Control Structures

Programming Languages

CS 214

Spaghetti Coding

In the early 1960s, spaghetti coding was common practice, whether using a HLL or a formal model like the RAM...

Example: What does this "spaghetti style" C function do?

```
double f(double n) {
double x = y = 1.0;
if (n < 2.0) goto label2;
label1: if (x > n) goto label2;
y *= x;
x++;
goto label1;
label2: return y;
}
```

→ The *structure does not indicate the flow of control*

→ Such code was expensive *to maintain*...

Control Structures

In 1968, Dijkstra published “Goto Considered Harmful”
-- a letter suggested the *goto* should be outlawed because it encouraged undisciplined coding (the letter raised a furor).

Language designers began building control structures
-- statements whose syntax made control-flow obvious:

- If — Fortran
- Case — Algol-W
- For — Algol-60
- Do — COBOL
- If-Then-Else — COBOL
- If-Then-Elsif — Algol-68
- While — Pascal

With Pascal (1970), all of these were available in 1 language, resulting in a new coding style: *structured programming*.

Structured Programming

Structured Programming emphasized *readability*, through:

- –Use of appropriate control structures
- –Use of descriptive identifiers
- –Use of white space (indentation, blank lines).

```
double factorial(double n)
{
  double result = 1.0;

  for (int count = 2; count <= n; count++)
      result *= count;

  return result;
}
```

With structured programming, the flow of control is clear!

The resulting programs were *less expensive to maintain*.

Sequential Execution

A C/C++ block has 0 or more statements:

<block-stmt> ::= { <stmt-list> }

<stmt-list> ::= <stmt> <stmt-list> | ε

```
{
  <stmt>
    :
  <stmt>
}
```

An Ada block has 1 or more stmts:

<block-stmt> ::= begin <stmt-list> end

<stmt-list> ::= <stmt> <more-stmts>

<more-stmts> ::= <stmt> <more-stmts> | ε

```
begin
  <stmt>;
    :
  <stmt>;
end
```

The block is the control structure for sequential execution the default control structure in imperative languages.

The guiding principle for control structures is:

One entry point, one exit point.

Smalltalk

Smalltalk also has a *block* construct, but it is an object

<block-object>	::= [<params> <locals> <expr-list>]
<params>	::= <param-list> ‘|’ | ε
<param-list>	::= : id <param-list> | ε
<locals>	::= ‘|’ <id-list> ‘|’ | ε
<id-list>	::= id <id-list> | ε
<expr-list>	::= <expr> <more-exprs> | ε
<more-exprs>	::= . <expr> <more-exprs> | ε

```
[
  <expr>.
  <expr>.
    :
  <expr>
]
```

Smalltalk computations consist of *messages* sent to *objects*:

[2 + 1] value → 3

Like C/C++, a Smalltalk *block* can declare local variables;
but as an object, a Smalltalk *block* can also have parameters:

[:i | i + 1] value: 2 → 3

| aBlock | aBlock := [:x :y | (x*x) + (y*y)] . aBlock value: 3 value: 4 → 25

Lisp

The expressions in the "body" of a Lisp function are executed sequentially, by default, with the value of the function being the value of the last (final) expression in the sequence

```
(defun summation (n)
        (setq t1 (+ n 1))
        (setq t2 (* n t1))
        (setq t3 (/ t2 2)))
summation
(summation 100)
5050
```

Of course, *summation()* can be written more succinctly:

```
(defun summation (n)
       (defun summation (n)
           (/ (* (+ n 1) n) 2)
       ))
```

Lisp (ii)

Some Lisp function-arguments must be a single expression.

Lisp's *progn* function can be used to execute several expressions sequentially, much like other languages' *block*:

```
(defun summation (n)
    (if (<= n 0)
        (progn
            (message "summation(n): n must be positive...")
            0)
        (/ (* (+ n 1) n) 2)))
```

The *progn* function returns the value of its *final expression*.

Lisp also has sequential *prog1* and *prog2* functions, that return the values of the 1st and 2nd expressions, respectively.

Selective Execution

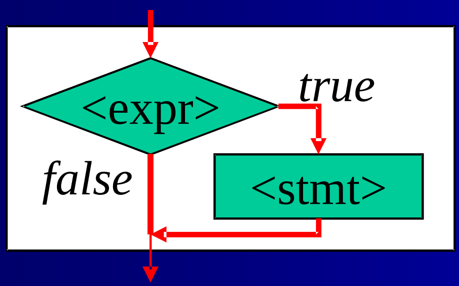
… lets us select/execute one statement and ignore another.

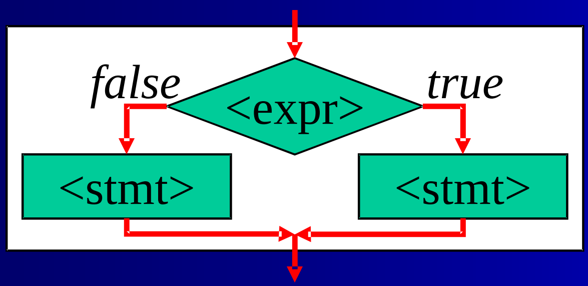
The *If Statement* provides selective execution:

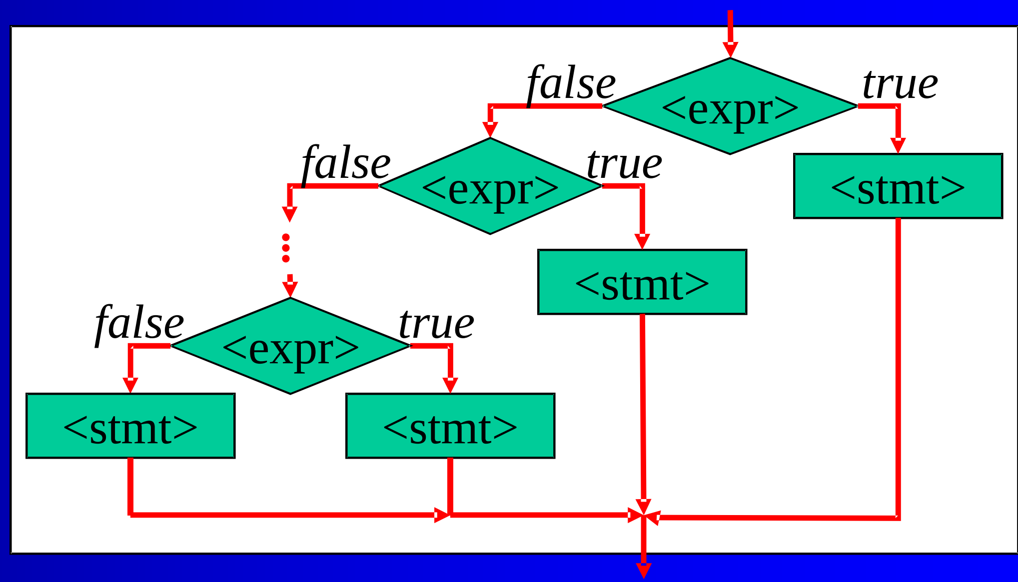
<if-statement> ::= if (<expr>) <stmt> <else-part>

<else-part> ::= else <stmt> | ε

These rules permit three different forms of flow control:







Examples

These three forms allow us to use selective execution in whatever manner is appropriate to solve a given problem:

Single-Branch Logic:

```
if (numValues != 0)
   avg = sum / numValues;
```

Dual-Branch Logic:

```
if (first < second)
   min = first;
else
   min = second;
```

Multi-Branch Logic:

```
if (score > 89)
   grade = 'A';
else if (score > 79)
   grade = 'B';
else if (score > 69)
   grade = 'C';
else if (score > 59)
   grade = 'D';
else
   grade = 'F';
```

The Dangling Else Problem

Every language designer must resolve the question of how to associate a "dangling else" following nested if statements...

```
if Condition1 then
      if Condition2 then
            Statement1
  else
      Statement2
```

?

The problem occurs in languages with *ambiguous grammars*.

→ Such a statement can be *parsed* in two different ways.

There are two different approaches to resolving the question:

- Add a semantic rule to resolve the ambiguity; vs.
- Design a statement whose syntax is not ambiguous.

Using Semantics

Languages from the 1970s (Pascal, C) tended to use simple but ambiguous grammars:

<if-stmt> ::= if (<expr>) <stmt> <else-part>
<else-part> ::= else <stmt> | ε

plus a semantic rule:

An else always associates with the nearest unterminated if.

```
if ( Condition1 )
      if ( Condition2 )
            Statement1
      else
            Statement2
```

```
if ( Condition1 )
{
      if ( Condition2 )
            Statement1
}
else
      Statement2
```

Block statements provided a way to circumvent the rule.

Newer C-family languages (C++, Java) have inherited this.

Using Syntax

Newer languages tend to use syntax that is unambiguous:

```
<if-stmt>     ::= if ( <expr> ) <stmt-list> <else-part> end if
<else-part>   ::= else <stmt-list> | ε
<stmt-list>   ::= <stmt> <stmt-list> | ε
```

Terminating an *if* with an *end if* "closes" the most recent *else*, eliminating the ambiguity without any semantic rules:

```
if ( Condition1 )
    if ( Condition2 )
        StmtList1
    else
        StmtList2
    end if
end if
```

```
if ( Condition1 )
    if ( Condition2 )
        StmtList1
    end if
else
    StmtList2
end if
```

Ada, Fortran, Modula-2, … use this approach.

Using Syntax (ii)

Perl uses a (different) syntax solution:

```
<if-stmt>     ::= if ( <expr> ) <block> <else-part>
<else-part>   ::= else <block> | ε
<block>       ::= { <stmt-list> }
```

By requiring each branch of an *if* to be a *block*, any nested if is enclosed in a block, eliminating the ambiguity:

```
if ( Condition1 ) {
      if ( Condition2 ) {
            StmtList1
      } else {
            StmtList2
      }
}
```

```
if ( Condition1 ) {
      if ( Condition2 ) {
            StmtList1
      }
} else {
      StmtList2
}
```

The end of the block serves to terminate the nested *if*.

Aesthetics

Multibranch selection can get clumsy using *end if*:

```
if ( Condition1 )
     StmtList1
else if ( Condition2 )
          StmtList2
     else if ( Condition3 )
             StmtList3
          else
             StmtList4
          end if
     end if
end if
```

```
if ( Condition1 )
     StmtList1
elsif ( Condition2 )
     StmtList2
elsif ( Condition3 )
     StmtList3
else
     StmtList4
end if
```

To avoid this problem, Algol-68 added the *elif* keyword that, substituted for *else if*, extends the same if statement.

Modula-2 and Ada replaced the error-prone *elif* with *elsif*.

Exercise

Write a BNF for Ada *if*-statements. Sample statements:

```
if numValues <> 0 then
   avg := sum / numValues;
end if;
```

```
if first < second then
   min := first;
   max := second;
else
   min := second;
   max := first;
end if;
```

```
if score > 89 then
   grade := 'A';
elsif score > 79 then
   grade := 'B';
elsif score > 69 then
   grade := 'C';
elsif score > 59 then
   grade := 'D';
else
   grade := 'F';
end if;
```

<Ada-if-stmt> ::= if <condition> then <stmt_list> <elsif_part> <opt_else-part> end if;

<elsif_part> ::= elsif <condition> then <stmt_list> <elsif_part> | epsilon

<opt_else-part> ::= else <stmt_list> | epsilon

Lisp's *if*

Lisp provides an if function as one if its expressions:

<if-expr> ::= (if <predicate> <expr> <opt-expr>)
<opt-expr> ::= <expr> | ε

Semantics: If the <predicate> evaluates to non-nil (i.e., not ()),
the <expr> is evaluated and its value returned;
else the <opt-expr> is evaluated and its value returned.

```
(if (> score 89)
  (setq grade "A")
  (if (> score 79)
   (setq grade "B")
   (if (> score 69)
    (setq grade "C")
    (if (> score 59)
     (setq grade "D")
     (setq grade "F")))))
```

It is not unusual for a Lisp expression to end with))))

-*Lost in Silly Parentheses*

Selection in Smalltalk

Smalltalk provides various *if True: and ifFalse: messages* that can be sent to <boolean objects>...

<selection-msg>	::=	<ifT-msg> | <ifF-msg> | <ifTF-msg> | <ifFT-msg>
<ifT-msg>	::=	ifTrue: <block>
<ifF-msg>	::=	ifFalse: <block>
<ifTF-msg>	::=	ifTrue: <block> ifFalse: <block>
<ifFT-msg>	::=	ifFalse: <block> ifTrue: <block>

```
n ~= 0
  ifTrue: [ avg := sum / n ]
```

```
first < second
  ifTrue:  [ min := first]
  ifFalse: [ min := second]
```

```
score > 89
 ifTrue: [grade:= 'A']
 ifFalse: [
  score > 79
   ifTrue: [grade:= 'B']
   ifFalse: [
    score > 69
     ifTrue: [grade:= 'C']
     ifFalse: [ … ] ] ]
```

These four are the only selection messages Smalltalk provides.

Problem: Non-Uniform Execution

```
if (score > 89)
   grade = 'A';      ←
else if (score > 79)
   grade = 'B';      ←
else if (score > 69)
   grade = 'C';      ←
else if (score > 59)
   grade = 'D';      ←
else
   grade = 'F';      ←
```

1 comparison to get here

2 comparisons to get here

3 comparisons to get here

4 comparisons to get here…

… and here

The times to execute different branches are different

- The 1st <stmt> executes after 1 comparison.
- The nth and final <stmt> execute after n comparisons.

The time to execute successive branches increases *linearly*.

The Switch Statement

The switch statement provides uniform-time multibranching.

<switch> ::= switch (<expr>) { <pair-list> <opt-default>}
<pair-list> ::= <case-list> <stmt-list> <pair-list> | ε
<case-list> ::= case <literal> : <case-list> | ε
<opt-default> ::= default: <stmt-list> | ε

Rewriting our grade program:

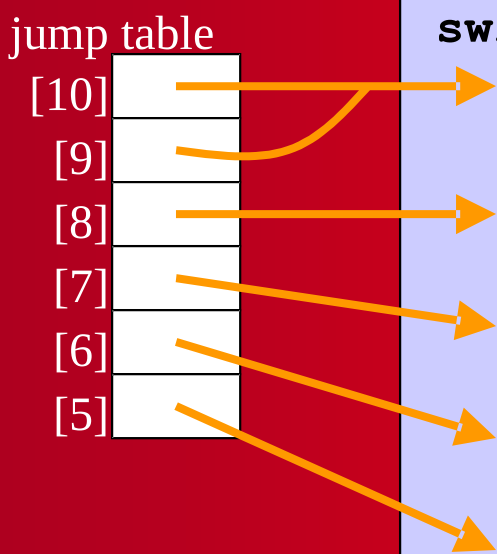
Note: If you neglect to supply *break* statements, control by default flows sequentially through the *switch* statement.

The *break* is a restricted-goto statement...

```
switch (score / 10) {
   case 9: case 10:
      grade = 'A'; break;
   case 8:
      grade = 'B'; break;
   case 7:
      grade = 'C'; break;
   case 6:
      grade = 'D'; break;
   default:
      grade = 'F';
}
```


Uniform Execution Time

Compiled *switch/case* statements achieve uniform response time via a *jump table*, that stores the address of each branch.




```
switch (score / 10) {
  case 9: case 10:
     grade = 'A'; break;
  case 8:
     grade = 'B'; break;
  case 7:
     grade = 'C'; break;
  case 6:
     grade = 'D'; break;
  default:
     grade = 'F';
}
```

This is simplified a bit, but it gives the general idea...

Uniform Execution Time (ii)

With a jump table, a compiler can translate a *switch/case* to something like this:

```
// code to evaluate <expr>
// and store it in register R

      cmp R, #highLiteral
      jle lowerTest
      mov #lowLiteral-1, R
      jmp  makeTheJump

lowerTest:
      cmp R, #lowLiteral
      jge makeTheJump
      mov #lowLiteral-1, R

makeTheJump:
      mov jumpTable[R], PC

// branches of the switch
```

For non-default branches, a *switch/case* needs 2 compares and 1 move to find the branch.

When a multibranch *if* does three or more comparisons a *switch* is probably faster.

A compiler spends compile time and space (to build the jump table) to decrease the average run-time needed to find a branch.

The Case Statement

The *switch* is a descendent of the *case* statement (Algol-W).

Only C-family languages use the *switch* syntax.

Unlike the *switch*, a *case* statement does not have drop-through behavior.

Most *case* stmts also let you use literal *lists* and ranges:

Ada uses the *when* keyword to begin each <literal-list>, and uses the => symbol to terminate each literal-list.

```
case score / 10 of
   when 9, 10 =>
      grade = 'A';
   when 8 =>
      grade = 'B';
   when 7 =>
      grade = 'C';
   when 6 =>
      grade = 'D';
   when 0..5 =>
      grade = 'F';
   when others =>
      put_line("error…");
end case;
```


Exercise

Build a BNF for Ada's *case* statement.

- There must be at least one branch in the statement.
- A branch must contain at least one statement.
- The *when others* branch is optional, but must appear last.

<Ada-case> ::= case <int-compatible-expression> of <when-clauses> <others-clause> end case; (again, semicolon optional)
<when-clauses> ::= <when-clause> <more-clauses>
<when-clause> ::= when <literals> => <stmt_list>
<more-clauses> ::= <when-clause> <more-clauses> | ε
<literals> ::= <literal-list> | <literal-range>
<literal-list> ::= <literal> <more-literals>
<more-literals> ::= , <literal> <more-literals> | ε
<literal-range> ::= <literal> .. <literal>
<others-clause> ::= when others => <stmt_list> | ε

Lisp

Lisp provides a *cond function* that looks similar to a *case*.

<cond-expr> ::= (cond <expr-pairs>)
<expr-pairs> ::= (<predicate> <expr>) <expr-pairs> | ε

However Lisp's *cond* uses arbitrary predicates (relational expressions) instead of literals.

```
(cond
 ((> score 89) "A")
 ((> score 79) "B")
 ((> score 69) "C")
 ((> score 59) "D")
 (t "F")
)
```

→ As a result, Lisp's *cond* cannot employ a jump table, so it has the same non-uniform execution time as an *if*.

The predicates are evaluated *sequentially* until a true <predicate> is found; its <expr> is then evaluated.

Repetition

A third control structure is repitition, or *looping*.

The C++ *while* loop is a *pretest* loop:

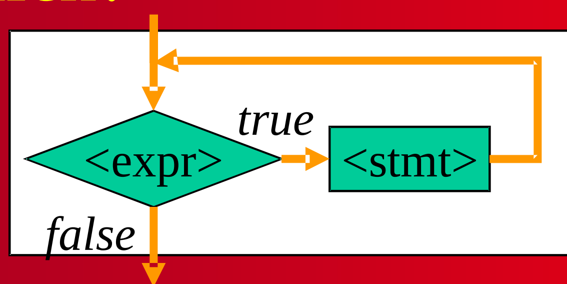
<while-stmt> ::= while (<expr>) <stmt>

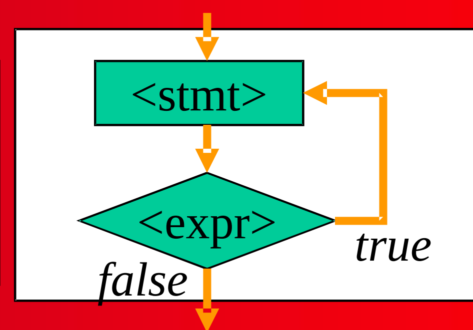
but the *do* loop is a *test-at-the-top* loop:

<do-stmt> ::= do <stmt> while (<expr>) ;

Which is which?

```
while (<expr>)
   <stmt>
```






```
do
   <stmt>
while (<expr>);
```

A pretest loop's <stmt> is executed 0+ times (zero-trip behavior).

A posttest loop's <stmt> is executed 1+ times (one-trip behavior).

Counting Loops

Like most languages, C++ provides a *for* loop for counting:

<for-stmt> ::= for (<opt-expr> ; <opt-expr> ; <opt-expr>) <stmt>

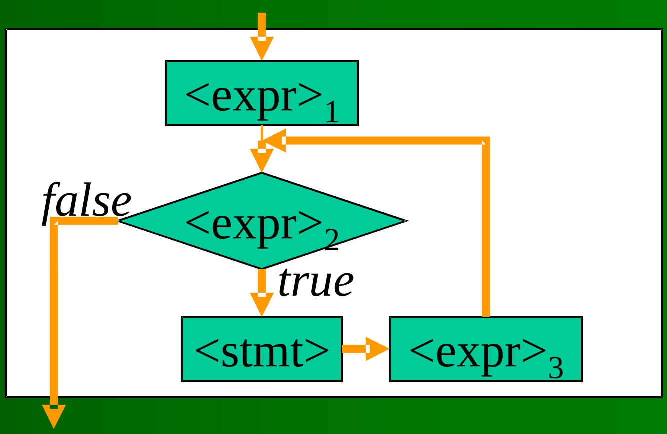
This provides unusual flexibility for an imperative language:

```
for (int i = 0; i <= 100; i++)
   <stmt> // do <stmt> 101 times; i = ________________

for (double d = -0.5; d <= 0.5; d += 0.1)
   <stmt> // do <stmt> 11 times; d = ________________

for (Node * ptr = myHead; ptr != myTail; ptr = ptr->next)
   <stmt>  // do <stmt> ____________________________
```

In most languages, the counting loop is a *pretest* loop:



Unrestricted Loops

Most modern languages also support an unrestricted loop.

- Such loops have no fixed exit point.
- All of the C/C++ loops can be made to behave this way.

```
for (;;)          while (true)     do
   <stmt>            <stmt>             <stmt>
                                   while (true);
```

- The language usually provides a statement to exit such loops.
- Unrestricted loops can be structured as test-at-the-top, test-at-the-bottom, or test-in-the-middle loops:

```
for (;;) {
  if (<expr>) break;
  <stmt>2
}
```

```
for (;;) {
  <stmt>1
  if (<expr>) break;
}
```

```
for (;;) {
  <stmt>1
  if (<expr>) break;
  <stmt>2
}
```

- <stmt>$_1$ executes ___ times; <stmt>$_2$ executes ____ times...

Ada

Ada provides *pretest*, *counting*, and *unrestricted* loops:

```
for i in 1..100 loop
  …
end loop;
```

```
for i in reverse 1..100 loop
  …
end loop;
```

```
while i <= 100 loop
  …
  i:= i+1;
end loop;
```

```
loop
  exit when i > 100;
  …
  i:= i+1;
end loop;
```

Exercise: How would you build a BNF for Ada's loops?

<Ada-loop-stmt>::= <loop-prefix> loop <loop-stmt-list> end loop
<loop-prefix> ::= <for-prefix> | <while-prefix> | ε
<while-prefix> ::= <while> <condition>
<for-prefix> ::= for id in <range-clause>
<range-clause> ::= <scalar> .. <scalar> | reverse <scalar> .. <scalar>
<loop-stmt> ::= <stmt> | <exit-when-clause>
<exit-when-clause> ::= exit when <condition>

What if you need a post-test loop, or to count by i != 1?

Smalltalk

Smalltalk provides 2 pretest and 3 counting loop-messages:

<loop-expr> ::= <while-expr> | <times-expr> | <to-expr>
<while-expr> ::= <block> <while-msg> <block>
<while-msg> ::= whileTrue: | whileFalse:
<times-expr> ::= <intExpr> timesRepeat: <block>
<to-expr> ::= <numExpr> to: <numExpr> <opt-by> do: <block>
<opt-by> ::= by: <numExpr> | ε

```
[i <= 100] whileTrue:
[   …
  i:= i+1
]
```

```
[i > 100] whileFalse:
[   …
  i:= i+1
]
```

```
100 timesRepeat:
[   …
]
```

```
0 to: 100 do:
[   …
]
```

```
-0.5 to: 0.5 by: 0.1 do:
[   …
]
```

Under what circumstances should a given loop be used?

Lisp

Lisp has no loop functions, because anything that can be done by repetition can also be done using *recursion*.

```
(defun f(n)
   …
   (f(+ n 1))
)
```

```
(defun factorial(n)
  (if (< n 2)
     1
     (* n (factorial (- n 1))))
)
```

Recursive functions can provide test-at-the-top, test-at-the-bottom, and test-in-the-middle behavior simply by varying The placement of the recursive call and the if controlling it:

```
(defun f(n)
  (if (< n max)
    (f(+ n 1))
  <expr-list>)
)
```

```
(defun g(n)
  <expr-list>
  (if (< n max)
    (f(+ n 1)))
)
```

```
(defun h(n)
  <expr-list>
  (if (< n max)
    (f(+ n 1))
  <expr-list>)
)
```


Summary

There are three basic control structures:

- Sequence
- Selection
- Repetition

Different kinds of languages accomplish these differently:

- Sequence is the default mode of control provided by the *block* construct of most languages (progn in Lisp).
- Selection is accomplished via:
  - Statements (e.g., *if*, *switch* or *case*) controlled by boolean expressions in imperative languages
  - Functions (e.g., *if* and *cond* in Lisp) with boolean arguments in functional languages
  - Messages (*ifTrue:*, *ifFalse:*, … in Smalltalk) sent to boolean objects in pure OO languages

Summary (ii)

- Repetition is accomplished via:
  - Statements (e.g., *while*, *do*, *for*) controlled by boolean expressions in imperative languages
  - Recursive functions in functional languages
  - Messages (*whileTrue:*, *timesRepeat:*, *to:by:do:*, … in Smalltalk) sent to boolean (or numeric) objects in pure OO languages

These 3 control structures and I/O are all we need to compute anything that can be computed (i.e., by a Turing machine).

Most of the other language constructs simply make the task of programming such computations *more convenient*.