# Control Flow

- Order of instructions is a crucial component of "telling another human being what one wants the computer to do" (Knuth)
- Seven forms of control flow:
  - *sequencing*
    - includes expressions
  - *selection:* choosing among alternatives (thus a.k.a. *alternation*)
  - *iteration:* repeating a fragment of code
  - *procedural abstraction:* grouping code into callable units (subroutines)
  - *recursion:* code that is defined in terms of itself
  - *concurrency:* perceived simultaneous execution/evaluation of code
  - *non-determinacy:* no specific ordering of execution, implying that any order will lead to the desired result
- "A programmer who thinks in terms of these categories…will find it easy to learn new languages…and design and reason about algorithms in a language-independent way."

# Much Ado About Goto

- Control flow constructs trace their roots to assembly language jumps and branches
- The earliest languages had something that approximated that very closely: *goto*
  - Heavy use in ForTran:

```
    do 100 i = 1, 10, 2
        ...
100: continue
```

  - Problematic in the context of many of today's languages
    - *goto* in mid-loop: replaceable *continue* (C, Java)
    - *goto* in mid-subroutine: explicit *return* (many languages)
    - *goto* due to errors: exceptions (C++, Java, ML, etc.)
- The move away from *goto* is embodied in *structured programming* — the "object-oriented programming" of the 70s

# Sequencing Miscellany

- Key issue for imperative languages, whose main mechanism is side effects
  - Distinction between "statements" and "functions" or "expressions"
  - Some languages expressly disallow the latter ("functions" or "expressions") from having side effects
  - One of my favorite words: expressions without side effects are known as *idempotent* — given the same arguments, they yield the same result regardless of when or in what order they are evaluated
    - Watch out, I may digress while talking about idempotence :)
  - In functional languages, of course, the emphasis is the other way around
- Certain functions explicitly need side effects: random number generators, name generators
- Compound statements or functions: when aggregated and viewed as an expression, the value of a *block* or *compound statement* is the value of its last component expression or statement

# Selection

- First appeared in Algol 60
- Variations:
  - separate *elsif* keyword to avoid excessive nesting and to facilitate easier parsing (as you may recall from Chapter 2)
  - rearranging clauses and conditions for greater readability, particularly Perl:
    - *unless* variant
    - switching the *if*/*unless* clause and the statement to execute

      ```
      go_outside() and play() unless $is_raining;
      print "Basset hounds have long ears" if $earLength >= 10;
      ```
  - conditionals as part of the *language library* and not its syntax (Smalltalk):

    ```
    value isNull ifTrue: [ ... ] ifFalse: [ ... ]
    ```
    - "`value isNull`" evaluates to a Boolean object
    - the Boolean class has a method called `ifTrue:ifFalse:`, which takes a code block to execute (expressed as the literal "`[ ... ]`")
- Short-circuiting can be used for more efficient generated code

# Case/Switch Selection

```
case (expr) of
    1: ...
    2, 7: ...
    3..5: ...
    10: ...
    else ...
end
```

```
switch (expr) {
    case 1: ...; break;
    case 2:
    case 7: ...; break;
    case 3: case 4:
    case 5: ...; break;
    case 10: ...; break;
    default: ...; break;
}
```

- Syntactically simpler, with implementation consequences
  - Instead of boolean evaluation/jumps, case/switch selection can use a "jump table" — see Figure 6.4 in Scott
- Semantic issue: to fall through (C, C++, Java) or not to fall through (Pascal, Modula)
- ML function matching looks similar, though must be in the context of a function, and is significantly more powerful

```
(*
 * roman: int -> string
 *
 * Returns the roman numeral equivalent of its input.  Raises an exception
 * if the input is non-positive.
 *)
local
    val symbols = [ (1000, "M"), (900, "CM"), (500, "D"), (400, "CD"), (100, "C"),
(90, "XC"), (50, "L"), (40, "XL"), (10, "X"), (9, "IX"),
(5, "V"), (4, "IV"), (1, "I") ];

    (*
     * Helper: r n symbols result => returns the roman equivalent of n
     * appended to result, using only the translations in the mapping
     * called symbols.
     *)
    fun r 0 symbols result = result
      | r n [] s = raise Fail "Cannot happen"
      | r n (symbols as (value, rep) :: tail) result =
        if n >= value then
            r (n - value) symbols (result ^ rep)
        else
            r n tail result
in
    fun roman n =
        if n <= 0 then
            raise Fail "No Roman equivalent"
        else
            r n symbols ""
end;
```

# Iteration

- Loops — without them, a program is strictly finite
- Two kinds of loops:
  - *enumeration-controlled*: do something for each element in a collection
  - *logically controlled*: do something while a condition is true or false

- Enumeration-controlled loops, the first generation
  - The classic "for loop" — enumerations restricted to ranges of numbers
  - Parts: index variable, start value, end value, optional step (also implies direction); also, many strict rules on what can and cannot change

```
for i := 5 to 20 by 2 do ...
```

  - Generalization: this really defines a set of discrete values, and the "loop body" is executed for each of these values…leading to the next generation of enumeration-controlled loops, based on iterators
  - Smalltalk again: for loops are methods of the Number classes

```
5 to: 20 by: 2 do: [ :i | ... ]
```

# Logically Controlled Loops

- When to test the condition?
  - *pre-test*: test the condition before entering the loop (while)
  - *post-test*: at least one pass through the loop (do-while, repeat-until)
  - *midtest*: no need to wait until the end of the loop block (exit, break)
    - If standalone keyword, need a static semantic check to make sure that the keyword is only used within a loop
    - Some languages combine the test condition with the exit construct (Ada: `exit when all_blanks(line, length)`)
    - For nested loops, the exit/break directive can specify how many "levels" of loop to exit (Ada, Java)

```
search: for (int i = 0; i < arrayOfInts.length; i++) {
    for (int j = 0; j < arrayOfInts[i].length; j++) {
        if (arrayOfInts[i][j] == searchfor) {
            foundIt = true;
            break search;
        }
    }
}
```

note *search* is an identifier, not a "goto label" !

# Logically Controlled Loops, cont’d

- Interesting variations (either for convenience, or based on the “spirit” of the language)
  - Perl: separate *continue* block, distinct midtest loop exit statements (*next*, *last*, *redo*)

```
LINE: while (<STDIN>) {
    next LINE if /^#/; # Skip the rest of the loop w/ continue.
    last LINE if /^$/; # Exit the LINE loop; no continue.
    if (s/\\$//) { redo LINE unless eof(); } # Do over; no continue.
    # Do something with the input (like print)...
} continue {
    $count++;
}
```

  - C/C++/Java: the *for* loop is really a logically controlled variant
  - Smalltalk: you guessed it, logically controlled loops are not part of the syntax but a method of a Block object

```
[ input := .... input isEmpty] whileTrue
```

# Enumeration-Controlled Loops: the Next Generation

- Explicitly define the collection over which loop is to operate
  - Maintains index variable from first-generation enumeration
  - All others are implicit in the collection
  - Iteration may be explicit or implicit

```
// Java < 1.5
for (Iterator it = coll.iterator(); it.hasNext(); ) {
    Object nextValue = it.next();
    ...
}

"Smalltalk" "(double quotes delimit comments in Smalltalk)"
employees do: [ :emp | emp name printOn: systemOut ].

# Perl
foreach $arg (@ARGV) { ...$arg... }

// Java >= 1.5
for (String s: stringColl) System.out.println(s);
```

# Recursion

- Frequently makes certain algorithms easy to write, though not required: recursion and logically controlled iteration have equivalent computational power

- Iteration feels more natural in imperative languages, while recursion feels more natural in functional languages

- Efficiency depends on implementation
  - Naïve implementation on either side tends to favor iteration
  - Certain forms of recursion, such as tail recursion, can be very efficient

- No extra syntax needed: just allow a function to call itself from its own body (or for multiple functions to call each other cyclically)

# Tail Recursion

- Primary argument for less efficiency in recursion is the cost incurred by a subroutine call: stack allocation, other bookkeeping
- *Tail recursion* eliminates this overhead: a *tail-recursive function* is a specific form of recursion where no additional computation follows a recursive call; i.e. the recursive call, if performed, is the final computation in the function

```
fun gcd a b =
    if a = b then
        a
    else
        if a > b then
            gcd (a - b) b
        else
            gcd a (b - a);
```

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

# Tail Recursion Helpers

- Many recursive functions that are not initially tail recursive can be transformed using (preferably locally-scoped) helpers

```
fun sum f low high =
    if low = high then
        f low
    else
        f low + sum f (low + 1) high
```

```
local
    fun sumhelper f low high subtotal =
        if low = high then
            subtotal + f low
        else
            sumhelper f (low + 1) high (subtotal + f low)
in
    fun sum f low high = sumhelper f low high 0
end;
```

# Applicative- and Normal-Order Evaluation

- Applicative-order evaluation: evaluate all arguments before passing to a subroutine
  - Used by most languages for subroutine evaluations
- Normal-order evaluation: evaluate arguments only when needed
  - Used by macros, such as in C

```
int square(int n) { return n * n; }
#define SQUARE(n) ((n) * (n))
```

- Beware of side effects in normal-order evaluation

```
int x = square(y++);
int x = SQUARE(y++); // becomes ((y++) * (y++))
```

- How about unit testers, particularly, testing for failure:

```
// Suppose toRoman() throws an exception.
assertFail(toRoman(-5));
```

# Non-Determinacy

- Already have some kind of non-determinacy with expression evaluation: `f(x) + g(x) + h(x)`
- Guarded command notation [Dijkstra]

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

  - Any command whose guard is true may execute, but there is no specification on which one will run
  - Variations on whether at least one guard must be true, or whether an *else* option is provided if no guard is true
- Non-determinism useful in concurrency
- How to choose the guarded command?
  - Randomization? Circular list (i.e. round robin)? — see Scott p. 307
  - "Fairness" = a guard that can be true infinitely often should be selected infinitely often

# Guarded Loops

- Compare these:

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

```
int gcd(int a, int b) {
    while (a > b) -> a = a - b
    [] (b > a) -> b = b - a;
    return a;
}
```

---

```
void server() {
    while (true) {
        if (read())
            processIn();
        else if (write())
            processOut();
    }
}
```

```
void server() {
    while (read()) -> processIn();
    [] (write()) -> processOut();
    [] true -> /* no-op */ ;
}
```