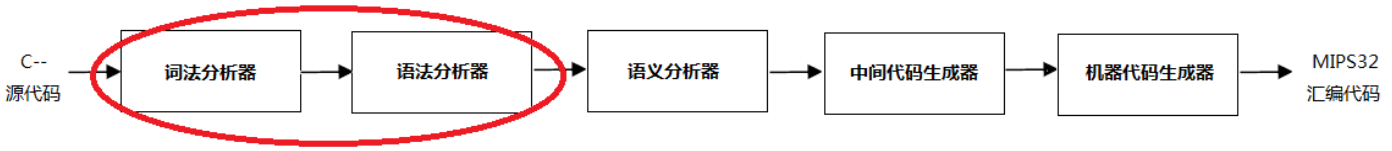
编译原理实习 1　词法分析与语法分析

许畅 陈嘉 朱晓瑞

编译原理课程的实习内容是为一个小型的类 C 语言(C--)实现一个编译器。如果你顺利完成了本课程规定的实习任务，那么不仅你的编程能力将会得到大幅提高，而且你最终会得到一个比较完整的、能将 C--源代码转换成 MIPS 汇编代码的编译器，所得到的汇编代码可以在一个被广泛使用且功能强大的 MIPS 汇编语言的汇编器+模拟器 SPIM Simulator 上运行。实习总共分为四个阶段：词法和语法分析、语义分析、中间代码生成和目标代码生成。每个阶段的输出是下一个阶段的输入，后一个阶段总是在前一个阶段的基础上完成，如下图所示：



实习 1 的任务是编写一个程序，该程序读入一个 C--源代码文件，对其进行词法分析和语法分析（C--语言的词法和语法参见 Grammar.pdf 文件），然后打印分析结果。该任务强制要求使用词法分析工具 GNU Flex、语法分析工具 GNU Bison 并使用 C 语言来完成。在两个强大工具的帮助下，你会发现编写一个能用来做词法分析和语法分析的程序是一件相当轻松愉快的事情。

需要注意的是，由于在后面的实习中还会用到本次实习你已经写好的代码，因此保持一个良好的代码风格、系统地设计代码结构和各模块之间的接口等对于整个实习来讲可以说是相当重要的。

输入格式

程序的输入是一个包含 C--源代码的文本文件（源代码中可能出现函数、结构体、一维和高维数组），你的程序要能够查出输入文件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不符合任何 C--词法单元定义的字符
- 语法错误（错误类型 B）

以上为每位同学的必做内容。除此之外，你的程序需要完成且**仅能完成**下列三个分组之一中的要求：

- 分组 1.1：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），你的词法分析程序能够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（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。
- 分组 1.2：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），你的词法分析程序能够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（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。

- 分组 1.3：识别“//”和“/**/”注释。若输入文件中包含符合定义的“//”和“/**/”注释，你的程序能够滤除注释；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引发的错误的提示信息。注释的定义参见附录。

你可以根据分配到的**任务编号**在任务说明文件（task.pdf）中查找你在整个编译课程实验中**必须**完成的分组。注意，除了必须完成的分组外，完成任何额外的分组都要**倒扣分**！你的程序在输出错误提示信息的时候要求输出具体的错误类型、出错的行号以及说明文字。

本次实习要求你的程序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。

输出格式

本次实习要求你的程序打印到标准输出之上。对于那些包含词法或者语法错误的输入文件，你只要输出词法或语法有误的信息即可。在这种情况下，注意一定**不要**输出和语法树有关的任何内容。要求输出的信息包括出错的**行号**、**错误类型**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，但是出错的行号和错误类型一定要写对，这是我们判断你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（即词法错误为错误类型 A，语法错误为错误类型 B），否则将严重降低你的分数。注意，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。

对于那些没有任何词法和语法错误的输入文件，你需要将构造好的语法树按照**先序遍历**的顺序遍历并且打印每一个节点的信息，这些信息应该包括：

- 如果当前节点是一个语法单元且该语法单元没有产生ε，则打印该语法单元名称以及它在输入文件中出现的行号（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出现的行号指的是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。
- 如果当前节点是一个语法单元且该语法单元产生了ε，则无需打印该语法单元的信息。
- 如果当前节点是一个词法单元，则只要打印该词法单元名称，无需打印该词法单元的行号。但是，
  - 如果当前节点是词法单元 ID，则要求额外打印该标识符所对应的词素
  - 如果当前节点是词法单元 TYPE，则要求额外打印究竟该类型为 int 还是 float

- 如果当前节点是词法单元 INT 或者 FLOAT，则要求以**十进制**的形式额外打印该标识符所对应的数值
- 词法单元所对应的信息与词法单元名之间以一个冒号和一个空格隔开

每一个词法/语法单元的信息单独占一行，而每个子节点的信息相对于其父节点的信息来说，在行首都要求缩进 2 个空格。具体输出格式请**仔细阅读**样例。

测试环境

你提交上来的源代码将在如下环境中被编译并运行：

- GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0-29
- GCC version 4.6.3
- GNU Flex version 2.5.35
- GNU Bison version 2.5

一般来说，只要你能避免使用那些过于冷门的特性，使用其他版本的 Linux（甚至是 Mac OS）或者 GCC 等也基本上不会出现兼容性方面的问题。如果你仍然担心自己的代码无法顺利编译和执行，我们可以提供一个满足上述要求的 VMWare 虚拟机给你测试一下。

提交要求

本次实习要求你提交如下内容：

- Flex、Bison 以及 C 语言的可以被正确编译运行的源代码
- 一份 PDF 格式的实验报告，内容主要包括：
  - 你的程序实现了哪些功能？简要说明你是如何实现这些功能的。如果因为你的说明不够充分而导致助教没有对你所实现的功能进行测试，那么后果自负。
  - 你所提交上来的程序应当如何编译？不管你使用了脚本也好，准备了 Makefile 也好甚至是单独地逐条命令手工输入进行编译也好，请**详细说明**具体需要键入哪些命令——无法顺利编译将会使你丢失相应分数，并且如果不去找助教进行修正，后面的正确分也会因你的程序无法运行而全部丢失，请谨记这一点。
  - 你的实验报告长度**不得超过** 3 页！因此，你需要好好考虑一下该往实验报告里写些什么。我们的建议是，实验报告中需要你重点描述的应当是你所提交的工作中的亮点，应当是那些你认为最**个性化**、最具有**独创性**的内容，而那些比较简单的、任何人都可以做出来的内容可以不提或者只简单的提一下，尤其要避免去大段大段地向报告里贴代码。

    为了避免大家通过减小字号来变相加长页数限制，我们规定实验报告中所出现的最小字号不得小于五号字（英文 11 号字）。

必做内容样例

样例输入 1

```
int main()
{
    int i = 1;
    int j = ~i;
}
```

样例输出 1

这个程序存在词法错误：第四行的字符"~"（C 语言中这是按位取非的运算符）并没有在我们的 C--词法中被定义，因此程序可以像这样输出一行错误提示信息：

```
Error type A at line 4: Mysterious character '~'
```

样例输入 2

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

样例输出 2

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以及逗号等字符，但二维数组正确的访问方式应该是 a[5][3]而不是 a[5,3]（后者是 Pascal 风格的数组访问形式）；其二，第5行的 if-else 语句在 else 之前少了一个分号。因此，程序应该像这样输出两行错误提示信息：

```
Error type B at line 4: Syntax error
Error type B at line 5: Syntax error
```

样例输入 3

```
int inc()
{
    int i;
    i = i + 1;
}
```

样例输出 3

这个例子非常地简单，也没有任何词法、语法错误，其对应的输出应为：

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: inc
        LP
        RP
      CompSt (2)
        LC
        DefList (3)
          Def (3)
            Specifier (3)
              TYPE: int
            DecList (3)
              Dec (3)
                VarDec (3)
                  ID: i
            SEMI
        StmtList (4)
          Stmt (4)
            Exp (4)
              Exp (4)
                ID: i
              ASSIGNOP
              Exp (4)
                Exp (4)
                  ID: i
                PLUS
                Exp (4)
                  INT: 1
            SEMI
        RC
```

样例输入 4

```
struct Complex
{
    float real, image;
};
int main()
{
    struct Complex x;
    y.image = 3.5;
}
```

样例输出 4

这个程序虽然包含语义错误（使用了未定义的变量 y），但不存在任何词法和语法上的错误，因此你的程序不能报错而是要输出语法树信息。至于把这个语义错误检查出来的任务，我们将放到下一次实验中来做。本样例输入所对应的正确输出应为：

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        StructSpecifier (1)
          STRUCT
          OptTag (1)
            ID: Complex
          LC
          DefList (3)
            Def (3)
              Specifier (3)
                TYPE: float
              DecList (3)
                Dec (3)
                  VarDec (3)
                    ID: real
                COMMA
                DecList (3)
                  Dec (3)
                    VarDec (3)
                      ID: image
              SEMI
          RC
      SEMI
```

```
ExtDefList (5)
  ExtDef (5)
    Specifier (5)
      TYPE: int
    FunDec (5)
      ID: main
      LP
      RP
    CompSt (6)
      LC
      DefList (7)
        Def (7)
          Specifier (7)
            StructSpecifier (7)
              STRUCT
              Tag (7)
                ID: Complex
          DecList (7)
            Dec (7)
              VarDec (7)
                ID: x
          SEMI
      StmtList (8)
        Stmt (8)
          Exp (8)
            Exp (8)
              Exp (8)
                ID: y
              DOT
              ID: image
            ASSIGNOP
            Exp (8)
              FLOAT: 3.500000
          SEMI
      RC
```

分组内容样例

样例输入 1

```
int main()
{
    int i = 0123;
    int j = 0x3F;
}
```

样例输出 1

对于**需要**完成分组1.1的同学，这个样例程序不包含任何词法、语法错误，其对应的输出为：

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: main
        LP
        RP
      CompSt (2)
        LC
        DefList (3)
          Def (3)
            Specifier (3)
              TYPE: int
            DecList (3)
              Dec (3)
                VarDec (3)
                  ID: i
                ASSIGNOP
                Exp (3)
                  INT: 83
            SEMI
          DefList (4)
            Def (4)
              Specifier (4)
                TYPE: int
              DecList (4)
                Dec (4)
```

```
            VarDec (4)
              ID: j
            ASSIGNOP
            Exp (4)
              INT: 63
          SEMI
      RC
```

样例输入 2

```
int main()
{
    int i = 09;
    int j = 0x3G;
}
```

样例输出 2

对于**需要**完成分组1.1的同学，样例程序中“09” 为错误的八进制数，其会因被识别为十进制整数“0”和“9”而引发语法错误；同样，“0x3G”为错误的十六进制数，其也会因被识别为十进制整数“0”和标识符“x3G”而引发语法错误。因此你的程序可以输出如下错误提示信息：

```
Error type B at line 3: Syntax error
Error type B at line 4: Syntax error
```

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

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

样例输入 3

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

样例输出 3

对于**需要**完成分组1.2的同学，这个样例程序不包含任何词法、语法错误，其对应的输出为：

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: main
        LP
        RP
      CompSt (2)
        LC
        DefList (3)
          Def (3)
            Specifier (3)
              TYPE: float
            DecList (3)
              Dec (3)
                VarDec (3)
                  ID: i
                ASSIGNOP
                Exp (3)
                  FLOAT: 0.000105
            SEMI
        RC
```

样例输入 4

```
int main()
{
    float i = 1.05e;
}
```

样例输出 4

对于**需要**完成分组1.2的同学，样例程序中“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错误提示信息：

```
Error type B at line 3: Syntax error
```

注意，你的程序也可以将“1.05e”识别为错误的指数形式的浮点数，此时你的程序可以输出如下词法错误提示信息：

```
Error type A at line 3: Illegal floating point number '1.05e'
```

样例输入 5

```
int main()
{
    // line comment
    /*
        block comment
    */
    int i = 1;
}
```

样例输出 5

对于**需要**完成分组1.3的同学，这个样例程序不包含任何词法、语法错误，其对应的输出为：

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: main
        LP
        RP
      CompSt (2)
        LC
        DefList (7)
          Def (7)
            Specifier (7)
              TYPE: int
            DecList (7)
              Dec (7)
                VarDec (7)
                  ID: i
                ASSIGNOP
                Exp (7)
                  INT: 1
            SEMI
        RC
```

注意，我们的测试用例中“//”注释的范围和“/* */”注释的范围不会重叠，因此你不需要处理“//”注释范围和“/* */”注释范围重叠的情况。

样例输入 6

```
int main()
{
    /*
      comment
      /*
         nested comment
      */
    */
    int i = 1;
}
```

样例输出 6

样例程序中程序员主观上想使用嵌套的“/* */”注释，但是C--语言不支持嵌套的“/* */”注释。因此对于**需要**完成分组1.3的同学，样例程序中第8行的“*/”会因被识别为乘号“*”和除号“/”而引发语法错误（你的程序只需报一次语法错误），你的程序可以像这样输出一行错误提示信息：

```
Error type B at line 8: Syntax error
```

附　C--语法补充说明

Tokens

- 这一部分产生式主要与词法有关
- 词法单元 INT 表示的是所有的（无符号）整数字面常量。一个十进制整数由一串数字（0~9）组成，数字与数字中间没有如空格之类的分隔符。除“0”外，十进制整数的首数字不为 0。例如，下面几个串都是**十进制**整数：0、234、10000。方便起见，你可以假设（或者只接受）输入的所有整数都在 32bits 之内。

  整数字面量还可以以八进制或者十六进制形式出现。八进制整数由 0~7 八个数字组成并以数字 0 开头，十六进制整数由 0~9，A~F（或 a~f）十六个数字组成并以 0x 或者 0X（其中的 0 同样为数字 0）开头，例如：0237（表示十进制的 159）、0xFF32（表示十进制的 65330）。
- 词法单元 FLOAT 表示的是所有（无符号）浮点数字面常量。一个浮点数由一串数字与一个小数点组成，小数点的前后必须有数字出现。例如，下面几个串都是浮点数：0.7、12.43、9.00。方便起见，你可以假设（或者只接受）输入的浮点数都符合 IEEE754 单精度标准（即都可以顺利转换成 C 语言中的 float 类型）。

  浮点数字面量还可以以指数形式（即科学记数法）表示：指数形式的浮点数**必须**包括基数、指数符号、和指数三个部分，且三部分依次出现。基数部分由一串数字（0~9）和一个小数点组成，小数点可以出现在数字串的**任何**位置；指数符号为“E”或“e”；指数部分由**可以**带“+”或“-”的一串数字（0~9）组成，“+”或“-”**必须**出现在数字串之前。例如 01.23E12（表示 1.23×10^{12}）、43.e-4（表示 43.0×10^{-4}）、.5E03（表示 0.5×10^{3}）等等。
- 词法单元 ID 表示的是除去保留字之外的所有标识符。标识符可以由大小写字母、数字以及下划线组成，但必须以字母或者下划线开头。方便起见，你可以假设（或者只接受）标识符的长度小于 32。

  以上说明都仅仅给出了整数、浮点数和标识符具体词法的描述，但没有明确给出它们的正则表达式：你将自己决定在 Flex 中如何为整数、浮点数和标识符的识别书写规则。
- 除了 INT、FLOAT 和 ID 三个词法单元之外，其它产生式中箭头右边都代表了具体的字符串。例如 TYPE→int | float 这条产生式表示：输入文件中的字符串“int”和字符串“float”都将被识别为词法单元 TYPE。

High-Level Definitions

- 这一部分产生式包含了 C--语言中所有的高层（全局变量以及函数定义）语法。
- 语法单元 Program 是初始语法单元，代表整个程序。
- 每一个 Program 可以产生一个 ExtDefList，这里 ExtDefList 代表零个或多个 ExtDef（像这种 xxList 代表零个或多个 xx 的定义下面还有不少，要习惯这种定义风格）。
- 每一个 ExtDef 就代表着一个全局变量的定义、一个结构体的定义或者一个函数的定义。其中，
  - ExtDef→Specifier ExtDecList SEMI 这条产生式代表对全局变量的定义，例如 int global1, global2; 其中 Specifier 代表类型，ExtDecList 为零个或者多个对一个变量的定义 VarDec。
  - ExtDef→Specifier SEMI 这条产生式专门为结构体的定义而准备，例如 struct {…};
    当然这条产生式会允许出现“int;”这样没有意义的语句，但实际上在标准 C 中这样的语句是合法的。这种情况不作为错误的语法（即不需要报错），并且我们也不会测试。
  - ExtDef→Specifier FunDec CompSt 这条产生式代表函数定义：Specifier 是返回类型，FunDec 是函数头，CompSt 代表函数体。

Specifiers

- 这一部分产生式主要与变量的类型有关。
- Specifier 是类型描述符，它有两种取值：一种（Specifier→TYPE）是直接变成基本类型 int 或 float，而另一种（Specifier→StructSpecifier）是变成结构体类型
- 对于结构体类型来说，
  - StructSpecifier→STRUCT OptTag LC DefList RC：这是定义结构体的基本格式，例如 struct Complex { int real, image;}。其中 OptTag 可有可无，因此也可以这样写：struct {int real, image;}。
  - StructSpecifier→STRUCT Tag：如果之前已经定义过了某个结构体，比如 struct Complex {…}，那么之后就可以直接使用该结构体定义变量，例如 struct Complex a, b;，而不需要重新定义这个结构体。

Declarators

- 这一部分产生式主要与变量和函数的定义有关。
- VarDec 代表了对一个变量的定义。一个变量可以直接是一个标识符（例如 int a 中的 a），也可以是一个标识符后面跟若干个方括号括起来的数字（如 int a[10][2]中的 a[10][2]，这种情况下 a 是一个数组）。
- FunDec 代表了对一个函数头的定义。它包括一个代表函数名的标识符以及由圆括号括起来的一个形参列表，该列表由 VarList 表示（也可能为空）。VarList 可以包括若干个 ParamDec，其中每一个 ParamDec 都是对一个形参的定义，该定义由类型描述符 Specifier 和变量定义 VarDec 组成。一个完整的函数头的例子为：foo(int x, float y[10])

Statements

- 这一部分产生式主要与语句有关。
- CompSt 代表了一个由一对花括号括起来的语句块。语句块内部先是一系列变量的定义 DefList，然后是一系列的语句 StmtList。可以发现，对 CompSt 的这种定义是不允许在程序的任何位置定义变量的，必须在每一个语句块的开头才能定义。
- StmtList 就是零个或多个 Stmt 的组合。每一个 Stmt 都代表一条语句，这条语句可以是一个在末尾添了分号的表达式（Exp SEMI），可以是另一个语句块（CompSt），可以是一条返回语句（RETURN Exp SEMI），可以是一条 if-then 语句（IF LP Exp RP Stmt），可以是一条 if-then-else 语句（IF LP Exp RP Stmt ELSE Stmt），也可以是一条 while 语句（WHILE LP Exp RP Stmt）

Local Definitions

- 这一部分产生式主要与局部变量的定义有关。
- DefList 这个语法单元前面曾出现在 CompSt 以及 StructSpecifier 产生式的右边，说白了它就是一串像 int a; float b, c; int d[10];这样的变量定义。一个 DefList 可以由零个或者多个 Def 组成。
- 每一个 Def 就是一条定义，它包括一个类型描述符 Speicifier 以及一个 DecList，例如 int a, b, c;。由于 DecList 中的每一个 Dec 又可以变成 VarDec ASSIGNOP Exp，这说明我们是允许对局部变量在定义时进行初始化的，例如 int a = 5;。

Expressions

- 这一部分产生式主要与表达式有关。
- 表达式可以演化出的形式多种多样，但总体上看不外乎下面几种：
  - 包含二元运算符的表达式包括赋值表达式（Exp ASSIGNOP Exp）、逻辑与（Exp AND Exp）、逻辑或（Exp OR Exp）、关系表达式（Exp RELOP Exp）以及四则运算表达式（Exp PLUS Exp 等）。
  - 包含一元运算符的表达式包括括号表达式（LP Exp RP）、取负（MINUSExp）以及逻辑非（NOT Exp）。
  - 不包含运算符但又比较特殊的表达式包括函数调用表达式（带参数的 ID LP Args RP 以及不带参数的 ID LP RP）、数组访问表达式（Exp LB Exp RB）以及结构体访问表达式（Exp DOT ID）。
  - 最基本的表达式包括整数常量（INT）、浮点数常量（FLOAT）以及普通变量（ID）。
- 语法单元 Args 代表实参列表，每一个实参都可以变成一个表达式 Exp。
- 由于表达式中可以包含各种各样的运算符，为了消除潜在的二义性问题，我们需要对这些运算符规定优先级（precedence）以及结合性（associativity）。

Precedence[1]	Operator	Associativity	Description
1	()	Left-to-right	Function call or parenthesis
	[]		Array subscripting
	.		Structure element selection by reference
2	－(Unary)	Right-to-left	Unary minus
	!		Logical NOT
3	*	Left-to-right	Multiplication
	/		Division
4	+		Addition
	—		Subtraction
5[2]	<		Less than
	<=		Less than or equal to
	>		Greater than
	>=		Greater than or equal to
	==		Equal to
	!=		Not equal to
6	&&		Logical AND
7	||		Logical OR
8	=	Right-to-left	Direct assignment

Comments

C--代码可以使用两种风格的注释方式：一种是使用双斜线“//”进行单行注释，在这种情况下该行在“//”符号之后的所有字符都将作为注释内容而直接被词法分析器丢弃掉；另一种是使用“/*”以及“*/”进行多行注释，这种情况下在“/*”与之最先遇到的“*/”之间的所有字符也都被视作注释内容。需要注意的是，“/*”与“*/”是不允许嵌套的：即在任意一对“/*”和“*/”之间不能再包含成对的“/*”和“*/”，否则编译器需要进行报错。

[1]这里数值越小代表优先级越高。
[2]标准 C 语言中，大于、小于、大于等于和小于等于四个关系运算符的优先级要比等于和不等于两个运算符的优先级高。这里我们为了方便处理而将它们的优先级统一了。