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();

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 a = foo() + x + foo();
What’s the final value of a?

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:
\[ a + x = a \text{ if and only if } x = 0 \]  
Additive identity
\[ (a + b) + c = a + (b + c) \]  
Associative
\[ ab + ac = (a+b)c \]  
Distributive

Misbehaving Floating-Point Numbers
\[ 1.0e20 + 1.0e-20 = 1.0e20 \]
\[ 1.0e-20 \ll 1.0e20 \]
\[ (1 + 9e-7) + 9e-7 \neq 1 + (9e-7 + 9e-7) \]
9e-7 \ll 1, so it is discarded, however, 1.8e-6 is large enough
\[ 1.00001 (1.000001 - 1) \neq 1.00001 \cdot 1.00001 - 1.00001 \cdot 1 \]
\[ 1.00001 \cdot 1.000001 = 1.00001 \cdot 100001 \text{ requires too much intermediate precision.} \]
Floating-point numbers are represented using an exponent/significand format:

\[
\begin{array}{c|c}
\text{exponent} & \text{significand} \\
10000001 & 01100000000000000000000 \\
\end{array}
\]

8-bit exponent 23-bit significand

\[
-1.011_2 \times 2^{129-127} = -1.375 \times 4 = -5.5.
\]

What to remember:

\[
1363.4568 \quad 46353963456293
\]

represented rounded

Results are often rounded:

\[
\begin{array}{c|c}
\text{1.0000100000} & \text{1.0000100000} \\
\end{array}
\]

\[
= 1.00001 \times 2^{129-127} = -1.375 \times 4 = -5.5.
\]

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.

Logical Operators

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

\[
\begin{align*}
\text{if (disaster_possible || case_it())} & \quad \{ \ldots \} \\
\text{cause_it()} & \quad \text{only called if disaster_possible is false.}
\end{align*}
\]

The \&\& operator does the same thing.

Useful when a later test could cause an error:

\[
\begin{align*}
\text{int a[10];} \\
\text{if (i >= 0 && i < 10 && a[i] == 0) \{ \ldots \}}
\end{align*}
\]

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

Structured Control-Flow

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

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

- if-then-else
- for
- while
- do..while
- break
- continue
- return

Gotos vs. Structured Programming

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

\[
\begin{align*}
10 & \text{ print I} \\
20 & \text{ I = I + 1} \\
30 & \text{ IF I < 10 GOTO 10}
\end{align*}
\]

A cleaner version in C using structured control flow:

\[
\begin{align*}
& \text{do \{} \\
& \quad \text{printf("%d
", i);} \\
& \quad i = i + 1; \\
& \} \text{while (i < 10)}
\end{align*}
\]

An even better version

\[
\begin{align*}
& \text{for (i = 0; i < 10; i++) printf("%d
", i);} \\
\end{align*}
\]

Structured Control-Flow

So-called because it’s easy to create spaghetti:

\[
\begin{align*}
& \text{beq A} \\
& \text{B:} \\
& \text{jmp C} \\
& \text{A:} \\
& \quad \text{beq D} \\
& \text{C:} \\
& \quad \text{beq B} \\
& \text{D:} \\
& \quad \text{bne B}
\end{align*}
\]

Short-Circuit Evaluation

When you write

\[
\begin{align*}
& \text{if (disaster\_could\_happen) avoid\_it(); else cause\_a\_disaster();} \\
& \text{cause\_a\_disaster()} \text{is not called when disaster\_could\_happens true.}
\end{align*}
\]

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

The section operator ? : does this, too.

\[
\begin{align*}
& \text{cost =} \\
& \text{disaster\_possible \? avoid\_it() : cause\_it();}
\end{align*}
\]

Gotos vs. Structured Programming

Break and continue leave loops prematurely:

\[
\begin{align*}
& \text{for (i = 0; i < 10; i++) \{} \\
& \text{\quad if (i == 5) continue;} \\
& \text{\quad if (i == 8) break;} \\
& \text{\quad printf("%d
", i);} \\
& \text{\}} \\
& \text{Again: if (!(i < 10)) goto Break;}
\end{align*}
\]

Gotos vs. Structured Programming

Again: if (!(i < 10)) goto Break;

\[
\begin{align*}
& \text{if (i == 5) goto Continue;} \\
& \text{if (i == 8) goto Break;} \\
& \text{printf("%d
", i);} \\
& \text{Continue: i++; goto Again;} \\
& \text{Break:}
\end{align*}
\]
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 1 µs to run, then?

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

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

```
10: continue
```

```
for i = 10 to 1 by 1
```

Test is done after the body. Does the right thing when the bounds are empty. Slightly less efficient (one extra test).

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 1 µs to run, then?

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

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. Does the right thing when the bounds are empty. Slightly less efficient (one extra test).

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 1 µs to run, then?

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

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. Does the right thing when the bounds are empty. Slightly less efficient (one extra test).
### 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: 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 sends control to one of the case labels. Break terminates the statement.

### 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;
}
```

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.

### Recursion and Iteration

Consider computing

```
\sum_{i=0}^{10} f(i)
```

In C, the most obvious evaluation is iterative:

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

```
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.
Tail-Recursion and Iteration

```c
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;
}
```

Tail-Recursion and Iteration

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

Less common for imperative languages.

But gcc -O was able to rewrite the gcd example.

Applicative- and Normal-Order Evaluation

```c
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;
    printf("%d ", total);
}
q( p(1), 2, p(3) );
```

Applicative- and Normal-Order Evaluation

Most languages use applicative order.

Macro-like languages often use normal order.

```c
#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:

```c
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.

```c
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, 3 2 1
```

Nondeterminism

Nondeterminism lurks in most languages in one form or another.

Especially prevalent in concurrent languages.

Sometimes it’s convenient, though:

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

Nondeterministic (irrelevant) choice when a=b.

Often want to avoid it, however.