需要给出输入文件有词法错误（即错误类型A）的提示信息。
    - 八进制数和十六进制数的定义参见附录A

1.1.1 实验要求

- 你的程序需要完成以下的要求：
  - 要求2：识别指数形式的浮点数。
    - 若输入文件中包含符合词法定义的指数形式的浮点数（如1.05e-4），你的程序需要得出相应的词法单元；
    - 若输入文件中包含不符合词法定义的指数形式的浮点数（如1.05e），你的程序需要给出输入文件有词法错误（即错误类型A）的提示信息。
    - 指数形式的浮点数的定义参见附录A

1.1.1 实验要求

- 你的程序需要完成以下的要求：
  - 要求3：识别“//”和“/*…*/”形式的注释。
    - 若输入文件中包含符合定义的“//”和“/*…*/”形式的注释，你的程序需要能够滤除这样的注释；
    - 若输入文件中包含不符合定义的注释（如“/*…*/”注释中缺少“/*”），你的程序需要给出由不符合定义的注释所引发的错误的提示信息。
    - 注释的定义参见附录A。
- 程序在输出错误提示信息时，需要输出具体的错误类型、出错的位置（源程序行号）以及相关的说明文字。

1.1.2.输入格式

- 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。

1.1.3.输出格式

- 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
  - 严格遵守实验要求中给定的错误分类（即词法错误为错误类型A，语法错误为错误类型B）
  - 输入文件中可能会包含一个或者多个错误（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

1.1.3.输出格式

- 对于那些没有任何词法或语法错误的输入文件，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：
  - 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。
  - 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。

1.1.3.输出格式

- 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。

1.1.4.环境

- 你的程序将在如下环境中被编译并运行：
  - GNU Linux Release: Ubuntu 20.04, kernel version 5.13.0-44-generic；
  - GCC version 7.5.0
  - GNU Flex version 2.6.4；
  - GNU Bison version 3.5.1
  - 使用其它版本的Linux或者GCC等也可以

1.1.4.提交要求

- 实验一要求提交如下内容：
  - 提交链接：https://send2me.cn/RnW5hIC1/TTiJ1TxqgykpjQ
  - 截止时间：2023.9.28 23:59
  - Flex、Bison以及C语言的可被正确编译运行的源程序
  - 实验报告（pdf）
    - ✓实现了哪些功能?
    - ✓可以使用脚本、makefile或逐条输入命令进行编译，请详细说明应该如何编译你的程序。

I.I.4.样例

- 必做:
  - 样例1
    - ✓ 输入

```
1  int main()
2  {
3    int i = 1;
4    int j = ~i;
5  }
```

    - ✓ 输出

```
Error type A at Line 4: Mysterious character "~".
```

说明：这个程序存在词法错误。第4行中的字符“~”没有在我们的C--词法中被定义过

  - 样例2
    - ✓ 输入

```
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

```
Error type B at Line 5: Missing "]".
Error type B at Line 6: Missing ";".
```

说明：两处语法错误

I.1.4.样例

- 必做：
  - 样例3
    - ✓ 输入

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

    - ✓ 输出

这个程序没有任何错误，序需要输出如下的语法树结点信息

```
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
```

1.1.4.样例

- 必做：
  - 样例4
    - ✓ 输入

```
1  struct Complex
2  {
3    float real, image;
4  };
5  int main()
6  {
7    struct Complex x;
8    y.image = 3.5;
9  }
```

```
1  Program (1)
2    ExtDefList (1)
3      ExtDef (1)
4        Specifier (1)
5          StructSpecifier (1)
6            STRUCT
7            OptTag (1)
8              ID: Complex
9            LC
10           DefList (3)
11             Def (3)
12               Specifier (3)
13                 TYPE: float
14               DecList (3)
15                 Dec (3)
16                   VarDec (3)
17                     ID: real
18                 COMMA
19                 DecList (3)
20                   Dec (3)
21                     VarDec (3)
22                       ID: image
23               SEMI
24           RC
25       SEMI
```

```
26     ExtDefList (5)
27       ExtDef (5)
28         Specifier (5)
29           TYPE: int
30         FunDec (5)
31           ID: main
32           LP
33           RP
34         CompSt (6)
35           LC
36           DefList (7)
37             Def (7)
38               Specifier (7)
39                 StructSpecifier (7)
40                   STRUCT
41                   Tag (7)
42                     ID: Complex
43               DecList (7)
44                 Dec (7)
45                   VarDec (7)
46                     ID: x
47               SEMI
48           StmtList (8)
49             Stmt (8)
50               Exp (8)
51                 Exp (8)
52                   Exp (8)
53                     ID: y
54                   DOT
55                   ID: image
56                 ASSIGNOP
57                 Exp (8)
58                   FLOAT: 3.500000
59               SEMI
60           RC
```

    - ✓ 输出

这个程序虽然包含了语义错误（即使用了未定义的变量y），但不存在任何词法或语法错误，因此你的程序不能报错而是要输出相应的语法树结点信息。

1.1.4.样例

- 必做：
  - 样例1
    - ✓ 输入

```
1  int main()
2  {
3    int i = 0123;
4    int j = 0x3F;
5  }
```

    - ✓ 输出

      如果你的程序完成要求1，该样例输入不包含任何词法或语法错误，其对应的输出信息：

```
1  Program (1)
2    ExtDefList (1)
3      ExtDef (1)
4        Specifier (1)
5          TYPE: int
6        FunDec (1)
7          ID: main
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                 ASSIGNOP
21                 Exp (3)
22                   INT: 83
23             SEMI
24           DefList (4)
25             Def (4)
26               Specifier (4)
27                 TYPE: int
28               DecList (4)
29                 Dec (4)
30                   VarDec (4)
31                     ID: j
32                   ASSIGNOP
33                   Exp (4)
34                     INT: 63
35               SEMI
36         RC
```

1.1.4.样例

- 必做：
  - 样例2
    - ✓ 输入

```
1  int main()
2  {
3    int i = 09;
4    int j = 0x3G;
5  }
```

    - ✓ 输出

你的程序可以直接将“09”和“0x3G”分别识别为错误的八进制数和错误的十六进制，此时你的程序也可以输出如下的错误提示信息：

```
Error type A at Line 3: Illegal octal number '09'.
Error type A at Line 4: Illegal hexadecimal number '0x3G'.
```

1.1.4.样例

- 必做：
  - 样例3
    - ✓ 输入

```
1  int main()
2  {
3    float i = 1.05e-4;
4  }
```

    - ✓ 输出

如果你的程序需要完成要求2，该样例程序不包含任何词法或语法错误，其对应的输出为：

```
1  Program (1)
2    ExtDefList (1)
3      ExtDef (1)
4        Specifier (1)
5          TYPE: int
6        FunDec (1)
7          ID: main
8          LP
9          RP
10       CompSt (2)
11         LC
12         DefList (3)
13           Def (3)
14             Specifier (3)
15               TYPE: float
16             DecList (3)
17               Dec (3)
18                 VarDec (3)
19                   ID: i
20                 ASSIGNOP
21                 Exp (3)
22                   FLOAT: 0.000105
23             SEMI
24         RC
```

I.I.4.样例

- 必做：
  - 样例4
    - ✓ 输入

```
1  int main()
2  {
3    float i = 1.05e;
4  }
```

    - ✓ 输出

如果你的程序需要完成要求2，该样例程序中的“1.05e”为错误的指数形式的浮点数，其会因被识别为浮点数“1.05”和标识符“e”而引发语法错误。因此你的程序可以输出如下的错误提示信息：

```
Error type B at Line 3: Syntax error.
```

你的程序也可以直接将“1.05e”识别为错误的指数形式的浮点数，此时你的程序也可以输出如下的错误提示信息：

```
Error type A at Line 3: Illegal floating point number "1.05e".
```

1.1.4.样例

- 必做：
  - 样例5
    - ✓ 输入

```
1  int main()
2  {
3    // line comment
4    /*
5   block comment
6    */
7    int i = 1;
8  }
```

    - ✓ 输出

如果你的程序需要完成要求3，该样例程序不包含任何词法或语法错误，其对应的输出为：

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

1.1.4.样例

- 必做：
  - 样例6
    - ✓ 输入

```
1  int main()
2  {
3    /*
4    comment
5    /*
6    nested comment
7    */
8    */
9    int i = 1;
10 }
```

    - ✓ 输出

如样例输入中程序员主观上想使用嵌套的“/*…*/”注释，但C--语言不支持嵌套的“/*…*/”注释。如果你的程序需要完成要求1.3，该样例输入中第8行的“*/”会因被识别为乘号“*”和除号“/”而引发语法错误（你只需为每行报告一个语法错误），你的程序可以输出如下的错误提示信息：

```
Error type B at Line 8: Syntax error.
```

1.2 实验指导

1.2.1词法分析概述

- 读入源程序的字符流，输出为有意义的词素 (lexeme)
  - <token-name, attribute-value>
  - token-name由语法分析步骤使用
  - attribute-value指向相应的符号表条目，由语义分析/ 代码生成步骤使用
- 例子
  - position = initial + rate * 60
  - <id, 1> <=, > <id, 2> <+, > <id, 3> <*, > <number, 4>

1.2.1词法分析概述

- 理论基础
  - 正则表达式

1) **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。

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 | …。

  - 有限状态自动机(FSA)

✓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。

1.2.2GNU Flex

```
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 }
```

- 安装编译：
  - 安装: sudo apt-get install flex
  - 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
    - 其中yylex() 函数的作用就是读取输入文件中的一个词法单元。为它编写一个main函数
    - 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。

1.2.2GNU Flex

- 安装编译：
  - 将这个main函数单独放到一个文件main.c中，此时需要要声明yyin为外部变量：extern FILE* yyin；然后编译这两个C源文件，将输出程序命名为scanner

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

  - 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
```

1.2.2GNU Flex

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

- Flex：编写源代码
  - Flex源代码文件包括三个部分，由“%%”隔开，如下所示：
    - 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

      ```
      name definition
      ```

    - 其中name是名字，definition是任意的正则表达式

```
27   letter  [_a-zA-Z]
28   digit   [0-9]
29   relop   >|<|>=|<=|==|!=
```

1.2.2GNU Flex

- Flex：编写源代码
  - 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。
  - 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  …
```

1.2.2GNU Flex

- Flex：编写源代码
  - 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
  - 利用Flex实现文字统计工具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：

```
%{
  /* 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;
}
```

1.2.2GNU Flex

- Flex：编写源代码
  - 大部分可以直接按照附录A写规则，INT\FLOAT\ID需要自己书写正则表达式

```
66   {relop} {handle("RELOP",yytext);return RELOP;}//left
67   "+" {handle("PLUS",yytext);return PLUS;}//left
68   "-" {handle("MINUS",yytext);return MINUS;}//right|left
69   "*" {handle("STAR",yytext);return STAR;}//left
70   "/" {handle("DIV",yytext);return DIV;}//left
```

  - 错误提示信息可以把这条规则放在最后

```
96   . {
97   printf("Error type A at Line %d: Mysterious characters \'%s\'\n",
98   yylineno, yytext);
99   error++;
100  }
```

1.2.2GNU Flex

- 正则表达式

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，[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

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“*”为Kleene闭包操作，匹配零个或者多个表达式。

5) 符号“+”为正闭包操作，匹配一个或者多个表达式。

7) 符号“?”匹配零个或者一个表达式。

8) 符号“|”为选择操作，匹配其之前或之后的任一表达式

9) 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。

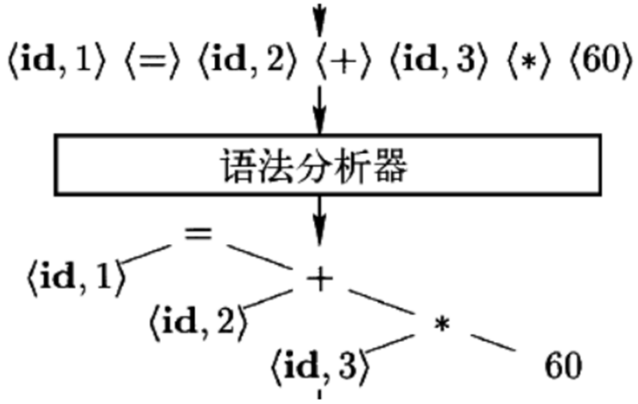
补充：初始测试 (交互)

- 编译运行步骤
  - vim lexical0.l
  - flex lexical0.l
  - gcc -o lexical0 lex.yy.c
  - ./lexical0
- lexical0.l 文件里面填右边代码

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

delim   [ \t\n]
whitespace  {delim}+
digit   [0-9]
letter  [A-Za-z_]
id  {letter}({letter}|{digit})*

%%
{whitespace}    {;}
{id}        {printf("(ID, %s)\n", yytext);}
.       {printf("ERROR\n");}

%%
int yywrap(){
    return 1;
}

int main(int argc, char ** argv){
    yylex();
    return 0;
}
```

1.2.3语法分析概述

- 根据各个词法单元的第一个分量来创建树型的中间表示形式，通常是语法树 (syntax tree)
- 中间表示形式指出了词法单元流的语法结构



1.2.3语法分析概述

- 上下文无关文法（Context Free Grammar或CFG）
  - 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
  - 产生式集合P
    - 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。
    - 实验使用的Bison所生成的语法分析程序采用了自底向上的LALR(1)分析技术

1.2.4 GNU Bison

安装sudo apt-get install bison
编译：bison syntax.y

- 安装编译：
  - 编译好的结果会保存在当前目录下的syntax.yy.c文件中。
  - 其中yyparse() 函数的作用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。
    - 语法分析程序的输入是一个个的词法单元，通过yylex()来获取。
    - 为了让Bison使用Flex生成的yylex()函数，可以在Bison源代码中引用lex.yy.c：

      ```
      #include "lex.yy.c"
      ```

      编译`bison -d syntax.y`
    - -d参数是将编译的结果分拆成syntax.tab.c和syntax.tab.h两个文件

Bison文档
https://www.gnu.org/software/bison/manual/

1.2.4GNU Bison

- 安装编译：
  - 得到.h文件之后，下一步是修改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，如图所示：
  - 重新编译

```
flex lexical.l
```

```
1  %{
2    #include "syntax.tab.h"
3    …
4  %}
5  …
6  %%
7  "+"  { return PLUS; }
8  "-"  { return SUB; }
9  "&&"  { return AND; }
10 "||"  { return OR; }
11 …
```

1.2.4GNU Bison

- 安装编译：
  - 重写main函数：

```
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
```

  - 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
```

1.2.4GNU Bison

- 编写源代码
  - Bison源代码也分为三个部分：
    - 定义部分
    - 规则部分，其中包括具体的语法和相应的语义动作
    - 用户函数部分

1.2.4GNU Bison

- 编写源代码
  - 例子：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

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

1.2.4GNU Bison

- 编写源代码
  - 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。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 }。

```
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 }
```

1.2.4GNU Bison

- 编写源代码
  - 终结符本身的属性值通过yylex()得到，想让终结符带有属性值，需要回头修改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：
    - yylval是Flex的内部变量，表示当前词法单元所对应的属性值

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

1.2.4GNU Bison

- 属性值的类型
  - 用%union{…}将所有可能的类型都包含进去
  - 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。

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

```
2   #include <stdio.h>
3  %}
4
5  /* declared types */
6  %union {
7    int type_int;
8    float type_float;
9    double type_double;
10 }
11
12 /* declared tokens */
13 %token <type_int> INT
14 %token <type_float> FLOAT
15 %token ADD SUB MUL DIV
16
17 /* declared non-terminals */
18 %type <type_double> Exp Factor Term
19
20 %%
21 Calc :  /* empty */
22   | Exp  { printf("= %lf\n, $1); }
23   ;
24 Exp : Factor
25   | Exp ADD Factor  { $$ = $1 + $3; }
26   | Exp SUB Factor  { $$ = $1 - $3; }
27   ;
28 Factor : Term
29   | Factor MUL Term  { $$ = $1 * $3; }
30   | Factor DIV Term  { $$ = $1 / $3; }
31   ;
32 Term : INT  { $$ = $1; }
33   | FLOAT  { $$ = $1; }
34   ;
35
36 %%
37 …
```

1.2.4GNU Bison

```
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 %}
```

- 语法单元的位置
  - 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：
  - 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; }
```

1.2.4GNU Bison

➢ 二义性与冲突处理

- 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
…
```

- 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

1.2.4 GNU Bison

➢ 二义性与冲突处理

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

- 在Bison源代码中， “%left”表示左结合， “%right”表示右结合，而 “%nonassoc”表示不可结合
- 优先级
  - 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?
  - 可以通过定义一个比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
```

1.2.4 GNU Bison

➢ 源代码调试

- 编译的时候加-v会产生.output文件，里面有比较详细的描述：

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

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


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)
```

1.2.4 GNU Bison

➢ 源代码调试

- 编译的时候加-t，打开其诊断模式，会打印相应的状态信息：

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

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

```
 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:
```