Control flow mechanisms and their implementation

Order of execution matters for statements, and for expressions with side effects. Ordering for statements is control flow.

[Expressions have values; statements don’t. Statements are evaluated for their side effects. Expressions may have side effects, but don’t necessarily (and it’s often considered bad style).]

Principal paradigms for control flow:
  - sequencing
  - selection
  - iteration
  - subroutines, recursion (and related control abstractions—e.g., iterators)
  - nondeterminacy
  - concurrency

Expression evaluation

Operators are built-in functions with, often
  - special non-identifier names (+, −, %, ...) 
  - infix syntax (as opposed to prefix for most functions)
  - precedence and associativity
    - C has 15 levels -- too many to remember
    - Pascal has 3 levels -- too few for good semantics
    - Fortran has 8; Ada has 6
    - I don’t like the rules in any of these (Fortran probably closest)
      - Ada puts and, or at same level
      - Pascal misgroups if a = b or c = d then
    - Lesson: when unsure, use parentheses!

evaluation order matters
  - consider f(a+b, c, d(e, h), h)
d might have side effects
reordering might improve register allocation
some arguments might already be available in registers
on old machines with old compilers, last-to-first would optimize stack access

Most languages say order of evaluation of arguments is undefined.

Likewise, most leave operand evaluation order undefined
   e.g., in \( f(a, b) + g(c, d) \)

arithmetic identities work in math class; not always on a computer
   commutativity of + and * is usually safe
   associativity is not
   \((a + b) + c\) works if \(a \approx \text{minint}\) and \(b \approx \text{maxint}\) and \(c\) is small
   \(a + (b + c)\) may not (esp. in floating point)

Compilers generally respect parentheses; if in doubt, use them

short-circuiting
   \(\text{if (b != 0 && a/b == c)} \ldots\)
   \(\text{if (*p && p->foo)} \ldots\)
   \(\text{if (f || messy())} \ldots\)

connection to lazy evaluation of arguments -- e.g., in Haskell

Not all languages evaluate and and or lazily -- beware!

**Variables as values v. variables as references**
value-oriented languages
   C, Ada
reference-oriented languages
   most functional languages (Lisp/Scheme, SML/OCaml/Haskell)
   Smalltalk
Java deliberately in-between
   built-in types are values
   user-defined types are objects -- references
C# similar, though user can choose which object variables are values
   (“expanded”, in Eiffel terminology) and which are references
Assignment
statement (or expression) executed for its side effect(s)

assignment operators (+=, -=, etc)
handy
avoid redundant work (or need for optimization)
perform side effects exactly once

\[ A[f()] += 1 \] is not the same as \[ A[f()] = A[f()] + 1 \]

C --, ++
prefix v. postfix semantics
postfix more than syntactic sugar
\[ A[++i] = 3 \] is the same as \[ A[i += 1] = 3 \]
\[ A[i++] = 3 \] is much harder to emulate

Initialization v. assignment
esp. important in OO languages

```cpp
foo b;
   // calls no-arg constructor  foo::foo()  
foo f = b;
   // calls one-arg “copy constructor”  foo::foo(&foo)  
   // This is syntactic sugar for  foo f(b)  
```

```cpp
foo b, f;
   // calls no-arg constructor
f = b;
   // calls foo::operator=(&foo)
```

also matters in other languages:
globals can be statically initialized
requiring (or defaulting) initialization avoids use of garbage
also avoids some races in parallel programs, but has costs

Side Effects
often discussed in the context of functions
a side effect is some permanent state change caused by execution of function -- some noticeable effect of call other than return value.
in a more general sense, assignment statements provide the ultimate example of side effects. They change the value of a variable.

**Side effects are fundamental to imperative computing**

In (pure) functional, logic, and dataflow languages, there are no such changes. These languages are called *single-assignment* languages. They might better be called “simple definition” languages.

---

**Sequencing**

execute one statement after another

very straightforward; very imperative

---

**Selection**

sequential if statements

if ... then ... else

if ... then ... elsif ... else

(\(\text{cond}\)
  (\(\text{C1}\) \(\text{E1}\))
  (\(\text{C2}\) \(\text{E2}\))
  ...
  (\(\text{Cn}\) \(\text{En}\))
  (\(\text{T}\) \(\text{Et}\))
)

match \(e\) with
  | \(\text{pat1 when cond1} \rightarrow\)
  | \(\text{pat2 when cond2} \rightarrow\)
  | ...
  | \(\text{patN when condN} \rightarrow\)
  | _ \(\rightarrow\)

value of explicit terminators or begin/end (or `{}`) brackets

need for elsif (elif)
jump code

When translating

```
if A < B then ... else ... fi
```

one might evaluate the condition to get a Boolean value in a register, then branch depending on its value.

That’s often more instructions than needed:

```
r1 := A
r2 := B
r1 := r1 < r2
if !r1 goto L1
    <then clause>
    goto L2
L1:
    <else clause>
L2:
```

For expressions with short-circuiting, the difference is more compelling (Example 6.49 in the text):

```
if ((A > B) and (C > D)) or (E <> F) then
    then_clause
else
    else_clause
```

w/out short-circuiting (as in, e.g., Pascal):

```
r1 := A          -- load
r2 := B
r1 := r1 > r2
r2 := C
r3 := D
r2 := r2 > r3
```
\[ r_1 := r_1 \land r_2 \]
\[ r_2 := E \]
\[ r_3 := F \]
\[ r_2 := r_2 \not= r_3 \]
\[ r_1 := r_1 \lor r_2 \]
\[ \text{if } r_1 = 0 \text{ goto } L_2 \]

\[ L_1: \text{<then clause>} \quad -- \text{label not actually used} \]
\[ \text{goto } L_3 \]

\[ L_2: \text{<else clause>} \]

\[ L_3: \]

with short-circuiting (as in, e.g., C):

\[ r_1 := A \]
\[ r_2 := B \]
\[ \text{if } r_1 \leq r_2 \text{ goto } L_4 \]
\[ r_1 := C \]
\[ r_2 := D \]
\[ \text{if } r_1 > r_2 \text{ goto } L_1 \]

\[ L_4: r_1 := E \]
\[ r_2 := F \]
\[ \text{if } r_1 = r_2 \text{ goto } L_2 \]

\[ L_1: \text{then\_clause} \]
\[ \text{goto } L_3 \]

\[ L_2: \text{else\_clause} \]

\[ L_3: \]

Note that this not only avoids performing unnecessary comparisons; it also avoids the \textit{and} and \textit{or} instructions.

\textit{guarded commands}

\textit{example of non-determinacy}

\[ \text{if} \]
\[ \quad \text{cond1} \rightarrow \text{stmt1} \]
\[ \[ \] \text{cond2} \rightarrow \text{stmt2} \]
\[ \[ \] \ldots \]
\[ \[ \] \text{condN} \rightarrow \text{stmtN} \]
\[ \text{fi} \]

\textit{similar version for loops}

\textit{Fortran computed gotos}
case/switch (introduced in Algol-W)
labels required to be disjoint

what should happen if there isn’t a matching label for value?
Ada: forbid at compile time
C: no-op
Pascal: dynamic semantic error

case implementation
sequential testing
small number of choices, non-dense range
characteristic array (jump table)
dense range
hashing
non-dense range w/out range labels
binary search
large range, range labels
(probably don’t need search tree, except perhaps if the key
distribution is highly nonuniform and we want better pivots than
we get with mean)

Should ranges be allowed in the label list?
they make it easy to state things for which a jump table or
hash table is awful: can be done efficiently (O(log n)) with
binary search

examples:
3: 1: 1: 1..48:
5: 2: 59: 97..283:
7: 3: 187: 900..1024:
9: ... ... ...
100: 1000000: 12345..67890:
Iteration

logically controlled v. enumeration controlled
“while condition is true” v. “for every element of set”
In the latter case, the number of elements (and their identities) are known before we even start the loop (and in general, we don’t want the values we iterate over to depend on anything we do in early iterations).

----------------------------------------
Logically-controlled loops

pre-test (while)
post-test (repeat)
mod-test (one-and-a-half loops — loop with exit)
   labels for non-closest exit?

implementation options:

L1:
   r1 := <condition>
   if !r1 goto L2
   <loop body>
   goto L1
L2:

That has two branches in every iteration.

L1:
   r1 := <condition>
   if !r1 goto L2
L1:
   <loop body>
   r1 := <condition>
   if r1 goto L1
L2:

That evaluates the condition in two different places.
Not a big deal if it doesn’t bloat code size.
If it’s complicated (long code) we can do this instead:
goto L2
L1:
    <loop body>
test:
    r1 := <condition>
    if r1 goto L1

That has one extra jump, but only one copy of the test.

C-style for loop
    semantically clean, but not really a for loop
    hard to apply the various optimizations possible for “real” for loops
    for (int i = first; i <= last; i+= step) {
        ...
    }

~=

{  
    int i = first;
    while (i <= last) {
        ...
            i += step;
    }

----------------------------------------
Enumeration-controlled loops
    for v in my_set /* in my favorite order */ do
        ...
    end

Arithmetic progressions are a common case:
    /* Modula-2 syntax */
    i : integer;
    ...
    for i := first to last by step do
        ...
    end
How might we implement this? Consider

```plaintext
i := first
goto L2
L1: ...
i += step
L2: if i <= last goto L1
```

Several things can go wrong (generally fixed in Ada and Fortran 90, to some extent in Modula-2)

- empty bounds
  - shouldn’t execute (did in Fortran I)

- changes to bounds or step size within loop
  - calculated up front in modern languages

- direction of step
  - constant stepsize
  - “downto” (Pascal)
  - “in reverse” (Ada)

- changes to loop variable within loop
  - not generally allowed in modern languages

- value after the loop
  - especially at end-of-legal range for type (overflow?)

  if local to loop, can’t even name afterward, so it’s just an implementation issue, not a semantic one

- iteration count translation technique
  - needed in Fortran, which has run-time step
  - helpful any time the end value may be the last valid one
  - supported by “dec. and branch if nonzero” instruction on many machines:

```plaintext
r1 := first
r2 := step
r3 := last
```
r3 := ⌊(r3−r1+r2)/r2⌋
if r3 ≤ 0 goto L2
L1: <loop body>
   r1 := r2
   if r3 > 0 goto L1
L2:

gotos in and out
modern languages allow only out, and structure as
exit/break/return (or exception)

Iterators

supply a for loop with the members of a set
abstraction/generalization of the “from A to B by C” sorts of
stuff you see built-in in older languages

pioneered by Clu:
   for i in <iterator> do ... end
   built-in iterators for from_to, from_to_by, etc.

wonderful for iterating over arbitrary user-defined sets
   very good for abstraction; for loop doesn’t have to know
   whether set is a linked list, hash table, dense array, etc.

may be true iterators (as in Clu, C#, Icon, Python, Ruby) or interface-based
approximation (“iterator objects,” as in Euclid, Java, and C++)

in Python:

    def uptoby(lo, hi, step):
        while True:
            if (step > 0 and lo > hi) \
                or (step < 0 and lo < hi): return
            yield lo
            lo += step  # ignore overflow

    for i in uptoby(1, 20, 2):
        print (i)
Iterator objects (Euclid, C++, Java)

Standard interface for abstraction to drive for loops. Supported in Euclid and Java with special loop syntax, and in C++ through clever use of standard constructor and operator overload mechanisms.

In Java:

```java
List<foo> myList = ...;
for (foo o : myList) {
    // use object o
}
```

requires that the to-be-iterated class (here, List) implements the Iterable interface, which exports a method

```java
public Iterator<T> iterator()
```

where Iterator is an interface exporting methods

```java
public boolean hasNext()
```

and

```java
public T next()
```

The for loop is syntactic sugar for

```java
for (Iterator<foo> i = myList.iterator(); i.hasNext();) {
    foo o = i.next();
    // use object o
}
```

C++ version looks like

```cpp
list<foo> my_list;
...
for (list<foo>::const_iterator i = my_list.begin(); i != my_list.end(); i++) {
    // make use of *i or i->field_name
}
```
Don’t have to have an equivalent of the Iterator interface (it’s just a convention), because C++ individually type-checks every use of a generic (template).

Note the different conceptual model:
Java has a special for loop syntax that uses methods of a special class
C++ standard library defines iterators as “pointer-like” objects with increment operations to drive ordinary for loops

All the standard library collection/container classes support iterators, in both languages.

True iterators (Clu, Icon, C#, Python)

iterator itself looks like a procedure, except it can include “yield” statements that produce intermediate values. when the iterator returns, the loop terminates

C# for loop resembles that of Java:

```
foreach (foo o in myList) {
    // use object o
}
```

This is syntactic sugar for

```
for (IEnumerator<foo> i = myList.GetEnumerator(); i.MoveNext()) {
    foo o = i.Current;
    // use object o
}
```

Current is an accessor -- a special method supporting field-like access:

```
public object Current {
    get {
        return ...;
    }
```
In contrast to Java, you don’t need to hand-create the `hasNext()` [MoveNext()] and `next()` [Current] methods. The compiler does this automatically when your class implements the `IEnumerable` interface and has an iterator -- a method containing “yield return” statements and “returning” an `IEnumerator`:

```csharp
class List : IEnumerable {
    ... 
    public IEnumerator GetEnumerator() {
        node n = head;
        while (n != null) {
            yield return n.content;
            n = n.next;
        }
        // NB: no return statement
    }
}
```

If you want to be able to have multiple iteration orders, your class can have multiple methods that each return an `IEnumerator`. Then you can say, e.g.

```csharp
foreach (object o in myTree.InPreOrder) { ... 
foreach (object o in myTree.InPostOrder) { ... 

detail:
    IEnumerator implements MoveNext and Current (also Reset)
    IEnumerable implements GetEnumerator, which returns an IEnumerator
```

Loop body as lambda (Smalltalk, Scheme, ML, Ruby, ...)

OCaml:
```ocaml
open Printf;;
let show n = printf "%d\n" n;;

let upto lo hi =
fun f -> let rec helper i =  
    if i > hi then ()  
    else (f i ; helper (i + 1)) in  
helper lo;;

 upto 1 10 show;; =>
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
  - : unit = ()

Ruby:

    sum = 0
    [ 1, 2, 3 ].each { |i| sum += i } => [1, 2, 3]  # array itself
    sum
    => 6

Here the (parameterized) brace-enclosed block is passed to the each method as a parameter.

There’s also more conventional-looking syntax:

    sum = 0
    for i in [1, 2, 3] do # ‘do’ is optional
        sum += i
    end
    sum

The for loop is syntactic sugar for a call to each.

Here’s a more object-oriented alternative:

    sum = 0
    1.upto 3 { |i| sum += i}
    sum
or instead of using braces:

```ruby
sum = 0
i.upto 3 do |i| sum += i end
sum
```

You can write your own iterators using ‘yield’.

```ruby
class Array
def find
  for i in 0...size
    value = self[i]
    return value if yield(value)
  end
  return nil
end
...
[1, 3, 5, 7, 9].find { |v| v*v > 30 } => 7
```

Think of yield as invoking the block that was juxtaposed (“associated”) with the call to the iterator.

(FWIW, the array class already has a `find` method in Ruby, but we can redefine it, and it probably looks like this anyway.)

Blocks can also be turned into first-class closures, with unlimited extent:

```ruby
def nTimes(aThing)
  # Ruby, like most scripting languages, is dynamically typed
  return proc { |n| aThing * n }
end

In recent Ruby, `->` is a synonym for proc
```

```ruby
p1 = nTimes(3)
p2 = nTimes("foo")
p1.call(4) => 12
p2.call(4) => "foofoofoofoo"
```
This lets us build higher-level functions. Here’s reduction for arrays:

```ruby
class Array
  def reduce(n)
    each { |value| n = yield n, value }  # that’s self.each
      # yield invokes (just once) the block associated
      # with the call to reduce. Note the lack of parens:
      # "yield (n, value)" would pass a single tuple.
    n  # return value
  end
  def sum
    reduce(0) { |a, v| a + v }
  end
  def product
    reduce(1) { |a, v| a * v }
  end
end

[2, 4, 6].sum => 12
[2, 4, 6].product => 48
```

All in all Ruby is pretty cool. Check it out.
(I do wish it let you associate more than one block with a call.)

Implementation of true iterators (section 9.5.3-CS)

- coroutines or threads
  - overkill

- single-stack
  - used in Clu
    - works, but would confuse a standard debugger, and not compatible
      with some conventions for argument passing

- implicit iterator object
  - kinda cool; used in C# and Python
    - same mechanism supports async in C# and JavaScript

- block as lambda expression (Ruby, functional languages)
Recursion

equally powerful to iteration, and as efficient when you can use tail recursion. mechanical transformations back and forth
often more intuitive (sometimes less)
naive implementation less efficient
no special syntax required
fundamental to functional languages like Scheme

tail recursion

(* OCaml: *)
let rec gcd b c =
  if b = c then b
  else if b < c then gcd b (c - b)
  else gcd (b - c) c;;

implemented as

gcd (b c)
start:
  if b = c
    return b
  if b < c
    c := c - b
    goto start
  if b > c
    b := b - c
    goto start

changes to create tail recursion (e.g. pass along an accumulator)

(* OCaml: *)
let rec summation f low high =
  if low == high then f low
  else f low + summation f (low+1) high;;

becomes

let rec summation2 f low high st =
  if low == high then st + f low
else summation2 f (low+1) high (st + f low);;

and then

let summation3 f low high =
let rec helper low st =
    let new_st = st + f low in
    if low == high then new_st
    else helper (low+1) new_st in
helper low 0;;

More generally (absent an associative operator), pass along a continuation.

This is perfectly natural to someone used to programming in a functional language. Note that the summation example depends for correctness on the associativity of addition. To sum the elements in the same order we could have counted down from high instead of up from low, but that makes a more drastic change to the structure of the recursive calls.

There is no perfectly general algorithm to discover tail-recursive versions of functions, but compilers for functional languages recognize all sorts of common cases.

Sisal and pH have “iterative” syntax for tail recursion:

    function sum (f : function (n : integer returns integer),
                       low : integer, high : integer returns integer)
    for initial
        st := f (low);
    while low <= high
        low := old low + 1
        st := old st + f (low)
    returns value of st
    end for
end function

The Sisal compiler was really good at finding tail recursive forms.

Concurrency specifies that statements are to occur (at least logically) concurrently. Concurrency is fundamental to probably half the research in computer science today.
Nondeterminacy
choice “doesn’t matter”
periodically popular, promoted by Dijkstra for use with selection
*(guarded command* syntax)*
can apply to execution order as well
useful for certain kinds of concurrency

```
process server
do
    receive read request ->
    reply with data
[]
    receive write request ->
    update data and reply
od
```

also nice for certain axiomatic proof schemes
raises issues of “randomness”, “fairness”, “liveness”, etc.