# Control Flow

Stephen A. Edwards

Columbia University

Fall 2012

# Control Flow

"Time is Nature's way of preventing everything from happening at once."

Scott identifies seven manifestations of this:

| | |
|---|---|
| 1. Sequencing | foo(); bar(); |
| 2. Selection | if (a) foo(); |
| 3. Iteration | while (i<10) foo(i); |
| 4. Procedures | foo(10,20); |
| 5. Recursion | foo(int i) { foo(i-1); } |
| 6. Concurrency | foo() \|\| bar() |
| 7. Nondeterminism | do a -> foo(); [] b -> bar(); |

# Ordering Within Expressions

What code does a compiler generate for

```
a = b + c + d;
```

Most likely something like

```
tmp = b + c;
a = tmp + d;
```

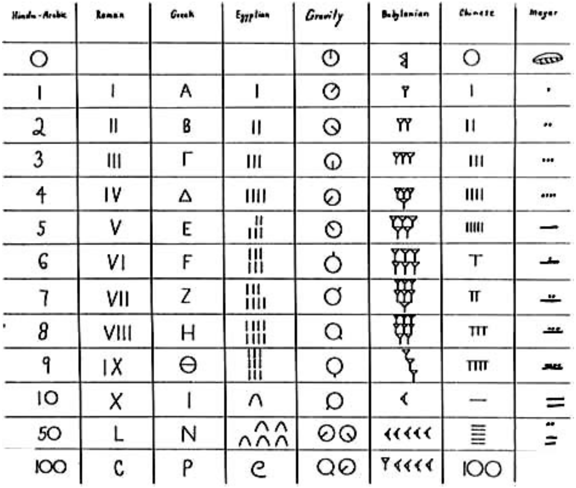
(Assumes left-to-right evaluation of expressions.)

# Order of Evaluation

Why would you care?

Expression evaluation can have side-effects.

Floating-point numbers don't behave like numbers.



# Side-effects

```
int x = 0;

int foo() {
  x += 5;
  return x;
}

int bar() {
  int a = foo() + x + foo();
  return a;
}
```

What does *bar()* return?

# Side-effects

```
int x = 0;

int foo() {
  x += 5;
  return x;
}

int bar() {
  int a = foo() + x + foo();
  return a;
}
```

What does *bar()* return?

GCC returned 25.

Sun's C compiler returned 20.

C says expression evaluation order is implementation-dependent.

# Side-effects

Java prescribes left-to-right evaluation.

```
class Foo {

  static int x;

  static int foo() {
     x += 5;
     return x;
  }

  public static void main(String args[]) {
    int a = foo() + x + foo();
    System.out.println(a);
  }

}
```

Always prints 20.

# Number Behavior

Basic number axioms:

| | | | |
|---|---|---|---|
| $a + x$ | $=$ | $a$ if and only if $x = 0$ | Additive identity |
| $(a + b) + c$ | $=$ | $a + (b + c)$ | Associative |
| $a(b + c)$ | $=$ | $ab + ac$ | Distributive |

# Misbehaving Floating-Point Numbers

`1e20 + 1e-20 = 1e20`

`1e-20` $\ll$ `1e20`

`(1 + 9e-7) + 9e-7` $\neq$ `1 + (9e-7 + 9e-7)`

`9e-7` $\ll$ `1`, so it is discarded, however, `1.8e-6` is large enough

$1.00001(1.000001 - 1) \neq 1.00001 \cdot 1.000001 - 1.00001 \cdot 1$

$1.00001 \cdot 1.000001 = 1.00001100001$ requires too much intermediate precision.

# What's Going On?

Floating-point numbers are represented using an exponent/significand format:

$$\begin{aligned} &1 \quad \underbrace{10000001}_{\text{8-bit exponent}} \quad \underbrace{01100000000000000000000}_{\text{23-bit significand}} \\ &= \quad -1.011_2 \times 2^{129-127} = -1.375 \times 4 = -5.5. \end{aligned}$$

What to remember:

$$\underbrace{\color{red}{1363.4568}}_{\text{represented}}\underbrace{46353963456293}_{\text{rounded}}$$

# What's Going On?

Results are often rounded:

$$\begin{array}{r} 1.000010\,00000 \\ \times 1.000001\,00000 \\ \hline 1.000011\,\underbrace{00001}_{\text{rounded}} \end{array}$$

When $b \approx -c$, $b + c$ is small, so $ab + ac \neq a(b + c)$ because precision is lost when $ab$ is calculated.

Moral: Be aware of floating-point number properties when writing complex expressions.

# Short-Circuit Evaluation

When you write

```
if (disaster_could_happen)
   avoid_it();
else
   cause_a_disaster();
```

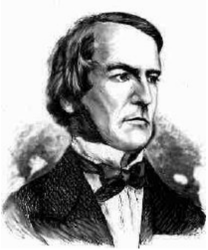
`cause_a_disaster()` is not called when `disaster_could_happen` is true.

The *if* statement evaluates its bodies lazily: only when necessary.

The section operator `? :` does this, too.

```
cost = disaster_possible ? avoid_it() : cause_it();
```

# Logical Operators



In Java and C, Boolean logical operators "short-circuit" to provide this facility:

```
if (disaster_possible || case_it()) { ... }
```

`cause_it()` only called if `disaster_possible` is false.

The && operator does the same thing.

Useful when a later test could cause an error:

```
int a[10];

if (i => 0 && i < 10 && a[i] == 0) { ... }
```

# Unstructured Control-Flow

Assembly languages usually provide three types of instructions:

Pass control to next instruction:

`add, sub, mov, cmp`

Pass control to another instruction:

`jmp rts`

Conditionally pass control next or elsewhere:

`beq bne blt`

# Unstructured Control-Flow

```
  BEQ A
B:

  JMP C
A:

  BEQ D
C:

  BEQ B
D:

  BNE B
  RTS
```

# Structured Control-Flow

The “object-oriented languages” of the 1960s and 70s.

Structured programming replaces the evil *goto* with structured (nested) constructs such as

for

while

break

return

continue

do .. while

if .. then .. else

# Gotos vs. Structured Programming

A typical use of a goto is building a loop. In BASIC:

```
10 PRINT I
20 I = I + 1
30 IF I < 10 GOTO 10
```

A cleaner version in C using structured control flow:

```
do {
  printf("%d\n", i);
  i = i + 1;
} while ( i < 10 )
```

An even better version

```
for (i = 0 ; i < 10 ; i++)
  printf("%d\n", i);
```

# Gotos vs. Structured Programming

Break and continue leave loops prematurely:

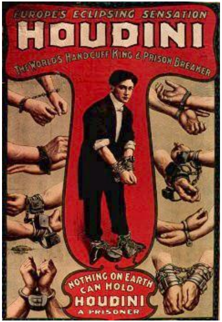
```
for ( i = 0 ; i < 10 ; i++ ) {
  if ( i == 5 ) continue;
  if ( i == 8 ) break;
  printf("%d\n", i);
}
```

```
  i = 0;
Again:
  if (!(i < 10)) goto Break;
  if ( i == 5 ) goto Continue;
  if ( i == 8 ) goto Break;
  printf("%d\n", i);
Continue: i++; goto Again;
Break:
```

# Escaping from Loops

Java allows you to escape from labeled loops:

```
a: for (int i = 0 ; i < 10 ; i++)
     for ( int j = 0 ; j < 10 ; j++) {
        System.out.println(i + "," + j);
        if (i == 2 && j == 8) continue a;
        if (i == 8 && j == 4) break a;
     }
```



# Gotos vs. Structured Programming

Pascal has no "return" statement for escaping from functions/procedures early, so goto was necessary:

```
procedure consume_line(var line : string);
begin
  if line[i] = '%' then goto 100;
  (* .... *)
100:
end
```

In C and many others, return does this for you:

```
void consume_line(char *line) {
  if (line[0] == '%') return;
}
```

# Loops

A modern processor can execute something like 1 billion instructions/second.

How many instructions are there in a typical program? Perhaps a million.

Why do programs take more than 1ms to run?

Answer: loops

This insight is critical for optimization: only bother optimizing the loops since everything else is of vanishing importance.

# Enumeration-Controlled Loops in FORTRAN

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

Executes body of the loop with i=1, 3, 5, . . . , 9

Tricky things:

What happens if the body changes the value of i?

What happens if gotos jump into or out of the loop?

What is the value of i upon exit?

What happens if the upper bound is less than the lower one?

# Changing Loop Indices

Most languages prohibit changing the index within a loop.

(Algol 68, Pascal, Ada, FORTRAN 77 and 90, Modula-3)

But C, C++, and Java allow it.

Why would a language bother to restrict this?

# Empty Bounds

In FORTRAN, the body of this loop is executed once:

```
        do 10 i = 10, 1, 1
           ...
10:     continue
```

"for i = 10 to 1 by 1"

Test is done *after* the body.

Modern languages place the test *before* the loop.

Does the right thing when the bounds are empty.

Slightly less efficient (one extra test).

# Scope of Loop Index

What happens to the loop index when the loop terminates?

Index is undefined: FORTRAN IV, Pascal.

Index is its last value: FORTRAN 77, Algol 60

Index is just a variable: C, C++, Java

Tricky when iterating over subranges. What's next?

```
var c : 'a'..'z';
for c := 'a' to 'z' do begin
  ...
end;  (* what's c? *)
```

# Scope of Loop Index

Originally in C++, a locally-defined index variable's scope extended beyond the loop:

```
for (int i = 0 ; i < 10 ; i++) { ... }
a = a + i;    // Was OK: i = 10 here
```

But this is awkward:

```
for (int i = 0 ; i < 10 ; i++) { ... }
...
for (int i = 0 ; i < 10 ; i++) // Error: i redeclared
```

# Scope of Loop Index

C++ and Java now restrict the scope to the loop body:

```
for (int i = 0 ; i < 10 ; i++ ) {
  int a = i; // OK
}
...
int b = i; // Error: i undefined
...
for (int i = 0 ; i < 10 ; i++ ) { // OK
}
```

Rather annoying: broke many old C++ programs.

Better for new code.

# Algol's Combination Loop

*for* → `for` *id* `:=` *for-list* `do` *stmt*

*for-list* → *enumerator* ( `,` *enumerator* )*

*enumerator* → *expr*
→ *expr* `step` *expr* `until` *expr*
→ *expr* `while` *condition*

Equivalent:

```
for i := 1, 3, 5, 7, 9 do ...
for i := 1 step 2 until 10 do ...
for i := 1, i+2 while i < 10 do ...
```

Language implicitly steps through enumerators (implicit variable).

# Mid-test Loops

```
while true do begin
  readln(line);
  if all_blanks(line) then goto 100;
  consume_line(line);
end;
100:
```

In Modula-2:

```
LOOP
  line := ReadLine;
WHEN AllBlanks(line) EXIT;
  ConsumeLine(line)
END;
```

# Multi-way Branching

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

```
switch (s) {
  case 1: goto One;
  case 2: goto Two;
  case 3: goto Three;
  case 4: goto Four;
}
goto Break;

One:   one(); goto Break;
Two:   two(); goto Break;
Three: three(); goto Break;
Four:  four(); goto Break;
Break:
```

Switch sends control to one of the case labels. Break terminates the statement. Really just a multi-way *goto*:

# Implementing multi-way branches

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

Obvious way:

```
if (s == 1) { one(); }
else if (s == 2) { two(); }
else if (s == 3) { three(); }
else if (s == 4) { four(); }
```

Reasonable, but we can sometimes do better.

# Implementing multi-way branches

If the cases are *dense*, a branch table is more efficient:

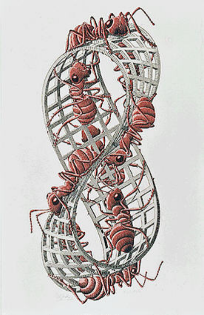
```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

A branch table written using a GCC extension:

```
  /* Array of addresses of labels */
  static void *l[] = { &&L1, &&L2, &&L3, &&L4 };

  if (s >= 1 && s <= 4)
     goto *l[s-1];
  goto Break;
L1: one(); goto Break;
L2: two(); goto Break;
L3: three(); goto Break;
L4: four(); goto Break;
Break:
```

# Recursion and Iteration

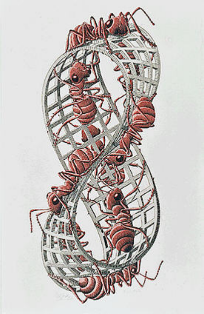


To compute $\sum_{i=0}^{10} f(i)$ in C,

the most obvious technique is iteration:

```
double total = 0;
for ( i = 0 ; i <= 10 ; i++ )
  total += f(i);
```

# Recursion and Iteration



To compute $\sum_{i=0}^{10} f(i)$ in C,

the most obvious technique is iteration:

```
double total = 0;
for ( i = 0 ; i <= 10 ; i++ )
  total += f(i);
```

But this can also be defined recursively

```
double sum(int i, double acc)
{
  if (i <= 10)
    return sum(i+1, acc + f(i));
  else
    return acc;
}

sum(0, 0.0);
```

# Tail-Recursion and Iteration

```
int gcd(int a, int b) {
  if ( a==b ) return a;
  else if ( a > b ) return gcd(a-b,b);
  else return gcd(a,b-a);
}
```

Notice: no computation follows any recursive calls.

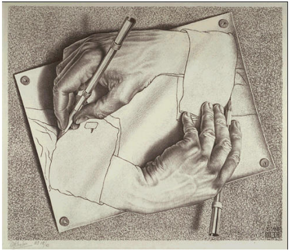
Stack is not necessary: all variables “dead” after the call.

Local variable space can be reused. Trivial since the collection of variables is the same.

Works in O’Caml, too

```
let rec gcd a b =
  if a = b then a
  else if a > b then gcd (a - b) b
  else gcd a (b - a)
```

# Tail-Recursion and Iteration



```
int gcd(int a, int b) {
  if ( a==b ) return a;
  else if ( a > b ) return gcd(a-b,b);
  else return gcd(a,b-a);
}
```

Can be rewritten into:

```
int gcd(int a, int b) {
start:
  if ( a==b ) return a;
  else if ( a > b ) a = a-b; goto start;
  else b = b-a; goto start;
}
```

Good compilers, especially those for functional languages, identify and optimize tail recursive functions.

Less common for imperative languages, but gcc -O was able to handle this example.

# Applicative- and Normal-Order Evaluation

```
int p(int i) {
   printf("%d ", i);
   return i;
}

void q(int a, int b, int c)
{
  int total = a;
  printf("%d ", b);
  total += c;
}

q( p(1), 2, p(3) );
```

What does this print?

# Applicative- and Normal-Order Evaluation

```
int p(int i) {
   printf("%d ", i);
   return i;
}

void q(int a, int b, int c)
{
  int total = a;
  printf("%d ", b);
  total += c;
}

q( p(1), 2, p(3) );
```

What does this print?

Applicative: arguments evaluated before function is called.

Result: 1 3 2

Normal: arguments evaluated when used.

Result: 1 2 3

# Applicative- vs. and Normal-Order

Most languages use applicative order.

Macro-like languages often use normal order.

```
#define p(x) (printf("%d ",x), x)

#define q(a,b,c) total = (a), \
   printf("%d ", (b)), \
   total += (c)

q( p(1), 2, p(3) );
```

Prints 1 2 3.

Some functional languages also use normal order evaluation to avoid doing work. “Lazy Evaluation”

# Argument Order Evaluation

C does not define argument evaluation order:

```
int p(int i) {
   printf("%d ", i);
   return i;
}

int q(int a, int b, int c) {}

q( p(1), p(2), p(3) );
```

Might print 1 2 3, 3 2 1, or something else.

This is an example of *nondeterminism*.

# Nondeterminism

Nondeterminism is not the same as random:

Compiler usually chooses an order when generating code.

Optimization, exact expressions, or run-time values may affect behavior.

Bottom line: don't know what code will do, but often know set of possibilities.

```
int p(int i) { printf("%d ", i); return i; }
int q(int a, int b, int c) {}
q( p(1), p(2), p(3) );
```

Will *not* print 5 6 7. It will print one of

1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1

# Nondeterminism

Nondeterminism lurks in most languages in one form or another.

Especially prevelant in concurrent languages.

Sometimes it's convenient, though:

```
if a >= b -> max := a
[] b >= a -> max := b
fi
```

Nondeterministic (irrelevant) choice when a=b.

Often want to avoid it, however.