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() jj
    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 a = foo() + x + foo();

What’s the final value of a?

Side-effects

int x = 0;

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

int a = foo() + x + foo();

GCC sets a=25.
Sun’s C compiler gave a=20.
C says expression evaluation order is implementation-dependent.

Basic number axioms:

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

Misbehaving Floating-Point Numbers

1e20 + 1e-20 = 1e20
1e-20 ≪ 1e20

(1 + 9e-7) + 9e-7 ≠ 1 + (9e-7 + 9e-7)
9e-7 ≪ 1, so it is discarded, however, 1.8e-6 is large enough

1.00001(1.000001 - 1) ≠ 1.00001 · 1.000001 - 1.00001 · 1
1.00001 · 1.000001 = 1.00001100001 requires too much intermediate precision.
What's Going On?

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

```
1 10000001 01100000000000000000000
8-bit exponent 23-bit significand
```

```
= -1.0112 \times 2^{129-127} = -1.375 \times 4 = -5.5.
```

What to remember:

```
1363.4568 | 4635963456293
```

represented rounded

What's Going On?

Results are often rounded:

```
1.00001000000
\times 1.00000100000001
= 1.000011000001
```

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.

Logical Operators

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

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

`case_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) { ... }
```

Short-Circuit Operators

Not all languages provide short-circuit operators. Pascal does not.

C and Java have two sets:

- Logical operators `||` `&&` short-circuit.
- Boolean (bitwise) operators `|` `&` do not.

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`

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

```
beq A
B:
jmp C
A:
beq D
C:
beq B
D:
bne B
```

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:

```plaintext
10 print I
20 I = I + 1
30 IF I < 10 GOTO 10
```

A cleaner version in C using structured control flow:

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

Gotos vs. Structured Programming

Break and continue leave loops prematurely:

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

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 1μs to run, then?

Answer: loops

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

Prohibiting Index Modification

Optimizing the behavior of loops is often very worthwhile.

Some processors have explicit looping instructions.

Some compilers transform loop index variables for speed or safety.

Letting the program do whatever it wants usually prevents optimizations.

Empty Bounds

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

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

“for i = 10 to 1 by 1”

Test is done after the body.
Empty Bounds

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?

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

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:

```c
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).
```

Pre- and Post-test Loops

Most loops want their tests first to allow the possibility of zero iterations.

```c
struct foo *p = head; // Sum a linked list
while (p != 0) {
  total += p->value;
  p = p->next;
}
But it’s sometimes useful to place the test at the end:
```
```c
char line[80];
do {
  scanf("%s", line);
} while (line[0] == '#'); /* skip comments */
```

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;
```

Mid-test Loops

```
loop
  statements
  when condition exit
  statements
  when condition exit
  ...
end
```


Compare with Tiger’s `break`, which must fall within a `while` or `for`. More difficult to check (static semantics).
Multi-way Branching

```c
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

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

Obvious way:

```c
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:

```c
labels[1] = { L1, L2, L3, L4 }; /* Array of labels */
if (s>=1 && s<=4) goto l[s-1]; /* not legal C */
L1: one(); goto Break;
L2: two(); goto Break;
L3: three(); goto Break;
L4: four(); goto Break;
Break:
```

Recursion and Iteration

Consider computing

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

In C, the most obvious evaluation is iterative:

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

Recursion and Iteration

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

But this can also be defined recursively

```c
double sum(int i)
{
  double fi = f(i);
  if (i <= 10) return fi + sum(i+1);
  else return fi;
}
sum(0);
```

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);
}
```

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:

```c
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

Grammars make a similar choice:

Iteration:

```c
clist : item ("," item)* ;
```

Recursion:

```c
clist : item tail ;
tail : "," item tail |
| /* nothing */
```

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;
}
What is printed by
q( p(1), 2, p(3) );
```

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;
}
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.

```c
#define p(x) (printf("%d ",x), x)
#define q(a,b,c) total = (a), \n      printf("%d ", (b)), \n      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 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, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1
```