语句：选择（if）

早期 Fortran 的 IF 语句是硬件条件转移的直接包装

Algol 60 引进了今天的单分支和两分支 if 语句，以逻辑条件选择，两个分支都可以是任意的完整结构

```
if ... then ...
if ... then ... else ...
```

在这里第一次遇到语言（分析）的歧义性问题。下面语句是什么意思：

```
if B1 then if B2 then S1 else S2
```

最后的 else 部分属谁？这就是著名的“悬空的 else问题”（dangling else）

三种解决方案：

- 不允许 then 部分出现 if 语句（Algol 60 等）
- 规定 else 部分与最近的无 else 的 if 匹配（Pascal、C、Java 等）
- 为 if 增加结束标志（用 fi，end，end if 等。Ada、Algol 68 等）

语句：选择（if）

if 语句的实现：结合条件转移和无条件转移

虽然条件是逻辑表达式，但用途不是求值而是控制执行流，使转到适当位置

```
if ((A > B) and (C > D)) or (E <> F) then
    then_clause
else
    else_clause
```

短路实现

```
    r1 := A
    r2 := B
    if r1 <= r2 goto L4
    r1 := C
    r2 := D
    if r1 > r2 goto L1
L4: r1 := E
    r2 := F
    if r1 = r2 goto L2
L1: then_clause
    goto L3
L2: else_clause
L3:
```

非短路实现

```
    r1 := A          -- load
    r2 := B
    r1 := r1 > r2
    r2 := C
    r3 := D
    r2 := r2 > r3
    r1 := r1 & r2
    r2 := E
    r3 := F
    r2 := r2 <> r3
    r1 := r1 | r2
    if r1 = 0 goto L2
L1: then_clause      -- (label not actually used)
    goto L3
L2: else_clause
L3:
```

语句：选择（case）

提供 case（switch）语句是为了一大类选择控制流的高效实现

早期：

- Fortran 的计算 goto。（根据变量 I 的值为1, 2, 3, ... 转跳到对应标号）

```
goto (15, 30, 100, 203), I
```

- Algol 60 的 switch 语句（标号数组和按下标取标号）

```
switch S := L1, L10, L23, L38;
... ...
goto S[i];
```

主要想法：基于整数值，得到快速的控制转移

不需要一个个比较数值，只需按位置找转移目标，即可立即转过去

今天一些语言里的 case（switch）语句是这种想法的结构化变形

Hoare 提出了 case 的想法，Wirth 在 Algol W 里第一次提供这种结构

case实现

```
CASE ...  (* potentially complicated expression *) of
    1:      clause_A
|   2, 7:   clause_B
|   3..5:   clause_C
|   10:     clause_D
    ELSE    clause_E
END
```

通过普通转跳码实现，代码见右框：

```
      r1 := ...                    -- c
      if r1 <> 1 goto L1
      clause_A
      goto L6
L1:   if r1 = 2 goto L2
      if r1 <> 7 goto L3
L2:   clause_B
      goto L6
L3:   if r1 < 3 goto L4
      if r1 > 5 goto L4
      clause_C
      goto L6
L4:   if r1 <> 10 goto L5
      clause_D
      goto L6
L5:   clause_E
L6:
```

重要特征：

- 完成整个语句可能要做许多比较
- 执行进入越在后面的分支，为确定执行路径付出的代价越大

case实现

```
CASE ...  (* potentially complicated expression *) of
    1:      clause_A
|   2, 7:   clause_B
|   3..5:   clause_C
|   10:     clause_D
    ELSE    clause_E
END
```

```
T:  &L1              -- tested expression = 1
    &L2
    &L3
    &L3
    &L3
    &L5
    &L2
    &L5
    &L5
    &L4              -- tested expression = 10
L6: r1 := ...        -- calculate tested expression
    if r1 < 1 goto L5
    if r1 > 10 goto L5     -- L5 is the "else" arm
    r1 -:= 1
    r2 := T[r1]
    goto *r2
L7:
```

用转跳表技术实现

```
    goto L6
L1: clause_A
    goto L7
L2: clause_B
    goto L7
L3: clause_C
    goto L7
    ...
L4: clause_D
    goto L7
L5: clause_E
    goto L7

L6: r1 := ...
    goto *r1
L7:
```

一般的5分支case结构

语句：case实现

实际程序里的 **case** 语句，可能根据 **case** 标号（整数值标号）的情况而编译为不同的形式，由编译器自动选择：

- 标号密集：编译到转跳表，速度快
- 标号不密集，但个数少：编译到普通的转跳码序列
- 标号不密集（存在较大的标号值），且标号较多
  - 建立排好序的标号表，通过折半查找，选择转跳
  - 建立散列表，通过散列函数，选择转跳

不同编译器的优化能力不同，产生的结果效率差别可能很大

为支持 **case** 的高效实现和方便使用，不同语言的 **case** 语句设计也不同

- **case** 标号是否允许子界？
- 有无默认选择（**default**，**others** 子句）
- 如果选择表达式算出的值没有对应的标号，如何处理？（默认选择？）

语句：循环（迭代）

没有循环（也没有递归）的程序是平凡的程序。

- 循环和递归给了计算机生命力
- 计算机的威力就在于它能反反复复地做一些事情
- 反复做之中还可以有变化

语言中的循环结构分为两种：

- 枚举控制的循环：对某个有限集合里的每个元素执行循环体一次
- 逻辑控制的循环：执行到某个逻辑条件变了

一些语言里这两种循环共有同一种形式，如 **Algol 60**，**C** 的 **for** 语句

循环：枚举控制

最早的 **Fortran** 只有枚举控制的循环：

```
      do 10 i = 1, 12, 2
      ... ...
10    continue
```

Pascal：　　没有步长，只能用步长 **1** 或 **-1**（**downto**）

```
FOR i := 1 TO 12 DO n := n + i
```

Modula：　　可以有步长

```
FOR i := 1 TO 12 BY 2 DO n := n + i END
```

Ada：　　没有步长，只能用步长 **1** 或 **-1**（**reverse**）

```
FOR i in 1..12 loop n := n + i; end loop
```

允许多种形式的枚举描述，包括针对数组下标的 **a'range**

循环：枚举控制

枚举循环的问题：

- 循环变量是否需要另行定义，如不需要，其作用域是否延伸到循环之后？
- 循环变量在循环体里的作用：看作常量？看作普通变量（允许修改）？
- 循环的上下界表达式什么时候求值？
  - 循环开始前求值一次，以后作为常量
  - 每次迭代之前判断循环是否应该结束时重新求值
- 循环结束后循环变量的值是什么？
- 是否允许以其他方式转入或者转出循环
  - 用 **goto** 转入，循环变量的值是什么？
  - 转出后能否访问循环变量？

```
for (short i = 0; i < 32700; i += 128) ...
```

设 `short` 为16位（注意什么时候条件不成立）

循环：枚举控制（实现）

```
for i := first to last by step do
    ...
end
```

```
    r1 := first
    r2 := step
    r3 := last
L1: if r1 > r3 goto L2
    ...
    r1 := r1 + r2
    goto L1
L2:
```

```
    r1 := first
    r2 := step
    r3 := last
    goto L2
L1: ...
    r1 := r1 + r2
L2: if r1 <= r3 goto L1
```

- 应注意 **0** 次循环的情况
- 循环结束时循环变量的值
- 如果考虑值可能超越类型的最大值的问题，实现会更复杂一些

循环：枚举控制的推广——迭代器

枚举循环的控制结构：

- 是一种能生成一系列离散值的机制
- 第一步总是初始化，建立循环的初始状态（设置循环变量初值）
- 提供一种操作，执行一次将得到一个新的（枚举）值
- 提供一种操作，通过它可以判断还有没有更多的枚举值

任何能提供这些功能的机制都可以称为是一种迭代器，通过“迭代器”机制控制循环，是程序里的一种很常见的“描述模式”

- 一些语言里提供了定义迭代器的专门机制（**Pragmatics** 书 **6.5.3**）

在其他语言里，可以通过模拟来实现迭代器，例如：

- **C** 语言标准库的 **stdarg** 功能（变长度参数表）就是一种迭代器
- **C++** 标准库的许多功能依赖于迭代器的概念

循环：逻辑控制

逻辑控制的循环，通过逻辑表达式的真假值来控制循环的继续或结束

- **Algol 60** 引进 **while** 概念，但是所提供的逻辑循环的形式不好
- 后来人们总结出 **while B do S** 的标准形式，被广泛接受和使用

后检测循环：

有时需要写：`S; while B do S` 同样计算需要描述两次，不好

- 把循环条件的检测放在循环体之后，这样的循环至少执行一次
- 这种循环有两种不同设计：
  - 描述终止条件，如 **Pascal** 的“直到循环”（**repeat-until** 循环）

    repeat ... ... until *condition*
  - 描述继续条件，如 **C** 语言的 **do-while** 循环

    `do` *statement* `while (`*condition*`);`

循环：逻辑控制

中间检测的循环：

- 有时需要在循环体的中间判断是否应结束循环
- 直接用 **while** 写，需要加入包裹后面部分的条件语句
- 如果出现多处，程序很难看

个别语言设计了专门的中间检查机制。如 **Modula**：

loop
... ...
when ***condition*** **exit**
... ...
when ***condition*** **exit**
... ...
end

这种机制比较清晰，但不能支持从嵌套的循环中退出（**exit** 只在循环的表层）

目前多数语言是另行提供退出机制，一般提供退出循环的 **break** 或者 **exit** 语句

这类语句很方便，但缺乏结构性，描述不够明显，退出到哪里也不很清晰

通过带标号 **break** 或 **exit** 与结构标号配合，可实现退出任意结构的控制转移，实现从深层嵌套的（循环或非循环）结构直接退出

非确定性（Nondeterminacy）

非确定性控制结构，就是有意将不同控制路径之间的选择留下来不予确定

前面表达式求值中的求值顺序具有非确定性

一些早期语言提供了一些非确定性结构，如 **Algol 68** 的并列语句，允许块中用逗号分隔的几个语句按任意顺序执行

Dijkstra 明确提出将非确定性作为程序控制的重要概念，他在论文中提出了卫式命令的概念，并提出了两种具有非确定性的卫式结构

卫式选择，其一般形式是：

$$\texttt{if } g_1 \rightarrow S_1 [] \ldots []g_n \rightarrow S_n \texttt{ fi}$$

其中的 g_i 称为卫（**guard**）。这一结构的语义是：

如果某个卫求值得到真，就执行它后面的语句；如果当时有多个卫的值为真，在这些卫之间的选择为非确定的；如果没有卫为真就是动态错误（最后这条是 **Dijkstra** 原来的建议，后来多半没采纳）

非确定性

卫式循环，其一般形式是：

$$\texttt{do } g_1 \rightarrow S_1 [] \ldots []g_n \rightarrow S_n \texttt{ od}$$

语义：若存在卫的值为真的分支，就在这些分支中非确定性地选择一个执行相应语句，而后重新考虑这一循环；没有卫为真时循环结束

提供非确定性结构的一个考虑是为了方便形式化的语义研究，以及程序的形式化推导，**Dijkstra** 在其重要著作 **Discipline of Programming** 里提出了基于卫式命令进行程序推导的技术，给出了许多例子

另一个考虑是程序里的对称性。如求绝对值的程序片段可以写成：

$$\texttt{if } x\texttt{>=0} \rightarrow y := x []x\texttt{<=0} \rightarrow y := -x \texttt{ fi}$$

普通的 **if** 语句中没有这种对称性，必须强加人为的顺序，使其中的一个分支享有优先权

非确定性

非确定性最重要的应用在并发程序。例如

process client:
loop
  get command from user
  if read, send `read` **request to server**
    wait for response
  if write, send `write` **request to server**
    wait for response

process server:
loop
  receive `read` **request**
    reply with data
  OR
  receive `write` **request**
    update data and reply

作为服务器端的 **server** 必须同时（随时）准备好接受 **read** 和 **write** 请求，任何预先安排好的顺序都必然会在某种情况下导致系统“死锁”

对于这类情况，就必须有非确定性的选择控制结构

一些支持并发性的高级语言引进了基于卫式命令概念的控制结构，例如 **Ada** 等

IO（输入输出）

程序通常都需要与外界交换信息。与程序交换信息的情况多种多样：

- 交换信息的对象是作为用户的人（交互式 IO）
- 交换对象是特定设备（各种 IO 设备）
- 交换对象是环境中的文件（文件 IO）

IO 的难点在于实际需求极其繁杂，其中的细节和变化太多

有些语言干脆完全回避 IO 机制的设计（如 Algol 60 和老的 C 语言）

大部分语言都采用某种方式提供了 IO 功能

交互式 IO 的发展是图形用户界面系统

有些语言把图形用户界面系统纳入语言设计中，都是通过一个非常庞大的界面功能库的方式（例如 Java 和一些脚本语言）

至今尚未看到特别成功的设计（总是一套繁复的规定，使用非常麻烦）

IO 问题

输入输出操作的设计需要考虑许多问题，包括：

- 由于与程序的外部打交道而引起的问题
  - IO 操作的对象在程序之外，写程序的人无法控制其作为，为正确安全地编程，必须为检查和处理操作失败提供方便易用的手段
  - 外部信息源可能是合作的，不合作的，甚至敌对的。即使是合作信息源也可能出现无意错误。IO 系统的设计应有利于抵御各种 IO 错误
  - 外部信息源可能在任何时候失败或者出错
- 为有利于编程本身而引起的问题
  - 程序员要求简单易用的，又能满足其任意复杂的需求 IO 操作
  - 语言需要提供适当手段，使程序员可以方便地实现复杂的格式控制
  - 程序员希望能为自定义类型等方便地扩充基本的 IO 功能

输入操作比输出操作更难设计，编程时也更难使用

IO

程序语言里 I/O 功能设计的基本追求：

- 方便易用，描述的紧凑性（容易描述，简洁）
- 灵活的控制（容易进行复杂的格式控制）
- 类型安全（赋给变量的值一定类型正确，一定按类型正确的方式输出）
- 用户可扩充性（特别是现在的程序里有大量的用户定义类型。能否像内部类型一样用？是否能采用与内部类型一样的形式？）

很少有 IO 系统的设计能满足上述的所有要求

语言里 IO 功能的提供方式：

- 通过语言里的特殊机制：
  - 可采用专门设计的形式和机制，不受任何限制
  - 增加了语言本身的负担，难以扩充
- 通过库：形式必须符合语言中的一般规定，但较灵活、容易扩充

IO

越来越多的语言采用库的方式提供 IO 功能，包括 C、C++、Ada、Java等

- 通常都定义了标准 IO 库
- 设法允许用户扩充

文件 IO，需要提供一组文件操作功能（是 IO 之外的辅助功能）

- 文件打开关闭（创建文件缓冲区，维护当前操作位置——文件“指针”）
- 文件指针操作
- 其他文件操作（创建、删除、改名、建立临时文件等）

- 顺序文件：顺序输入输出
- 随机文件：允许随机的定位，对任意位置做输入和输出
- 二进制文件：直接输入输出，把数据的内存映像保存到文件或装入内存
- 正文文件（文本文件）：内部形式与正文形式之间的转换

IO 功能：Pascal

下面看几个典型语言的 IO 系统设计（主要看正文 IO）

Pascal：

- 把文件看作一种类似数组的类型构造　**cardfile : FILE of card**
- 语言内部的 IO 功能，提供了一组 IO 操作，每次可以输入输出任意多项。其中带文件名的版本用于文件 IO，不带的用于交互式 IO。例如

  read(x, y, z)　　readln　　write(fn, x, y)　　writeln
- 操作正文文件时，IO 过程可根据变量类型自动完成转换，类型安全
- 其中的变量必须是几个基本类型，或者是字符串（字符的紧缩数组）
- 没有格式控制，只能自己通过输出字符的方式控制
- 不能扩充即有的 IO 功能以支持用户定义类型
- 提供了直接 IO 和二进制文件的概念

IO 功能：Ada

- 通过库提供 IO 功能，提供了几个标准库 IO 包

  `sequential_IO`　`direct_IO`　`text_IO`　`low_level_IO`
- 借助子程序重载机制，针对每个内部类型有两个输出过程（一个带文件名，另一个不带）和两个输入过程。另外有一些带换行的过程。类型安全
- 每次子程序调用只能输出一个数据项，IO 功能较弱
- 格式控制的功能比较有限
- 利用过程的默认参数，为每种数值定义了默认的输出转换形式
  - 如果不提供附加参数，就按照默认形式输出
  - 可以通过提供格式参数要求得到所需形式的输出
  - 可以通过其他过程修改格式的默认值，从而影响随后的一系列输出
  - 频繁修改默认形式很麻烦
- 可以通过重载，为用户定义类型提供类似的 IO 功能

IO 功能：Ada

例子：

```
s : array (1..20) of character;
x : real;
a : array (1..10) of integer;
... ...
set_output(my_file);  -- 此后的输出目标是 my_file
put(s);
put(x, 10, 8, 2); -- 整数位数，小数位数，指数位数
for k in 1..10 loop put(a(k), 10); end loop
     -- 10 表示输出域的宽度
```

直接用文件输出函数：

```
put(my_file, s);
put(my_file, x, 10, 8, 2);
for k in 1..10 loop put(my_file, a(k), 10); end loop
```


IO 功能：C

- 利用库提供 IO 功能
- 为文本文件 IO 和交互式 IO 提供了两套不同的输入输出函数
- 提供了丰富而紧凑的格式控制功能
  - 一些数据类型有默认的输出方式，程序员可以自己另行设定
- 不检查实际的输出表达式或者接受输入的变量的类型，不能根据类型选择转换动作，因此其输入输出没有类型安全性
- 程序员可借助标准库功能为用户定义类型实现与标准库功能相当的 IO 函数，但不能用标准库 IO 函数输入输出用户定义类型的对象（不能给标准库 IO 函数增加用户定义类型 IO 功能，有些实现提供了这种功能）
- 利用特殊库功能（stdarg.h）可以实现参数个数可变的函数。通过这种功能定义的 IO 机制可一次输入输出任意多项数据（与 printf/scanf 类似）

IO：C++

除 C 语言的 IO 外，提供了流式 IO 库

- 通过综合利用多种机制，提供了一套统一的文本 IO 机制
  - 同样描述形式可用于交互式 IO 和文件 IO；可以为用户定义类型扩充服务于语言内部类型的 IO 功能
  - 利用运算符重载和解析实现类型的识别和处理，有类型安全性
  - 通过返回对象引用的方式支持级联式的输入或输出，因此可以较紧凑地描述一系列值的输入或者输出
  - 提供了两套格式控制机制：属性设置方式和操控符方式
    - 一些设置方式有持续性作用，一些只对一次实际 IO 起作用
    - 允许用户根据需要定义自己的操控符

C++ 的流式 IO 库基本上满足了前面提出的 IO 系统设计的全部追求

但非常复杂，一些机制重复（并与由 C 继承来的 stdio.h 库重复）

流库与直接 IO 不太协调（虽然它也支持建立二进制流）

IO：C++

```
my_stream << s << n << o << endl;
cout.precision(8); cout.width(12); // width只影响一次输出
cout << x << ' '  << y << endl;
```

通用格式设置函数 setf，可以写 cout.setf(...)

操控符方式：

```
cout << oct << n << flush << m ;
cout << setprecision(8) << scientific << ratio;
```

IO：Java

通过库提供 IO 功能

- 通过标准类库提供基于流概念的 IO 功能
- 提供两对基本的 IO 类：InputStream 和 OutputStream 完成字节输入和输出，Reader 和 Writer 完成字符输入和输出
- 提供了一组预定义功能非常丰富的流类，不仅作为用户交互、文件输入输出的接口，还可用于其他许多用途
  - 用字节数组、字符数组、字符串作为流的基础，使用各种 IO 操作
  - 过滤器流，支持流中的过滤操作
  - 管道流，支持流的串联
  - 格式化流
- 利用其他类型到字符串的转换和字符串运算，实现从各个类型到字符串形式的转换，进而利用字符串的 IO 功能实现其他类型的 IO

总结

- 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