Principles of Programming Languages

Control

Controlling Program Flows

- Computations in imperative-language programs
  - Evaluating expressions – reading variables, executing operations
  - Assigning resulting values to variables
  - Selecting among alternative control flow paths
  - Causing repeated execution
- A control structure is a control statement and the statements whose execution it controls
- Most programming languages follow a single thread of control (or scheduling)

Overview

- Selection Statements (Sec. 8.2)
- Iterative Statements (Sec. 8.3)
- Unconditional Branching (Sec. 8.4)
- Guarded Commands (Sec. 8.5)

Selection Statements

- A selection statement chooses between two or more paths of execution
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Two-Way Selection Statements

- General form:

```
if control_expression
then clause
else clause
```

- Control expression:
  - In C89, C99, Python, and C++, the control expression can be arithmetic
  - In languages such as Ada, Java, Ruby, and C#, the control expression must be Boolean

Then and Else Clauses

- In contemporary languages, **`then`** and **`else`** clauses can be single or compound statements
  - In Perl, all clauses must be delimited by braces (they must be compound even if there is only 1 statement)
  - Python uses indentation to define clauses

```
if x > y :
    x = y
    print "case 1"
```

Nesting Selectors

- Consider the following Java code:

```
if (sum == 0)
    if (count == 0)
        result = 0;
else result = 1;
```

- Which `if` gets the `else`? (dangling else)
- Java's static semantics rule: `else` matches with the nearest `if`

Nesting Selectors (cont.)

- To force an alternative semantics, compound statements may be used:

```
if (sum == 0) {
    if (count == 0)
        result = 0;
}
   else result = 1;
```

- The above solution is used in C, C++, and C#
- Perl requires that all **`then`** and **`else`** clauses to be compound and avoid the above problem

Nesting Selectors (cont.)

- The problem can also be solved by alternative means of forming compound statements, e.g., using a special word **end** in Ruby

```
if sum == 0 then
   if count == 0 then
     result = 0
   else
     result = 1
   end
end
```

```
if sum == 0 then
   if count == 0 then
     result = 0
   end
else
   result = 1
end
```

Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups
- Switch in C, C++, Java:

```
switch (expression) {
    case const_expr_1: stmt_1;
    …
    case const_expr_n: stmt_n;
    [default: stmt_n+1]
  }
```

Switch in C, C++, Java

- Design choices for C's **`switch`** statement
  - Control expression can be only an integer type
  - Selectable segments can be statement sequences, blocks, or compound statements
  - Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments); **`break`** is used for exiting **`switch`** → reliability of missing **`break`**
  - **`default`** clause is for unrepresented values (if there is no **`default`**, the whole statement does nothing)

Switch in C, C++, Java

```
switch (x)
  default:
    if (prime(x))
     case 2: case 3: case 5: case 7:
        process_prime(x);
    else
      case 4: case 6: case 8:
      case 9: case 10:
        process_composite(x);
```

Multiple-Way Selection in C#

- It has a static semantics rule that disallows the implicit execution of more than one segment
  - Each selectable segment must end with an unconditional branch (**`goto`** or **`break`**)
- The control expression and the case constants can be strings

```
switch (value) {
  case -1:  Negatives++;  break;
  case 0: Zeros++;    goto case 1;
  case 1: Positives++;  break;
  default: Console.WriteLine(“!!!\n”); }
```

Multiple-Way Selection in Ada

- Ada

```
case expression is
when choice list => stmt_sequence;
...
when choice list => stmt_sequence;
when others => stmt_sequence;]
end case;
```

- More reliable than C's **`switch`**
  - Once a **`stmt_sequence`** execution is completed, control is passed to the first statement after the **`case`** statement

Multiple-Way Selection Using **`if`**

- Multiple selectors can appear as direct extensions to two-way selectors, using **`else-if`** clauses, for example in Python:

```
if count < 10 :
  bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

More readable than deeply nested two-way selectors!

Can compare ranges

Overview

- Selection Statements (Sec. 8.2)
- Iterative Statements (Sec. 8.3)
- Unconditional Branching (Sec. 8.4)
- Guarded Commands (Sec. 8.5)

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- Counter-controlled loops:
  - A counting iterative statement has a loop variable, and a means of specifying the loop parameters: initial, terminal, stepsize values
  - Design Issues:
    - What are the type and scope of the loop variable?
    - Should it be legal for the loop variable or loop parameters to be changed in the loop body?

Iterative Statements: C-based

```
for ([expr_1] ; [expr_2] ; [expr_3])
  statement
```

- The expressions can be whole statements or statement sequences, separated by commas
  - The value of a multiple-statement expression is the value of the last statement in the expression
  - If second expression is absent, it is an infinite loop
- Design choices:
  - No explicit loop variable → the loop needs not count
  - Everything can be changed in the loop
  - 1st expr evaluated once, others with each iteration

Iterative Statements: C-based

```
for (count1 = 0, count2 = 1.0;
     count1 <= 10 && count2 <= 100.0;
     sum = ++count1 + count2, count2 *= 2);
```

- C++ differs from earlier C in two ways:
  - The control expression can also be Boolean
  - Initial expression can include variable definitions (scope is from the definition to the end of loop body)
- Java and C#
  - Differs from C++ in that the control expression must be Boolean

Logically-Controlled Loops

- Repetition control based on Boolean expression
- C and C++ have both pretest and posttest forms, and control expression can be arithmetic:

```
while (ctrl_expr)    do
loop body              loop body
                     while (ctrl_expr)
```

- Java is like C, except control expression must be Boolean (and the body can only be entered at the beginning -- Java has no goto)

User-Located Loop Control

- Programmers decide a location for loop control (other than top or bottom of the loop)
- Simple design for single loops (e.g., **`break`**)
- C , C++, Python, Ruby, C# have unconditional unlabeled exits (**`break`**), and an unlabeled control statement, **`continue`**, that skips the remainder of current iteration, but not the loop
- Java and Perl have unconditional labeled exits (**`break`** in Java, **`last`** in Perl) and labeled versions of **`continue`**

User-Located Loop Control

- In Java:

```
outerLoop:
  for (row = 0; row < numRows; row++)
    for (col = 0; col < numCols; col++)
    { sum += mat[row][col];
      if (sum > 1000.0)
        break outerLoop;
    }
```

Iteration Based on Data Structures

- Number of elements in a data structure control loop iteration
- Control mechanism is a call to an iterator function that returns the next element in the data structure in some chosen order, if there is one; else loop is terminated
- C's **`for`** statement can be used to build a user-defined iterator:

```
for(p=root; p==NULL; traverse(p)){}
```

Iteration Based on Data Structures

- PHP:

```
reset $list;
print("1st: "+current($list) + "<br />");
while($current_value = next($list))
  print("next: "+$current_value+"<br />");
```

- Java 5.0 (uses **`for`**, although called foreach)
  - For arrays and any other class that implements Iterable interface, e.g., ArrayList

```
for (String myElement : myList) { … }
```

Overview

- Selection Statements (Sec. 8.2)
- Iterative Statements (Sec. 8.3)
- Unconditional Branching (Sec. 8.4)
- Guarded Commands (Sec. 8.5)

Unconditional Branching

- Transfers execution control to a specified place in the program, e.g., **`goto`**
- Major concern: readability
  - Some languages do not support **`goto`** statement (e.g., Java)
  - C# offers **`goto`** statement (can be used in **`switch`** statements)
- Loop exit statements are restricted and somewhat hide away goto's

Overview

- Selection Statements (Sec. 8.2)
- Iterative Statements (Sec. 8.3)
- Unconditional Branching (Sec. 8.4)
- Guarded Commands (Sec. 8.5)

Guarded Commands

- Designed by Dijkstra
- Purpose: to support a new programming methodology that supports verification (correctness) during development
- Basis for two linguistic mechanisms for concurrent programming (in CSP and Ada)
- Basic Idea: if the order of evaluation is not important, the program should not specify one

Selection Guarded Command

- Form

```
if <Boolean exp> -> <statement>
[] <Boolean exp> -> <statement>
  ...
[] <Boolean exp> -> <statement>
fi
```

- Semantics:
  - Evaluate all Boolean expressions
  - If > 1 are true, choose one non-deterministically
  - If none are true, it is a runtime error
  - Prog correctness cannot depend on statement chosen

Selection Guarded Command

```
if x >= y -> max := x
[] y >= x => max := y
fi
```

Compare with the following code:

```
if (x >= y)
  max = x;
else
  max = y;
```

Loop Guarded Command

- Form

```
do <Boolean> -> <statement>
 [] <Boolean> -> <statement>
  ...
 [] <Boolean> -> <statement>
od
```

- Semantics: for each iteration
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically; then start loop again
  - If none are true, exit loop

访问内存中的数据

访问数据

- 为了得到内存中的变量要做什么?
    - 取决于位置，取决于存储的类型（静态、自动、动态）

```
int siA;
void static_auto_local() {
        int aiB;
        static int siC=3;
        int * apD;
        int aiE=4, aiF=5, aiG=6;

        siA = 2;
        aiB = siC + siA;
        apD = & aiB;
        (*apD)++;
        apD = &siC;
        (*apD) += 9;
        apD = &siA;
        apD = &aiE;
        apD = &aiF;
        apD = &aiG;
        (*apD)++;
        aiE+=7;
        *apD = aiE + aiF;
}
```

静态变量

- 静态变量可以安排在32位内存空间的任何地方，所以需要一个32位的指针
- 在16位指令中放不下32位的指针，32位指令也不行，所以指针和指令是分开的，但是就在附近，这样可以用短小的PC相关的偏移量来访问到
- 把这个指针装入寄存器（r0）
- 这样就可以用r0中的指针把那个变量的值从内存中装入寄存器（r1）
- 类似的方式可以用来把一个新的值写入内存中的变量

Load r0 with pointer to variable
Load r1 from [r0]
Use value of variable

Label:
32-bit pointer to Variable

Variable

静态变量

- 关键
  - 变量的值
  - 变量的地址
  - 变量的地址的拷贝的地址
- 代码
  - 把siA的地址（位于|L1.240|）装入r2
  - （通过指针r2，偏移量为0）把siA的内容装入r1
  - 同样的方式得到地址在|L1.244|的siC
- siA和siC的地址作为要装入指针的字面量保存着
- 变量siC和siA带着初始值位于.data区

```
AREA ||.text||, CODE, READONLY, ALIGN=2
;;;20          siA = 2;
00000e  2102  MOVS    r1,#2
000010  4a37  LDR     r2,|L1.240|
000012  6011  STR     r1,[r2,#0]  ; siA
;;;21      aiB = siC + siA;
000014  4937  LDR     r1,|L1.244|
000016  6809  LDR     r1,[r1,#0]  ; siC
000018  6812  LDR     r2,[r2,#0]  ; siA
00001a  1889  ADDS    r1,r1,r2
...

|L1.240|
                    DCD          ||siA||
|L1.244|
                    DCD          ||siC||
        AREA ||.data||, DATA, ALIGN=2
||siC||
                    DCD     0x00000003
||siA||
                    DCD     0x00000000
```

堆栈中的自动变量

- 自动变量保存在函数的活动记录中（除非被优化、提升到寄存器中了）
- 活动记录位于堆栈中
- 调用一次函数就在堆栈中分配空间、创建一个活动记录
- 从函数中返回时删除活动记录，释放堆栈中的空间

```
int main(void) {
        auto vars
        a();
}

void a(void) {
        auto vars
        b();
}

void b(void) {
        auto vars
        c();
}

void c(void) {
        auto vars
        …
}
```

自动变量

```
int main(void) {
        auto vars
        a();
}

void a(void) {
        auto vars
        b();
}

void b(void) {
        auto vars
        c();
}

void c(void) {
        auto vars
        …
}
```

较低地址

	(空闲堆栈空间)	
当前函数C的活动记录	本地存储	<- 执行C时的堆栈指针
	保存了的寄存器	
	参数 （可选）	
调用者函数B的活动记录	本地存储	<-执行B时的堆栈指针
	保存了的寄存器	
	参数 （可选）	
调用者的调用者函数A的活动记录	本地存储	<-执行A时的堆栈指针
	保存了的寄存器	
	参数 （可选）	
调用者的调用者的调用者函数main的活动记录	本地存储	<-执行main时的堆栈指针
	保存了的寄存器	
	参数 （可选）	

较高地址

自动变量的寻址

- 程序必须在堆栈中为变量分配空间
- 堆栈寻址用的是从堆栈指针开始的偏移量： [sp, #offset]
    - 指令中有一个字节用作偏移量，是需要乘以四的
    - 可能的偏移量是：0, 4, 8, …, 1020
    - 这样最大的地址范围就是1024字节

地址	内容
SP	
SP+4	
SP+8	
SP+0xC	
SP+0x10	
SP+0x14	
SP+0x18	
SP+0x1C	
SP+0x20	

自动变量

地址	内容
SP	aiG
SP+4	aiF
SP+8	aiE
SP+0xC	aiB
SP+0x10	r0
SP+0x14	r1
SP+0x18	r2
SP+0x1C	r3
SP+0x20	lr

- 初始化aiE
- 初始化aiF
- 初始化aiG
- 保存aiB的值

```
;;;14     void static_auto_local( void )
{
000000  b50f          PUSH   {r0-r3,lr}
;;;15  int aiB;
;;;16  static int siC=3;
;;;17  int * apD;
;;;18  int aiE=4, aiF=5, aiG=6;
000002  2104 MOVS            r1,#4
000004  9102          STR      r1,
[sp,#8]
000006  2105          MOVS   r1,#5
000008  9101          STR      r1,
[sp,#4]
00000a  2106          MOVS   r1,#6
00000c  9100          STR      r1,
[sp,#0]
…
;;;21       aiB = siC + siA;
…
00001c  9103          STR      r1,
[sp,#0xc]
```

使用指针

指向自动变量的指针

- C的指针：保存数据地址的变量
- aiB在堆栈中，位于 SP+0xc
- 用堆栈指针和偏移量（0xc)计算变量的地址赋给r0
- 从内存中把变量的值装入到r1
- 对r1对运算，存回变量的地址

```
;;;22                       apD = & aiB;
00001e  a803                ADD
r0,sp,#0xc
;;;23                       (*apD)++;
000020  6801                LDR     r1,
[r0,#0]
000022  1c49                ADDS    r1,r1,#1
000024  6001                STR     r1,
[r0,#0]
```

指向静态变量的指针

- 从变量地址的名字那里把变量的地址装入 r0
- 从内存中把变量的值装入 r1
- 对 r1做运算，存回变量的地址

;;;24 apD = &siC;
000026 4833 LDR r0,|L1.244|
;;;25 (*apD) += 9;
000028 6801 LDR r1,
[r0,#0]
00002a 3109 ADDS r1,r1,#9
00002c 6001 STR r1,
[r0,#0]
|L1.244|
DCD ||
siC||
AREA ||.data||, DATA,
ALIGN=2
||siC||
DCD
0x00000003

访问数组

访问数组

- 怎样能得到内存中的数组单元?
  - 取决于多少个维度
  - 取决于单元的大小和每行的宽度
  - 取决于位置，也就是存储的类型（静态、自动、动态）

```
unsigned char buff2[3];
unsigned short int buff3[5][7];

unsigned int arrays(unsigned char
n, unsigned char j) {
  volatile unsigned int i;

  i = buff2[0] + buff2[n];
  i += buff3[n][j];
                                return i;
}
```

访问一维数组单元

- 需要计算单元的地址——以下两项的和：
  - 数组开头的地址
  - 偏移量：索引 * 单元大小
- Buff2是无符号字符的数组

- 把n（参数）从r0移入r2
- 把buff2的指针装入r3
- 把buff2的第一个单元（字节）装入r3
- 把buff2的指针装入r4
- 把buff2+r2地址上的单元（字节）装入r4
  - r2里是参数n
- 把r3和r4加起来

地址	内容
buff2	buff2[0]
buff2 + 1	buff2[1]
buff2 + 2	buff2[2]

```
00009e  4602        MOV    r2,r0
;;;76    i = buff2[0] + buff2[n];
0000a0  4b1b        LDR    r3,l
L1.272l
0000a2  781b        LDRB   r3,
[r3,#0]  ; buff2
0000a4  4c1a        LDR    r4,l
L1.272l
0000a6  5ca4        LDRB   r4,
[r4,r2]
0000a8  1918        ADDS   r0,r3,r4
lL1.272l
                    DCD    buff2
```

访问二维数组

short int buff3[5][7]

地址	内容
buff3	buff3[0][0]
buff3+1	
buff3+2	buff3[0][1]
buff3+3	
(等……)	
buff3+10	buff3[0][5]
buff3+11	
buff3+12	buff3[0][6]
buff3+13	
buff3+14	buff3[1][0]
buff3+15	
buff3+16	buff3[1][1]
buff3+17	
buff3+18	buff3[1][2]
buff3+19	
(等……)	
buff3+68	buff3[4][6]
buff3+69	

- var[行][列]
- 大小
    - 单元：2字节
    - 行：7*2字节 = 14 字节（0xe）
- 基于行和列的下标计算偏移量
    - 列偏移量 = 列下标 * 单元大小
    - 行偏移量 = 行下标 * 行大小

访问二维数组的代码

- 把行大小装入r3
- 乘以行下标（r2中的n），得到行偏移量装入r3
- 把buff3的地址装入r4
- 把buff3的地址加到r3的行偏移量上
- 列下表（r1中的j）左移一位，就是成一2（每个单元2个字节）
- 把r3+r4地址（buff3+行偏移量+列偏移量）上的单元（半字）装入r3
- 把r3加到变量i上（r0）

```
;;;77        i += buff3[n][j];
0000aa  230e        MOVS   r3,#0xe
0000ac  4353        MULS   r3,r2,r3

0000ae  4c19        LDR    r4,l
L1.276l

0000b0  191b        ADDS   r3,r3,r4

0000b2  004c        LSLS   r4,r1,#1

0000b4  5b1b        LDRH   r3,
[r3,r4]

0000b6  1818        ADDS   r0,r3,r0

lL1.276l
                    DCD    buff3
```

控制流

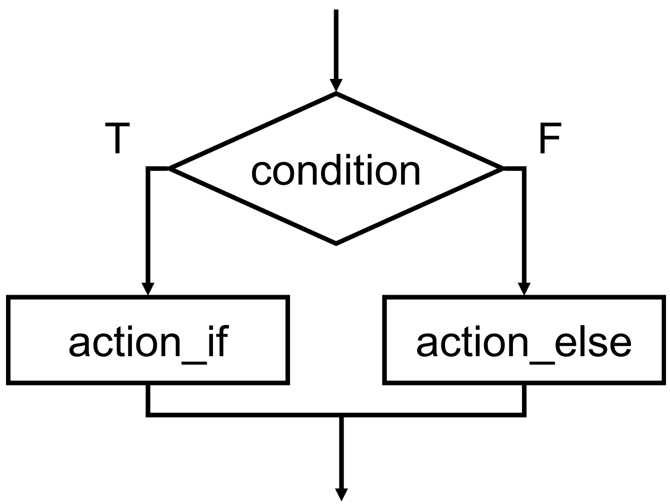
控制流：条件和循环

- 编译器如何实现条件判断和循环?

```
if (x){
    y++;
} else {
    y--;
}

switch (x) {
  case 1:
    y += 3;
    break;
  case 31:
    y -= 5;
    break;
  default:
    y--;
    break;
  }
```

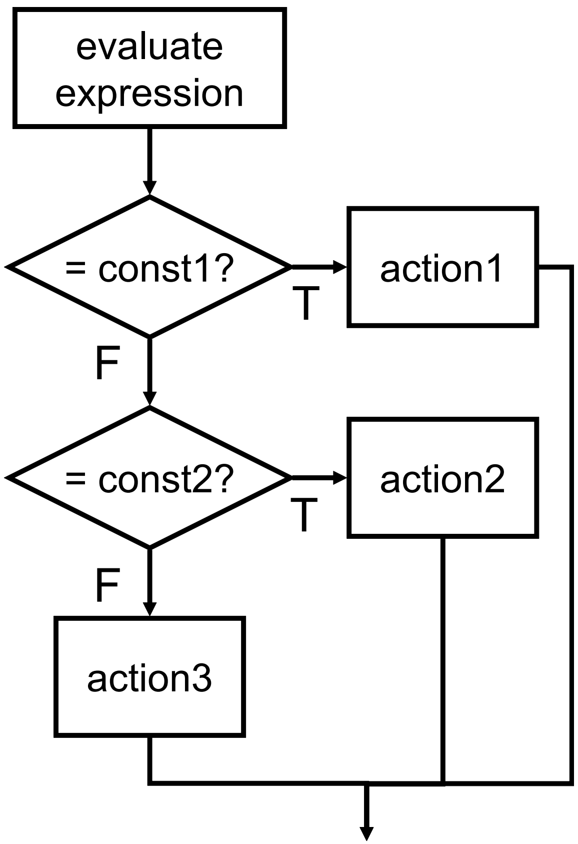
```
while (x<10) {
    x = x + 1;
}

for (i = 0; i < 10; i++){
    x += i;
}

do {
    x += 2;
} while (x < 20);
```

控制流： if/else



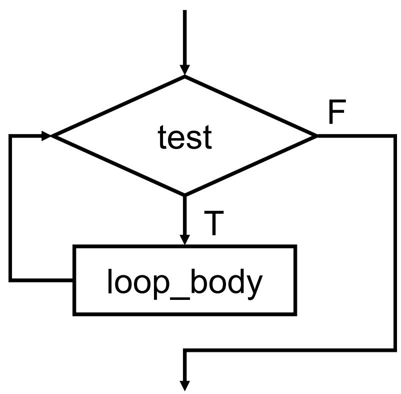

```
;;;39       if (x){
000056  2900        CMP     r1,#0
000058  d001        BEQ     l
L1.94l
;;;40          y++;
00005a  1c52        ADDS
r2,r2,#1
00005c  e000        B       l
L1.96l

   lL1.94l
;;;41       } else {
;;;42         y--;
00005e  1e52        SUBS
r2,r2,#1

       lL1.96l
;;;43       }
```

控制流：switch



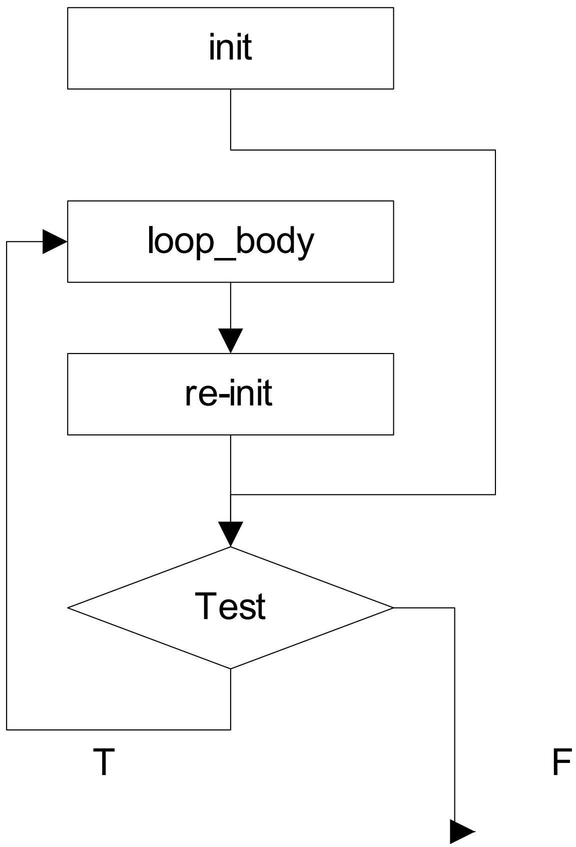

```
;;;45        switch (x) {
000060  2901          CMP      r1,#1
000062  d002          BEQ      |L1.106|
000064  291f          CMP      r1,#0x1f
000066  d104          BNE      |L1.114|
000068  e001          B        |L1.110|
        |L1.106|
;;;46        case 1:
;;;47          y += 3;
00006a  1cd2          ADDS     r2,r2,#3
;;;48          break;
00006c  e003          B        |L1.118|
|L1.110|
;;;49        case 31:
;;;50          y -= 5;
00006e  1f52          SUBS     r2,r2,#5
;;;51          break;
000070  e001          B        |L1.118|
|L1.114|
;;;52        default:
;;;53          y--;
000072  1e52          SUBS     r2,r2,#1
;;;54          break;
000074  bf00          NOP
|L1.118|
000076  bf00          NOP
;;;55        }
```

循环：while



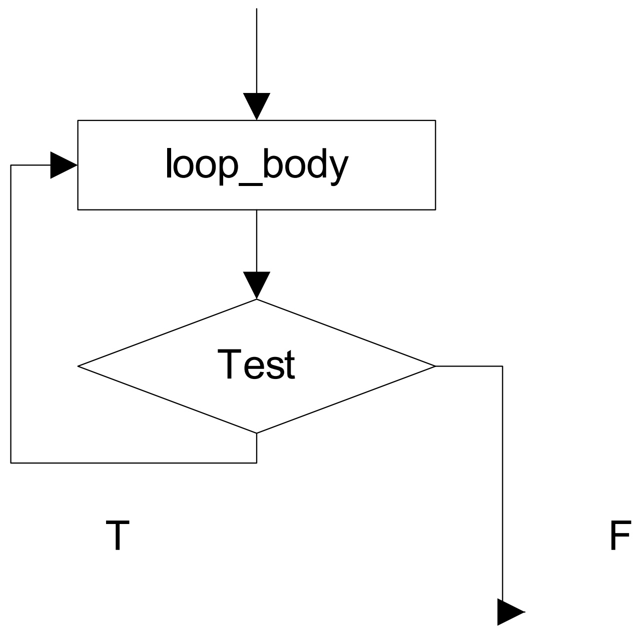

```
;;;57        while (x<10) {
000078  e000         B           |
L1.124|
                     |L1.122|
;;;58         x = x + 1;
00007a  1c49         ADDS
r1,r1,#1
                     |L1.124|
00007c  290a         CMP
r1,#0xa          ;57
00007e  d3fc         BCC     |
L1.122|
;;;59        }
```

循环：for




```
;;;61       for (i = 0; i < 10; i++){
000080  2300          MOVS  r3,#0
000082  e001          B     |
L1.136|

                      |L1.132|
;;;62        x += i;
000084  18c9          ADDS
r1,r1,r3
000086  1c5b          ADDS
r3,r3,#1              ;61

                      |L1.136|
000088  2b0a          CMP
r3,#0xa
;61
00008a  d3fb          BCC   |
L1.132|
;;;63       }
```

循环：do/while



;;;65 do {
00008c bf00 NOP
|L1.142|
;;;66 x += 2;
00008e 1c89 ADDS
r1,r1,#2
;;;67 } while (x < 20);
000090 2914 CMP
r1,#0x14
000092 d3fc BCC |
L1.142|

Summary

- Variety of statement-level structures
- Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability
- Functional and logic programming languages are quite different control structures