# COP4020 Programming Languages

## Control Flow

*Robert van Engelen & Chris Lacher*

# Overview

- Structured and unstructured flow
  - Goto's
  - Sequencing
  - Selection
  - Iteration and iterators
  - Recursion
  - Nondeterminacy
- Expressions evaluation
  - Evaluation order
  - Assignments

# Control Flow: Ordering the Execution of a Program

- Constructs for specifying the execution order:
  1. *Sequencing*: the execution of statements and evaluation of expressions is usually in the order in which they appear in a program text
  2. *Selection* (or alternation): a run-time condition determines the choice among two or more statements or expressions
  3. *Iteration*: a statement is repeated a number of times or until a run-time condition is met
  4. *Procedural abstraction*: subroutines encapsulate collections of statements and subroutine calls can be treated as single statements
  5. *Recursion*: subroutines which call themselves directly or indirectly to solve a problem, where the problem is typically defined in terms of simpler versions of itself
  6. *Concurrency*: two or more program fragments executed in parallel, either on separate processors or interleaved on a single processor
  7. *Nondeterminacy*: the execution order among alternative constructs is deliberately left unspecified, indicating that any alternative will lead to a correct result

# Structured and Unstructuted Flow

- *Unstructured flow*: the use of *goto statements* and *statement labels* to implement control flow
  - Merit or evil?
  - Generally considered bad, but sometimes useful for jumping out of nested loops and for coding the flow of exceptions (when a language does not support exception handling)
  - Java has no goto statement (supports labeled loops and breaks)
- *Structured flow*:
  - Statement sequencing
  - Selection with “if-then-else” statements and “switch” statements
  - Iteration with “for” and “while” loop statements
  - Subroutine calls (including recursion)
  - All of which promotes “structured programming”

# Sequencing

- A list of statements in a program text is executed in top-down order
- A *compound statement* is a delimited list of statements
  - A compund statement is called a *block* when it includes variable declarations
  - C, C++, and Java use { and } to delimit a block
  - Pascal and Modula use **`begin ... end`**
  - Ada uses **`declare ... begin ... end`**
- Special cases: in C, C++, and Java expressions can be inserted as statements
- In pure functional languages sequencing is impossible (and not desired!)

# Selection

- If-then-else selection statements in C and C++:
  - **`if`** (<expr>) <stmt> [**`else`** <stmt>]
  - Condition is a bool, integer, or pointer
  - Grouping with { and } is required for statement sequences in the *then clause* and *else clause*
  - Syntax ambiguity is resolved with "*an else matches the closest if*" rule
- Conditional expressions, e.g. **`if`** and **`cond`** in Lisp and **`a?b:c`** in C
- Java syntax is like C/C++, but condition must be Boolean
- Ada syntax supports multiple **`elsif`**'s to define nested conditions:
  - **`if`** <cond> **`then`**
    <statements>
    **`elsif`** <cond> **`then`**
    ...
    **`else`**
    <statements>
    **`end if`**

# Selection (cont'd)

- Case/switch statements are different from if-then-else statements in that an expression can be tested against multiple constants to select statement(s) in one of the arms of the case statement:
  - C, C++, and Java:
    **`switch`** (<expr>)
    { **`case`** <const>: <statements> **`break`**;
      **`case`** <const>: <statements> **`break`**;
      ...
      **`default`**: <statements>
    }
  - A **`break`** is necessary to transfer control at the end of an *arm* to the end of the switch statement
  - Most programming languages support a switch-like statement, but do not require the use of a break in each arm
- A switch statement is much more efficient compared to nested if-then-else statements

# Iteration

- *Enumeration-controlled loops* repeat a collection of statements a number of times, where in each iteration a loop index variable takes the next value of a set of values specified at the beginning of the loop
- *Logically-controlled loops* repeat a collection of statements until some Boolean condition changes value in the loop
  - *Pretest loops* test condition at the begin of each iteration
  - *Posttest loops* test condition at the end of each iteration
  - *Midtest loops* allow structured exits from within loop with exit conditions

# Enumeration-Controlled Loops

- History of failures on design of enumeration-controlled loops
- Fortran-IV:

```
      DO 20 i = 1, 10, 2
      ...
20    CONTINUE
```

which is defined to be equivalent to

```
      i = 1
20    ...
      i = i + 2
      IF i.LE.10 GOTO 20
```

Problems:

- Requires positive constant loop bounds (1 and 10) and step size (2)
- If loop index variable i is modified in the loop body, the number of iterations is changed compared to the iterations set by the loop bounds
- GOTOs can jump out of the loop and also from outside into the loop
- The value of counter `i` after the loop is implementation dependent
- The body of the loop will be executed at least once (no empty bounds)

# Enumeration-Controlled Loops (cont'd)

- Fortran-77:
  - Same syntax as in Fortran-IV, but many dialects support **`ENDDO`** instead of **`CONTINUE`** statements
  - Can jump out of the loop, but cannot jump from outside into the loop
  - Assignments to counter **`i`** in loop body are not allowed
  - Number of iterations is determined by
    $$\max(\lfloor (H-L+S)/S \rfloor, 0)$$
    for lower bound $L$, upper bound $H$, step size $S$
  - Body is not executed when $(H-L+S)/S < 0$
  - Either integer-valued or real-valued expressions for loop bounds and step sizes
  - Changes to the variables used in the bounds do not affect the number of iterations executed
  - Terminal value of loop index variable is the most recent value assigned, which is
    $$L + S*\max(\lfloor (H-L+S)/S \rfloor, 0)$$

# Enumeration-Controlled Loops (cont'd)

- Algol-60 combines logical conditions in *combination loops*:

  **`for`** <id> := <forlist> **`do`** <stmt>

  where the syntax of <forlist> is

  <forlist> ::= <enumerator> [, enumerator]*
  <enumerator> ::= <expr>
  | <expr> **`step`** <expr> **`until`** <expr>
  | <expr> **`while`** <cond>

- Not orthogonal: many forms that behave the same:

```
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 ...
```

# Enumeration-Controlled Loops (cont'd)

- Pascal's enumeration-controlled loops have simple and elegant design with two forms for up and down:

  **`for`** <id> := <expr> **`to`** <expr> **`do`** <stmt>

  and

  **`for`** <id> := <expr> **`downto`** <expr> **`do`** <stmt>

- Can iterate over any discrete type, e.g. integers, chars, elements of a set
- Lower and upper bound expressions are evaluated once to determine the iteration range
- Counter variable cannot be assigned in the loop body
- Final value of loop counter after the loop is undefined

# Enumeration-Controlled Loops (cont'd)

- Ada's for loop is much like Pascal's:

```
for <id> in <expr> .. <expr> loop
  <statements>
end loop
```

and

```
for <id> in reverse <expr> .. <expr> loop
  <statements>
end loop
```

- Lower and upper bound expressions are evaluated once to determine the iteration range
- Counter variable has a local scope in the loop body
  - Not accessible outside of the loop
- Counter variable cannot be assigned in the loop body

# Enumeration-Controlled Loops (cont'd)

- C, C++, and Java do not have true enumeration-controlled loops
- A "for" loop is essentially a logically-controlled loop

```
for (i = 1; i <= n; i++) ...
```

  which iterates **`i`** from 1 to **`n`** by testing **`i <= n`** before the start of each iteration and updating **`i`** by 1 in each iteration
- Why is this not enumeration controlled?
  - Assignments to counter **`i`** and variables in the bounds are allowed, thus it is the programmer's responsibility to structure the loop to mimic enumeration loops
- Use **`continue`** to jump to next iteration
- Use **`break`** to exit loop
- C++ and Java also support local scoping for counter variable

```
for (int i = 1; i <= n; i++) ...
```

# Enumeration-Controlled Loops (cont'd)

- Other problems with C/C++ for loops to emulate enumeration-controlled loops are related to the mishandling of bounds and limits of value representations
  - This C program never terminates (do you see why?)

```
#include <limits.h> // INT_MAX is max int value
main()
{ int i;
  for (i = 0; i <= INT_MAX; i++)
    printf("Iteration %d\n", i);
}
```

  - This C program does not count from 0.0 to 10.0, why?

```
main()
{ float n;
  for (n = 0.0; n <= 10; n += 0.01)
    printf("Iteration %g\n", n);
}
```

# Enumeration-Controlled Loops (cont'd)

- How is loop iteration counter overflow handled?
- C, C++, and Java: nope
- Fortran-77
  - Calculate the number of iterations in advance
  - For **`REAL`** typed index variables an exception is raised when overflow occurs
- Pascal and Ada
  - Only specify step size 1 and -1 and detection of the end of the iterations is safe
  - Pascal's final counter value is undefined (may have wrapped)

# Iterators

- *Iterators* are used to iterate over elements of containers such as sets and data structures such as lists and trees
- Iterator objects are also called *enumerators* or *generators*
- C++ iterators are associated with a container object and used in loops similar to pointers and pointer arithmetic:

```
vector<int> V;
…
for (vector<int>::iterator it = V.begin(); it != V.end(); ++it)
      cout << *n << endl;
```

An in-order tree traversal:

```
tree_node<int> T;
…
for (tree_node<int>::iterator it = T.begin(); it != T.end(); ++it)
      cout << *n << endl;
```

# Iterators (cont'd)

- Java supports *generics* similar to C++ *templates*
- Iterators are similar to C++, but do not have the usual C++ overloaded iterator operators:

```
TreeNode<Integer> T;
…
for (Integer i : T)
     System.out.println(i);
```

Note that Java has the above special for-loop for iterators that is essentially syntactic sugar for:

```
for (Iterator<Integer> it = T.iterator(); it.hasNext(); )
{    Integer i = it.next();
     System.out.println(i);
}
```

# Iterators (cont'd)

- Iterators typically need special loops to produce elements one by one, e.g. in Clu:

```
for i in int$from_to_by(first, last, step) do
    …
end
```

- While Java and C++ use *iterator objects* that hold the state of the iterator, Clu, Python, Ruby, and C# use *generators* (="true iterators") which are functions that run in "parallel" to the loop code to produce elements
  - The *yield* operation in Clu returns control to the loop body
  - The loop returns control to the generator's last yield operation to allow it to compute the value for the next iteration
  - The loop terminates when the generator function returns

# Logically-Controlled Pretest loops

- *Logically-controlled pretest loops* check the exit condition before the next loop iteration
- Not available Fortran-77
- Ada has only one kind of logically-controlled loops: midtest loops
- Pascal:

  **`while`** <cond> **`do`** <stmt>

  where the condition is a Boolean-typed expression
- C, C++:

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

  where the loop terminates when the condition evaluates to 0, NULL, or false
  - Use **`continue`** and **`break`** to jump to next iteration or exit the loop
- Java is similar C++, but condition is restricted to Boolean

# Logically-Controlled Posttest Loops

- *Logically-controlled posttest loops* check the exit condition after each loop iteration
- Not available in Fortran-77
- Ada has only one kind of logically-controlled loops: midtest loops
- Pascal:

  **`repeat`** <stmt> [; <stmt>]* **`until`** <cond>

  where the condition is a Boolean-typed expression and the loop terminates when the condition is true
- C, C++:

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

  where the loop terminates when the expression evaluates to 0, NULL, or false
- Java is similar to C++, but condition is restricted to Boolean

# Logically-Controlled Midtest Loops

- Ada supports *logically-controlled midtest loops* check exit conditions anywhere within the loop:

```
loop
   <statements>
exit when <cond>;
   <statements>
exit when <cond>;
   ...
end loop
```

- Ada also supports labels, allowing exit of outer loops without gotos:

```
outer: loop
   ...
   for i in 1..n loop
      ...
      exit outer when a[i]>0;
      ...
   end loop;
end outer loop;
```

# Recursion

- Recursion: subroutines that call themselves directly or indirectly (mutual recursion)
- Typically used to solve a problem that is defined in terms of simpler versions, for example:
  - To compute the length of a list, remove the first element, calculate the length of the remaining list in *n*, and return *n*+1
  - Termination condition: if the list is empty, return 0
- Iteration and recursion are equally powerful in theoretical sense
  - Iteration can be expressed by recursion and vice versa
- Recursion is more elegant to use to solve a problem that is naturally recursively defined, such as a tree traversal algorithm
- Recursion can be less efficient, but most compilers for functional languages are often able to replace it with iterations

# Tail-Recursive Functions

- *Tail-recursive functions* are functions in which no operations follow the recursive call(s) in the function, thus the function returns immediately after the recursive call:

*tail-recursive*

```
int trfun()
{ …
  return trfun();
}
```

*not tail-recursive*

```
int rfun()
{ …
  return rfun()+1;
}
```

- A tail-recursive call could *reuse* the subroutine's frame on the run-time stack, since the current subroutine state is no longer needed
  - Simply eliminating the push (and pop) of the next frame will do
- In addition, we can do more for *tail-recursion optimization*: the compiler replaces tail-recursive calls by jumps to the beginning of the function

# Tail-Recursion Optimization

- Consider the GCD function:

```
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);
}
```

a good compiler will optimize the function into:

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

which is just as efficient as the iterative version:

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

# Converting Recursive Functions to Tail-Recursive Functions

- Remove the work after the recursive call and include it in some other form as a computation that is passed to the recursive call
- For example, the non-tail-recursive function

```
(define summation (lambda (f low high)
  (if (= low high)
      (f low)
      (+ (f low) (summation f (+ low 1) high)))))
```

can be rewritten into a tail-recursive function:

```
(define summation (lambda (f low high subtotal)
  (if (=low high)
      (+ subtotal (f low))
      (summation f (+ low 1) high (+ subtotal (f low))))))
```

# Example

- Here is the same example in C:

```
typedef int (*int_func)(int);
int summation(int_func f, int low, int high)
{ if (low == high)
    return f(low)
  else
    return f(low) + summation(f, low+1, high);
}
```

rewritten into the tail-recursive form:

```
int summation(int_func f, int low, int high, int subtotal)
{ if (low == high)
    return subtotal+f(low)
  else
    return summation(f, low+1, high, subtotal+f(low));
}
```

# When Recursion is Bad

- The Fibonacci function implemented as a recursive function is very inefficient as it takes exponential time to compute:

```
(define fib (lambda (n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2)))))))
```

with a tail-recursive helper function, we can run it in O(n) time:

```
(define fib (lambda (n)
  (letrec ((fib-helper (lambda (f1 f2 i)
                         (if (= i n)
                             f2
                             (fib-helper f2 (+ f1 f2) (+ i 1))))))
    (fib-helper 0 1 0))))
```

# Expression Syntax and Effect on Evaluation Order

- An expression consists of
  - An atomic object, e.g. number or variable
  - An operator applied to a collection of operands (or arguments) that are expressions
- Common syntactic forms for operators:
  - Function call notation, e.g. **`somefunc(A, B, C)`**
  - *Infix* notation for binary operators, e.g. A + B
  - *Prefix* notation for unary operators, e.g. -A
  - *Postfix* notation for unary operators, e.g. i++
  - *Cambridge Polish* notation, e.g. (* (+ 1 3) 2) in Lisp
  - "*Multi-word*" infix, e.g. **`a>b?a:b`** in C and
    **`myBox displayOn: myScreen at: 100@50`**
    in Smalltalk, where **`displayOn:`** and **`at:`** are written infix with arguments **`mybox`**, **`myScreen`**, and **`100@50`**

# Operator Precedence and Associativity

- The use of infix, prefix, and postfix notation sometimes lead to ambiguity as to what is an operand of what
  - Fortran example: **`a+b*c**d**e/f`**
- *Operator precedence*: higher operator precedence means that a (collection of) operator(s) group more tightly in an expression than operators of lower precedence
- *Operator associativity*: determines evaluation order of operators of the same precedence
  - *Left associative*: operators are evaluated left-to-right (most common)
  - *Right associative*: operators are evaluated right-to-left (Fortran power operator **, C assignment operator = and unary minus)
  - *Non-associative*: requires parenthesis when composed (Ada power operator **)

# Operator Precedence and Associativity

- Pascal's flat precedence levels is a design mistake

```
if A<B and C<D then
```

  is the same as

```
if A<(B and C)<D then
```

- Note: levels of operator precedence and associativity are easily captured in a grammar as we saw earlier

# Evaluation Order of Expressions

- Precedence and associativity state the rules for structuring expressions, but do not determine the operand evaluation order!
  - Expression

    `a-f(b)-b*c`

    is structured as

    `(a-f(b))-(b*c)`

    but either `(a-f(b))` or `(b*c)` can be evaluated first
- The evaluation order of arguments in function and subroutine calls may differ, e.g. arguments evaluated from left to right or right to left
- Knowing the operand evaluation order is important
  - Side effects: suppose `f(b)` above modifies the value of `b` (`f(b)` has a "side effect") then the value will depend on the operand evaluation order
  - Code improvement: compilers rearrange expressions to maximize efficiency, e.g. a compiler can improve memory load efficiency by moving loads up in the instruction stream

# Expression Operand Reordering Issues

- Rearranging expressions may lead to arithmetic overflow or different floating point results
  - Assume **`b`**, **`d`**, and **`c`** are very large positive integers, then if **`b-c+d`** is rearranged into **`(b+d)-c`** arithmetic overflow occurs
  - Floating point value of **`b-c+d`** may differ from **`b+d-c`**
  - Most programming languages will not rearrange expressions when parenthesis are used, e.g. write **`(b-c)+d`** to avoid problems
- Design choices:
  - Java: expressions evaluation is always left to right in the order operands are provided in the source text and overflow is always detected
  - Pascal: expression evaluation is unspecified and overflows are always detected
  - C anc C++: expression evaluation is unspecified and overflow detection is implementation dependent
  - Lisp: no limit on number representation

# Short-Circuit Evaluation

- *Short-circuit evaluation* of Boolean expressions: the result of an operator can be determined from the evaluation of just one operand
- Pascal does not use short-circuit evaluation
  - The program fragment below has the problem that element **`a[11]`** is read resulting in a dynamic semantic error:

```
var a:array [1..10] of integer;
...
i := 1;
while i<=10 and a[i]<>0 do
  i := i+1
```

- C, C++, and Java use short-circuit conditional and/or operators
  - If **`a`** in **`a&&b`** evaluates to false, **`b`** is not evaluated
  - If **`a`** in **`a||b`** evaluates to true, **`b`** is not evaluated
  - Avoids the Pascal problem, e.g.

```
while (i <= 10 && a[i] != 0) ...
```

  - Ada uses **`and then`** and **`or else`**, e.g. cond1 **`and then`** cond2
  - Ada, C, and C++ also have regular bit-wise Boolean operators

# Assignments and Expressions

- Fundamental difference between imperative and functional languages
- Imperative: "computing by means of side effects"
  - Computation is an ordered series of changes to values of variables in memory (state) and statement ordering is influenced by run-time testing values of variables
- Expressions in functional language are *referentially transparent*:
  - All values used and produced depend on the local referencing environment of the expression
  - A function is *idempotent* in a functional language: it always returns the same value given the same arguments because of the absence of side-effects

# L-Values vs. R-Values and Value Model vs. Reference Model

- Consider the assignment of the form: *a* := *b*
  - The left-hand side *a* of the assignment is an *l-value* which is an expression that should denote a location, e.g. array element a[2] or a variable foo or a dereferenced pointer *p
  - The right-hand side *b* of the assignment is an *r-value* which can be any syntactically valid expression with a type that is compatible to the left-hand side
- Languages that adopt the *value model* of variables copy the value of *b* into the location of *a* (e.g. Ada, Pascal, C)
- Languages that adopt the *reference model* of variables copy references, resulting in shared data values via multiple references
  - Clu copies the reference of *b* into *a* so that *a* and *b* refer to the same object
  - Java is a mix: it uses the value model for built-in types and the reference model for class instances

# Special Cases of Assignments

- Assignment by *variable initialization*
  - Use of *uninitialized variable* is source of many problems, sometimes compilers are able to detect this but with programmer involvement e.g. *definite assignment* requirement in Java
  - Implicit initialization, e.g. 0 or NaN (not a number) is assigned by default when variable is declared
- Combinations of *assignment operators*
  - In C/C++ **`a+=b`** is equivalent to **`a=a+b`** (but **`a[i++]+=b`** is different from **`a[i++]=a[i++]+b`**, ouch!)
  - Compiler produces better code, because the address of a variable is only calculated once
- *Multiway assignments* in Clu, ML, and Perl
  - **`a,b := c,d`** assigns **`c`** to **`a`** and **`d`** to **`b`** simultaneously, e.g. **`a,b := b,a`** swaps **`a`** with **`b`**
  - **`a,b := 1`** assigns 1 to both **`a`** and **`b`**