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(int i) { foo(i-1); }
5. Recursion 
   foo() 

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.

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:

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

Misbehaving Floating-Point Numbers

\[ 1e20 + 1e-20 = 1e20 \]
\[ 1e-20 \ll 1e20 \]
\[ (1 + 9e-7) + 9e-7 \neq 1 + (9e-7 + 9e-7) \]
\[ 9e-7 \ll 1, \text{ so it is discarded, however, } 1.8e-6 \text{ is large enough} \]
\[ 1.00001(1.00001 – 1) \neq 1.00001 \cdot 1.00001 – 1.00001 \cdot 1 \]
\[ 1.00001 \cdot 1.00001 = 1.00001100001 \text{ requires too much intermediate precision.} \]
What's Going On?
Floating-point numbers are represented using an exponent/significand format:

1 10000001 01100000000000000000000
8-bit exponent 23-bit significand
= \(-1.0111 \times 2^{129-127}\) = \(-1.375 \times 4\) = \(-5.5\).

What to remember:
1363.456846353963456293
represented rounded

Short-Circuit Evaluation
The section operator ? : does this, too.

\[
cost = \begin{cases} 
\text{disaster_possible} \Rightarrow \text{avoid_it}() & : \text{cause_it}(); \\
\text{cause_it}() & : \text{avoid_it}(); \\
\end{cases}
\]

cause_it is not called if disaster_possible is true.

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

\[
\text{if} (\text{disaster_possible} \text{ || } \text{case_it}()) \{ \ldots \}
\]

cause_it() only called if disaster_possible is false.
The && operator does the same thing.
Useful when a later test could cause an error:

\[
\text{int a}[10];
\text{if} (i \geq 0 \text{ && } i < 10 \text{ && } a[i] == 0) \{ \ldots \}
\]

Short-Circuit Operators in Tiger
From the LRM:
The logical operators \& and | are lazy logical operators.
They do not evaluate their right argument if evaluating
the left determines the result. Zero is considered false;
everything else is considered true.
We will dismantle these operators into conditional
branches in the next assignment.

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

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
So-called because it’s easy to create spaghetti:
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
- do..while
- break
- continue
- return

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:

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

An even better version

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

Again:

```c
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:

```java
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:

```pascal
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 1 µs 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.

Enumeration-Controlled Loops in FORTRAN

```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?
Loops in Tiger

The Tiger LRM:

The for expression, \texttt{for id := expr to expr do expr}, evaluates the first and second expressions, which are loop bounds. Then, for each integer value between the values of these two expressions (inclusive), the third expression is evaluated with the integer variable named by \texttt{id} bound to the loop index. The scope of this variable is limited to the third expression, and may not be assigned to. This expression may not produce a result and is not executed if the loop's upper bound is less than the lower bound.

Empty Bounds

In \textsc{fortran}, the body of this loop is executed once:

\begin{verbatim}
do 10 i = 10, 1, 1
...
10: continue
\end{verbatim}

"for \texttt{i} = 10 to 1 by 1"

Test is done after the body.

Scope of Loop Index

What happens to the loop index when the loop terminates?

Index is undefined: \textsc{fortran} IV, Pascal.
Index is its last value: \textsc{fortran} 77, Algol 60
Index is just a variable: C, C++, Java

Tricky when iterating over subranges. What's next?

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

Algol's Combination Loop

\texttt{\textcolor{red}{for -- for id := for-list do stmt}}

\texttt{for-list} \rightarrow \texttt{enumerator ( , enumerator )}

\texttt{enumerator} \rightarrow \texttt{expr}

\begin{verbatim}
-- expr step expr until expr
-- expr while condition
\end{verbatim}

Equivalents:

\begin{verbatim}
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 ...
\end{verbatim}

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.

\begin{verbatim}
struct foo *p = head; // Sum a linked list
while (p != 0) {
    total += p->value;
    p = p->next;
}
\end{verbatim}

But it's sometimes useful to place the test at the end:

\begin{verbatim}
char line[80];
do {
    scanf("%s", line);
} while (line[0] == '#'); /* skip comments */
\end{verbatim}

Algol's Combination Loop

Needlessly general, it turns out.

C's logically controlled loop retains most of the functionality:

\begin{verbatim}
for (i = 1 ; i < 10 ; i += 2) { ... }
\end{verbatim}

is equivalent to

\begin{verbatim}
i = 1;
while (i < 10) {
    ...
i += 2;
}
\end{verbatim}

Scope of Loop Index

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

\begin{verbatim}
for (int i = 0 ; i < 10 ; i++) {
    int a = i; // Was OK: i = 10 here
    a = a + i; // OK
}
\end{verbatim}

But this is awkward:

\begin{verbatim}
for (int i = 0 ; i < 10 ; i++) {
...for(int i = 0 ; i < 10 ; i++) // Error: i redeclared
    ...
for (int i = 0 ; i < 10 ; i++) // Error: i undefined
    ...
for (int i = 0 ; i < 10 ; i++) { // OK
    ...
}
\end{verbatim}

Rather annoying: broke many old C++ programs.
Better for new code.
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;
}

Implementing multi-way branches

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

Recurse 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

Grammars make a similar choice:

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

Recursion:
```
clist : item tail;
```

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.

Advantage: a syntactic construct.
Errors caught in parser.
Compare with Tiger’s break, which must fall within a while or for. More difficult to check (static semantics).
Tail-Recursion and Iteration

\[
\text{int } \gcd(\text{int } a, \text{ int } b) \{
    \text{if } (a == b) \text{ return } a;
    \text{else if } (a > b) \text{ return } \gcd(a-b, b);
    \text{else return } \gcd(a, b-a);
\}
\]

Can be rewritten into:

\[
\text{int } \gcd(\text{int } a, \text{ int } b) \{
    \text{start:}
    \text{if } (a == b) \text{ return } a;
    \text{else if } (a > b) a = a-b; \text{ goto start;}
    \text{else } b = b-a; \text{ goto start;}
\}
\]

Applicative- and Normal-Order Evaluation

\[
\text{int } p(\text{int } i) \{ \text{ printf("%d ", i); return } i; \}\\
\text{void } q(\text{int a, int b, int c}) \\
\{ \text{ int total = a; } \text{ printf("%d ", b); } \text{ total += c;} \}\\
q( p(1), 2, p(3) );
\]

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 Evaluation

Most languages use applicative order.
Macro-like languages often use normal order.

\[
\text{#define } p(x) \{ \text{ printf("%d ", x); return } x; \}\\
\text{#define } q(a,b,c) \text{ total = (a), } \text{ printf("%d ", (b)); } \text{ total += (c)}\\
q( p(1), 2, p(3) );
\]

Prints 1 2 3.
Some functional languages also use normal order evaluation to avoid doing work. “Lazy Evaluation”

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.

\[
\text{int } p(\text{int } i) \{ \text{ printf("%d ", i); return } i; \}\\
\text{int } q(\text{int a, int b, int c}) \\
q( p(1), p(2), p(3) );
\]
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 prevalent in concurrent languages.
Sometimes it’s convenient, though:

\[
\text{if } a \geq b \rightarrow \text{ max := a}\\\text{[] b \geq a \rightarrow max := b}\\\text{fi}
\]
Nondeterministic (irrelevant) choice when a=b.
Often want to avoid it, however.