程序设计语言原理

Principle of Programming Languages

裘宗燕

北京大学数学学院

2012.2～2012.6

5. 基本操作和控制

- 表达式和语句
- 表达式构造
- 求值过程
- 基本语句
- 控制语句
- 输入输出和文件

表达式和语句

表达式和语句是常规语言里描述计算过程的两个最基本层次。它们描述计算的执行顺序，实现语言的基本操作语义

- 表达式描述计算值的过程，常见控制手段是优先级、括号等
- 语句是命令，基本语句是程序里的基本动作。常规命令式语言中与数据有关的最基本动作主要的就是赋值
- 语句层控制提供一批控制结构，每种控制结构产生一种特定计算流程
  - 产生一些规范的计算序列（如条件分支、循环等）
  - 一些机制（如**break**，**continue**），用于改变规范的计算序列
- 程序员可以通过不同结构的组合应用，实现应用所需的特定控制流程

一些语言中，语句和表达式之间的界限模糊

如 **Algol 68** 的各种结构都有值，都看成表达式，**C** 语言最主要的基本语句是表达式语句。在函数式语言（如 **Lisp**、**ML** 等）里一切都是表达式

表达式

表达式是描述计算的最基本手段，它们描述值的计算过程

作为一个抽象层，表达式使程序员摆脱了许多具体实现细节：

- 计算过程中如何调配和使用 **CPU** 里的寄存器（紧缺资源）
- 中间结果存放在哪里？如何存放？
- 是否可能重新整理表达式以减少存储用量、提高速度，等等

形式上，表达式采用类似数学表达式的记法：

- 基本运算对象：变量、常量（文字量）等
- 表达式构造：使用运算符和函数等

对表达式计算过程进行控制的手段是一组规定，常见的有：

优先级　　结合顺序　　括号　　运算对象的计算顺序

前 **3** 项大家都比较熟悉，最后一个问题也很重要，但经常被忽视

表达式：求值和副作用

表达式是一种抽象，理解程序里表达式的意义，要考虑两点：

- 表达式实现的计算过程（完成计算的顺序，决定的求值过程）
- 表达式的求值过程对运行环境的影响

如果一个表达式的求值对环境没有任何影响，就其称为引用透明的。具有引用透明性的表达式，其求值时间早晚对其他计算不产生任何影响

如果一个表达式的计算不仅求出了一个值，还会造成环境的改变，就说它有副作用（**side-effect**）。例如 C 语言里的 **... a++ ...**

在纯函数式语言里，所有表达式都是引用透明的。程序的语义很清晰

曾有些语言设计者倡导完全禁止有副作用的表达式（可以做到）

实际上多数语言里都可以写带有副作用的表达式。有客观需要，例如：

- 允许函数调用可以有副作用（典型：随机数生成函数）
- 输入/输出函数，需要有副作用

表达式：表示形式

在不同语言里，表达式的表示形式可能不同

- 最常见的是中缀形式（需要引进括号描述计算顺序。为减少括号使用，通常定义一套优先级规则）
  - **Smalltalk** 的运算符无优先级（所有运算符的优先级相同）
- 一些语言采用后缀形式（运算符在运算对象后面，不需要括号和优先级规定，只需规定每个运算符的元数），例如 **Forth**，**Postscript** 等
- 一些语言采用前缀形式或其变形（运算符出现在运算对象前面，不需要括号和优先级，只需规定某个运算符的元数）。例如 **Lisp**：

```
(+ (* (- a 4) (+ b 5)) (- c 2))
```

- 大多数语言采用的实际上是混合形式
  - 函数调用 `f(a, b, c)` 就是前缀形式
  - 特殊运算符（例如，三元运算符 `x>2 ? y : z` 等）

表达式：计算过程

理解表达式，首先要理解表达式确定的计算过程

大家都很熟悉的：优先级、结合顺序（左结合或右结合）和括号

现在讨论一下运算对象的求值顺序。例：

(a + b) * (c + d)　　**fun(a++, b, a+5)**

一些语言明确规定了二元运算符的运算对象、函数调用的实参表达式的特定计算顺序。例如，**Java** 明确规定从左到右计算各运算对象和函数参数

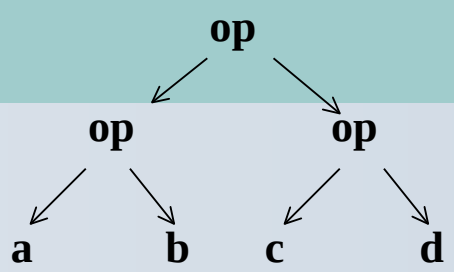
多数语言对运算对象的求值顺序“不予规定”，目的是允许编译器采用任何求值顺序，使编译表达式时可能做更多优化。例如 **C/C++**

例：谁知道下面 **C** 语句给 **n** 赋什么值？

m = 1; n = m+++m++; /* 也就是 n = m++ + m++; 最长可能原则 */

正确回答：不知道！牵涉到运算对象求值顺序以及值更新方式（下面讨论）

表达式：求值规则



表达式可以用树形结构精确表示，其他形式（中缀/前缀等）都是树形表示的某种平坦化

我们可以在树形表示上讨论求值规则

常规求值规则采用树上的后序遍历顺序：

在对一个运算符为根的子表达式进行求值前，先完成其各子树的求值

称为先行求值或积极求值（**eager evaluation**），也称应用序求值（先求出运算对象，再应用运算符/函数）。对算术表达式，这种方式很自然。

有时希望采用其他求值方式。例：

```
z = (y == 0 ? x : x/y);
```

如果采用应用序求值，`y` 等于 `0` 时，语句执行就会出错

正则序求值：在需要用表达式里的某部分时才对这个部分求值，其余部分则处于未求值的状态。也称拖延求值或懒求值（**lazy evaluation**）

表达式：求值规则

Pascal 完全采用应用序求值。有些表达式不好表示。如数组上的循环：

```
while i <= n and a[i] > 0 do begin
   ...a[i]...; ...; i:=i+1
end
```

计算中会出错（数组越界）。正确写法要增加嵌套的条件语句：

```
while i <= n do begin
   if a[i] > 0 then begin
      ... a[i] ...; ...;
   end;
   i:=i+1;
end ...
```

类似情况很多，例如在遍历链接表、树上周游时写下面条件都不行：

```
p <> NIL and p^.data > 0
```


表达式：求值规则

C/C++ 对大部分运算符采用应用序求值，而条件运算符和逻辑运算符采用拖延求值规则，只计算必要的运算对象。常见：

```
for (i=0; i<n && a[i]!=0; ++i) ... a[i] ...
for (p = L; p != NULL && p->n > 0; p = p->next) ...
```

如果根据运算符左边对象的值已经可以确定最终结果，就不再计算右边的运算对象，这种方式也称为短路求值

- C/C++/Java 等的逻辑运算符都采用短路求值。在表达式值仅为 0 或 1 的情况下，可以用按位运算符模拟非短路逻辑运算符
- Ada 语言提供了两套逻辑运算符，分别实现短路和非短路的逻辑运算

  and，or 采用普通的应用序求值规则

  and then ， or else 采用短路求值规则

表达式：副作用及其实现

如果程序里出现了修改环境的动作，什么时候（在此后程序里的什么位置）能看到这个动作的效果？（这是问题吗？）

例：设程序里有 **a[i]++ ... a[j] ...**，假定当时 **i** 与 **j** 的值恰好相等，且 **a[i]++** 在 **a[j]** 之前做。问题：**a[i]++** 对 **a[i]** 的修改能反映到 **a[j]** 的求值中吗？

（由于 **i** 与 **j** 相等不能静态判定，两个访问需要通过独立代码完成）

CPU 里的计算要在寄存器里做（现代计算机都如此），问题是

取 **a[j]** 的值之前，是否已经把 **a[i]** 的新值保存到内存里 **a[i]** 的位置

程序语言通常规定了变量修改的最晚实现时间（称为执行点或顺序点）

- 编译器保证，在程序执行到达每个顺序点的时刻，此前出现的所有修改都会体现到内存里，此后出现的任何修改都没有发生
- 程序执行在两个顺序点之间，就不能保证发生的修改能得到体现

表达式：副作用

不同语言对于顺序点有不同的定义

如果语言允许表达式有副作用（大部分语言里都允许），顺序点的概念就显得特别重要：

- 表达式的副作用是否影响上下文中的其他计算动作（其他表达式的求值）
- 从什么时候开始影响？（必须明确知道，不能依靠没有保证的东西）

C 语言表达式可能有副作用，规定程序中的顺序点为：

- 完整表达式的结束位置。包括变量初始化表达式，表达式语句，**return** 的表达式，条件、循环和 **switch** 的控制表达式（**for** 头部有三个表达式）
- 运算符 **&&**、**||**、**?:** 和逗号运算符（**,**）的第一个运算对象之后
- 函数调用中对所有实参和函数名表达式（要求调用的函数也可能通过复杂的表达式来描述）的求值完成之后（进入函数体之前）

表达式：副作用

Java 语言的表达式可能有副作用。定义：

- 运算对象或者参数总是从左到右逐个计算
- 每个计算的副作用将立即体现（每个计算动作之后都有顺序点）

显然，Java 程序执行中的顺序点大大多于 C 程序

- 顺序点就是寄存器和内存的同步点，需要把修改过的寄存器内容同步到内存。寄存器速度是 ns 级，内存延时在100ns量级，差几十倍
- 语言的顺序点较稀疏，使计算可以在更长期间内一直在寄存器组里进行，因此有可能做更多优化。程序可以在两个顺序点之间的任何适当时刻把结果存入内存，因此带来语义的“不确定性”（在两个顺序点之间）
- 为频繁实现计算的效果，必须执行更多的内存保存和寄存器加载指令，带来效率损失，同时也带来更高的语义确定性

实现效率 ⟷ 语义的确定性

表达式：代码改进（优化）

- 与处理器速度相比，内存速度太慢。访问内存通常会导致处理器停下来等数据，频繁访问内存将大大影响处理速度
- 提高程序的执行效率，最重要的技术之一就是充分利用寄存器

  寄存器是处理器里的紧缺资源，需要对其使用有很好的安排

  m * n + f(a) 如果先计算 f(a) 有可能减少访问内存的次数
- 子表达式中的求值顺序可能影响寄存器分配和指令调度
  - 为代码优化提供更多的可能性，是许多语言（尤其是 C 语言）等里面不完全精确地定义运算对象求值顺序的主要原因
  - 优化编译程序可能在程序里挖掘各种可能的值共享，在可能情况下尽量减少内存操作（保存和装载）
  - Java 的严格求值顺序将大大限制优化的可能性， Java 语言的这一特征使得程序不可能做深度优化，必然效率较低

表达式：执行时形式

常见的表达式执行时形式有几种：

- 机器代码序列：一个表达式被翻译为一段机器指令，求值时直接执行这段指令。速度快，但固定（是常规语言的运行时形式，C，Fortran 等）
- 采用表达式的树形表示，计算过程通过解释器的周游实现。效率低，但允许动态修改（Lisp 等，也常作为程序的调试时表示）
- 某种抽象机语言，用一个抽象机解释（例如作为 Java 程序的编译结果的字节码程序和抽象机 JVM）
- 采用前缀或后缀形式的某种内部表示（后缀形式更多见），可以用一个简单的解释器求值（用一个或几个栈。如 Forth 等语言）
- 混合形式

表达式：错误处理

表达式求值中可能出现错误，典型的：

- 数值溢出、除 0 错
- 函数执行中出错，动态分配存储时出错等

不同语言对这些情况有不同的规定

- 运行中不动态检查也不处理，完全由程序员负责（例如， C 语言规定出现除 0 时的“结果”无定义）
- 动态检查求值中的错误。规定出错时程序非正常终止并报告错误，具体实现可以提供与运行平台有关的处理机制，用于截获和处理错误
- 提供一套完整的处理机制。运行中出错时，运行系统自动生成与错误事件有关的信号，语言提供捕获和处理这种信号的机制

人们早已认识到支持错误处理的重要性，最新的语言多采用最后一种方式，为此开发的一套机制称为异常处理，后面有详细讨论

基本语句：赋值

赋值是最基本的操作，前面已仔细讨论过

一种相关概念是复合赋值运算符

下面这类形式的赋值语句写起来困难，特别难看，也很容易写错（需要仔细检查两边是否一致）

counter := counter * 5

current^.data[j + 3].number := current^.data[j + 3].number + 1

两次访问不但可能造成重复计算开销，稍微不慎还可能造成隐藏的错误，写出不安全的程序。例，下面语句可能是错（如果表达式求值有副作用）：

a[fun1(num)] := a[fun1(num)] + 1

如果 **fun1(num)** 的求值有副作用，就应该改写为：

n := fun1(num); a[n] := a[n] + 1

复合赋值运算符

从 **Algol 68** 开始，人们为处理这类情况引进了一组更新运算符。形式如：

counter *:= 5

current^.data[j + 3].number +:= 1

C 语言的形式大家都很熟悉了。**C** 还为使用最频繁的加一和减一专门地各引进了一对运算符，前缀和后缀的 **++ / - -**

Algol 68 和 **C** 的赋值符都是运算符，赋值是一种表达式

基本赋值操作构造出的表达式是有副作用的表达式，在关注其副作用（赋值的效果）的同时还要关注其值。例如，下面表达式的意义是什么：

a[n] = b[n++]; 在 **C** 语言里，这一语句的意义没有定义

一些语言里提供了另一种形式的复合赋值运算符，如

a, b := c, e; a, b := b, a

基本语句和控制结构

赋值之外的基本操作包括输入输出。一些语言提供了专门语句，一些语言通过库实现。输入和赋值一样可以改变变量的状态

- 在不同语言之间，输入输出功能的设计差异很大。主要问题是灵活性，支持复杂的格式控制，类型安全性和易用性
- 图形用户界面给语言的输入输出功能设计提出了许多新问题
- 有关输出输入的问题在本章最后讨论

基本语句之上的控制结构是人们最熟悉的东西，也是语言中最清楚的结构

- 所有的流程控制语句都是结构化的受限的直接控制转移（goto）
- 一种控制结构形成某种较规范的控制流，能取代 goto 的一些使用
- 一些新控制语句的加入已经使 goto 语句彻底出局

  许多新语言完全删除了 goto， Java 是其中使用最广泛的一个

语句：goto

最原始的流程控制——标号和 goto ——是基本硬件控制机制的直接反映

无条件/条件转移是机器语言中的基本控制手段。最早的高级语言（Fortran）就提供了标号和 goto 语句

随着人们对程序中常见控制流程模式的认识，归纳出许多规范的流程，提出了许多更高级、更结构化的控制机制，逐步削减 goto 的领地：

- 许多 goto 是为了根据情况选择性地执行一段代码，或者从两段代码中选择执行 －－ if 语句，case/switch 语句等更清晰明确
- 许多 goto 是为实现在一定条件下重复执行一段代码，或者重复执行一段代码若干次 －－ 各种循环语句更清晰明确

剩下有用的 goto 就是为了在某些特殊位置改变规范的控制流

- 退出循环或者其他控制结构（break 等）
- 根据需要任意地改变控制流（现在已经无人提倡了）

语句：结构化革命

导致结构化程序设计革命的事件：

- C. Bohm 和 G. Jacopini 证明（1966），任何流程图程序都可以变换为等价的只使用顺序和 while 结构的程序（提供图灵机的计算能力）
- Dijkstra（1968）给 CACM 编辑的信“goto 是有害的”，向 goto 宣战

争论：

- 保守派：goto 很有用，没有 goto 许多事情就没法做了。或者导致很别扭的程序，或者导致程序效率下降，或者导致程序变得更复杂
- 造反派：goto 很有害，没有 goto 不但什么都能做，而且做得更好

在这个讨论中开发出许多想法和有关语言控制结构的建议

结构化程序设计革命，大家都认可 goto 是有害结构，应尽量少用

新语言都采纳了结构化的控制结构，基本上已经形成标准：单支和两分支的 if（及多分支选择），枚举循环 for 和逻辑循环 while 等等

语句：消除 goto

今天，人们已经把由顺序、条件和循环结构形成的控制流看作是正常控制流，其他局部控制机制是改变正常控制流的手段

正常控制流的遗留问题：	机制：
• 从循环体中间退出或继续	break（exit）语句，和 continue 语句
• 直接退出多层循环	为结构化语句引入名字（标志），通过带标志 break（exit）退出任意深的嵌套循环
• 子程序的提前退出	引入 return 语句，允许在子程序中的任何地方调用 return 语句
• 出错处理（例：Pascal 等允许从被调子程序转移到某个调用子程序	引入高级的非局部转移机制，目前应用最广泛的是结构化的异常处理机制（后面讨论）

今天，我们已经可以完全摆脱 goto 这种低级控制机制了

当然，没有 goto 的程序并不是有 goto 程序的简单翻译

语句：结构化

任意控制流的缺点：

- 程序静态结构无层次性，意义难以把握，容易隐藏危险的错误
- 静态结构与动态执行流之间没有清晰对应，实际执行流程可能很混乱
- 一组语句可能处于多条控制流程中，有许多不同的用途，使程序的理解、修改变得异常困难，更难以证明程序的正确性

结构化程序设计的思想：

- 清晰的分层结构
- 程序描述与执行流程的清晰对应
- 一个语句组（基本代码块）服务于单一的目的

控制结构设计：

提供一组控制语句，使之能反应结构化程序设计的思想，又能较好地满足实际程序设计工作的需要，还要考虑实现效率问题

语句

基本控制结构：顺序、选择、循环

顺序结构（复合语句）是最基本的最简单的控制结构（硬件里的默认控制）

不同语言的差异：

- 分号作为语句分隔符（**Algol 60**、**Pascal** 等），还是作为语句结束符（**C**、**Ada**、**Java** 等）
- 用 **begin ... end** 或者 **{ ... }** 作为界定标志

顺序语句的实现极其简单（完全不必有专门的实现）

基本的选择结构是根据逻辑条件选择，另一种是根据（整数）值选择

循环结构的变形较多，分为根据逻辑条件和采用计数方式的两类