Control Structures

Programming Languages

CS 214

Why was the little ink-drop sad?

Because his mother was in the pen, doing a long sentence!

Spaghetti Coding

In the early 1960s, _____ 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 _____ does not indicate _____…

→ Such code was _____ ...

Control Structures

In 1968, _____ published _____ a letter suggested the ___ should be outlawed because it encouraged _____ (the letter raised a furor).

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

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

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

Structured Programming

Structured Programming emphasized _____, 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;
}
```

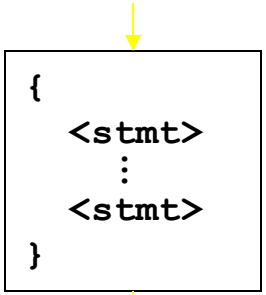
With structured programming, _____!

The resulting programs were _____.

Sequential Execution

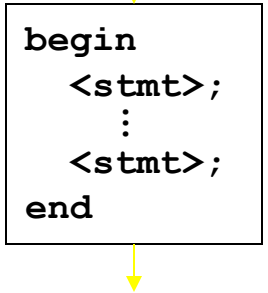
A C/C++ block has 0 or more statements:

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



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



The block is the control structure for _____,
the default control structure in imperative languages.

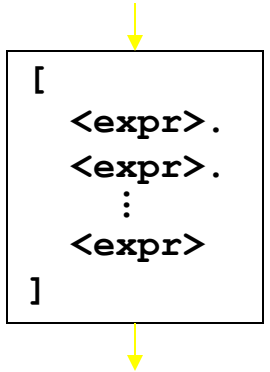
The guiding principle for control structures is:

_____.

Smalltalk

Smalltalk also has a _____ construct, but it is an _____:

<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



Smalltalk computations consist of _____ sent to _____:

[2 + 1] value → 3

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

[: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 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)
       (/ (* (+ n 1) n) 2))
```


Lisp (ii)

Some Lisp function-arguments must be a single expression.

Lisp's _____ function can be used to execute several expressions sequentially, much like other languages' ____:

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

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

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

Note: Clojure names this function __ instead of _____.

What happened when the semicolon broke the grammar laws?

It got back-to-back sentences.

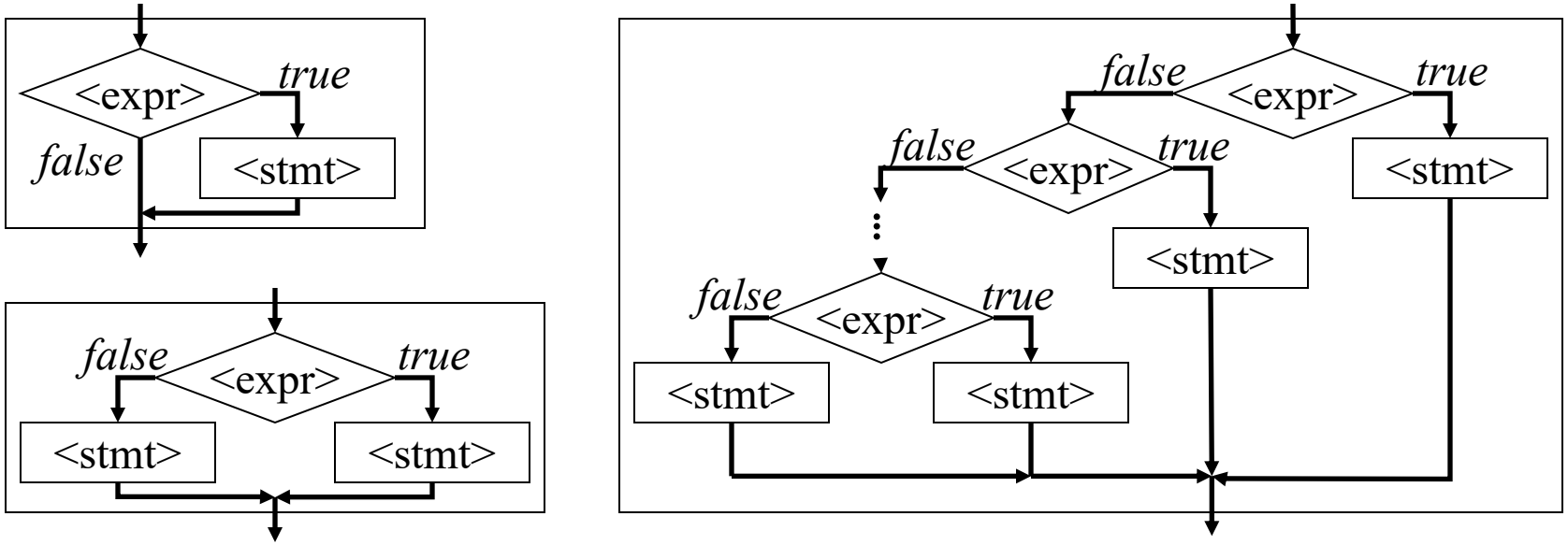
Selective Execution

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

The _____ 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:

:

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

:

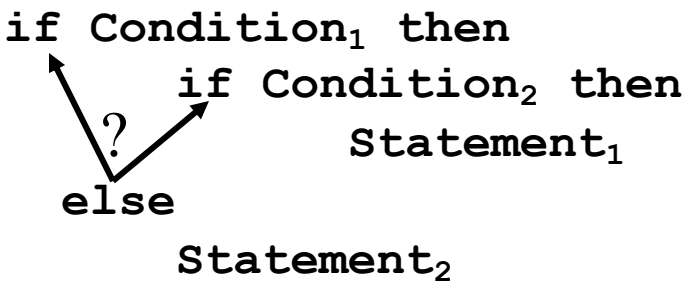
```
if (first < second)
   min = first;
else
   min = second;
```

:

```
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 _____ following nested if statements...



The problem occurs in languages with _____.

→ Such a statement can be _____ in two different ways.

There are two different approaches to resolving the question:

- Add a _____ to resolve the ambiguity; vs.
- Design a statement whose _____.

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
```

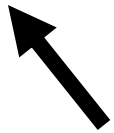
```
if ( Condition1 )
      if ( Condition2 )
            StmtList1
      end if
else
      StmtList2
end if
```

Ada, Fortran-90, 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 __ to be a ___, any nested __ 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 ___.

Aesthetics

Multibranch selection can get clumsy using _____:

```
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 ___ keyword that, substituted for _____, extends the same __ statement.

Modula-2 and Ada replaced the error-prone ____ with _____.

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> ::= _____

_____ ::= _____

_____ ::= _____

Specify an Ada if Stmt

<Ada_if_stmt> ::= if <condition> then <stmts> <elsif_part> <else_part> end if

<elsif_part> ::= elsif <condition> then <stmts> <elsif_part> | ε

<else_part> ::= else <stmts> | ε

<stmts> ::= <stmt> ; <more_stmts>

<more_stmts> ::= <stmt> ; <more_stmts> | ε

Lisp's *if*

Lisp provides an ___ function as one if its expressions:

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

Semantics: If the _____ evaluates to non-___ (i.e., not ()),
the _____ is evaluated and its value returned;
else the _____ 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 _____: and _____: 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]
```

These four are the only selection messages Smalltalk provides.

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


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 not uniform:

- The _____ executes after __ comparison.
- The _____ execute after __ comparisons.

The time to execute successive branches increases _____.

Santa and Mrs. Claus grew apart and so decided to get a divorce. Living at the North Pole, there was no local divorce court, so they decided to use a semicolon instead, because…

… semicolons are great for separating independent clauses.

For our last Halloween party, I wanted to dress up as a stork.

My wife said she didn’t want to be seen with me dressed like that, so...

… I had to put my foot down.

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 not uniform:

- The _____ executes after __ comparison.
- The _____ execute after __ comparisons.

The time to execute successive branches increases _____.

The Switch Statement

The _____ statement provides _____.

<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:

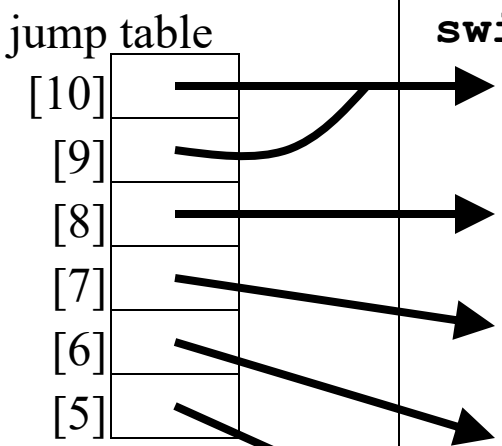
Note: If you neglect to supply _____ statements, control by default flows sequentially through the _____ statement.

The _____ is a _____ 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 _____ statements achieve uniform response time via a _____, 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 _____ and _____ to find the branch.

When a multibranch *if* does _____, a *switch* is probably faster.

A compiler spends _____ and space (to build the jump table) to decrease the average _____ needed to find a branch.

The Case Statement

The _____ is a descendent of the ____ statement (Algol-W).

Only _____ languages use the _____ syntax.

Unlike the *switch*, a *case* statement _____ behavior; no _____ is needed!

Most *case* stmts also let you use literal _____ and _____:

Ada uses the _____ 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 ____ statement.

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

<Ada-case> ::= _____

Specify an Ada case Stmt

<Ada_case> ::= case <scalar_expr> of <branch> <branches> <opt_others> end case

<branch> ::= when <cases> => <stmts>

<cases> ::= <case> <more_cases>

<case> ::= <literal> | <literal> .. <literal>

<more_cases> ::= , <case> <more_cases> | ε

<branches> ::= <branch> <branches> | ε

<opt_others> ::= when others => <stmts> | ε

Python 2 switch statement

```
grade = 8
{
    10: lambda: print("A");
     9: lambda: print("A");
     8: lambda: print("B");
     7: lambda: print("C");
     6: lambda: print("D");
     5: lambda: print("E");
}.get(
    grade,
    lambda: print("F")
)()
```

This is what a python 3 switch statement looks like behind the scenes

Lisp

Lisp provides a _____ function that looks similar to a _____.

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

However Lisp's _____ 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 ____ cannot employ a jump table, so it has the same _____ as an *if*.

The predicates are evaluated _____ until a true _____ is found; its _____ is then evaluated.

Clojure

Clojure modernizes Lisp by adding a _____ function:

<case-expr> ::= (case <expr-pairs> <default_expr>)
<expr-pairs> ::= <expr> <expr> <expr-pairs> | ε

Unlike Lisp's ____, Clojure's ____ provides _____ for all cases, but it builds a _____, not a jump table.

```
(case (quot score 10)
  (9 10) "A"
       8 "B"
       7 "C"
       6 "D"
         "F"
)
```

This allows _____ (strings, reals, etc.) unlike other languages' case or switch statements.

If there are no matches and no <default_expr> is provided, _____ throws a run-time exception.

When is a dog’s tail not a tail?

When it’s a-waggin’!

Repetition

A third control structure is _____, or _____.

The C-family _____ loop is a _____, or _____ loop:

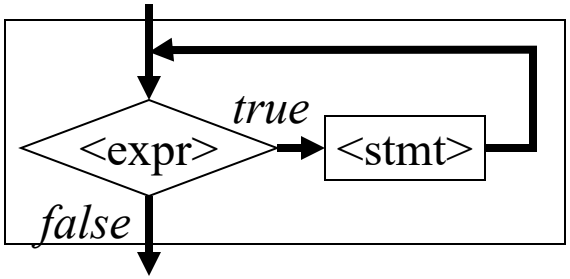
<while-stmt> ::= while (<expr>) <stmt>

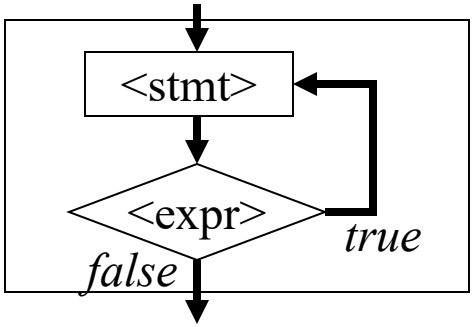
but the __ loop is a _____, or _____ loop:

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

Which is which?

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






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

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

A _____ 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>

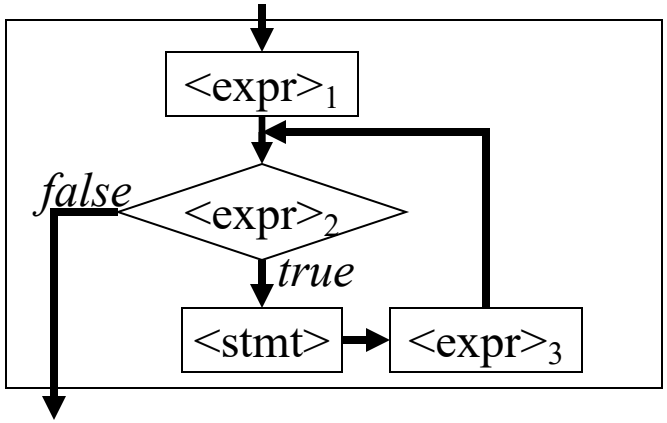
This provides unusual flexibility for an imperative language:

```
for (int i = 0; i <= 100; i++)
  <stmt> // do <stmt> 101 times; i = 0, 1, 2, 3, …, 100

for (double d = -0.5; d <= 0.5; d += 0.1)
  <stmt> // do <stmt> 11 times; d = -.5, -.4, -.3, …, .5

for (Node * ptr = myHead; ptr != nullptr; ptr = ptr->next)
  <stmt>  // do <stmt> once for each node in the list
```

In most languages, the counting loop is a _____ loop:



Unrestricted Loops

Most modern languages also support an _____.

- –Such loops have _____.
- –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
}
```

- – $\langle stmt \rangle_1$ executes 1+ times; $\langle stmt \rangle_2$ executes 0+ times...

Ada

Ada provides _____, _____, and _____ 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-stmt>

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 _____.

```
(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 _____:

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

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

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


What’s blue and doesn’t weigh much?

Light blue!

Summary

There are three basic control structures:

- _____
- _____
- _____

Different kinds of languages accomplish these differently:

- _____ is the default mode of control provided by the _____ construct of most languages (Lisp ____; Clojure __).
- _____ is accomplished via:
  - _____ (e.g., __, _____ or _____) controlled by _____ in imperative languages
  - _____ (e.g., __ and ____ in Lisp) with _____ (aka _____) in functional languages
  - _____ (_____:, _____:, … in Smalltalk) sent to _____ in pure OO languages

Summary (ii)

- _____ is accomplished via:
  - _____ (e.g., _____, __, ___) controlled by _____ in imperative languages
  - _____ in functional languages
  - _____ (_____:, _____:, _____:, … 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 _____.