CSCI: 4500/6500 Programming Languages

Control Flow

Chapter 6

Big Picture: Control Flow
Ordering in Program Execution

Ordering/Flow Mechanisms:

- Sequencing (statements executed (evaluated) in a specified order)
  - Imperative language - very important
  - Functional - doesn't matter as much (emphasizes evaluation of expression, de emphasize or eliminates statements, e.g., pure fl don't have assignment statements)
- Selection -- Choice among two or more
  - Deemphasized in logical languages
- Iteration
  - » Repeating structure
    - emphasized in imperative languages
- Procedural abstraction, recursion, requires stack
- Concurrency
  - » 2 or more code fragments executed at the same time
- Non-determinacy (unspecified order)

Expression Evaluation: Classification Outline

- Infix, Prefix or Postfix
- Precedence & Associativity
- Side effects
- Statement versus Expression Oriented Languages
- Value and Reference Model for Variables
- Orthogonality
- Initialization
- Aggregates
- Assignment

Evaluation: * fix operators

- Expression:
  - » Operator (built-in function) and operands (arguments)
- Infix, prefix, postfix operators
  - » (+ 5 5) or 5 + 6
  - » operators in many languages are just `syntactic sugar'[1] for a function call:
    - a + b ⇒ a.operator+(b) in C++
    - "+"(a, b) in Ada
  - » Cambridge Polish prefix and function name *inside* parenthesis.
  - » Postfix - postscript, Forth input languages, calculators

[1] Landin "adding "sugar" to a language to make it easier to read (for humans)

Expression Evaluation: Precedence & Associativity

How should this be evaluated?

- a + b * c**d**e / f

How about?

- ((((a + b) * c) ** d) ** e) / f
- a + (((b * c) ** d) ** (e / f))
- a + ((b * (c ** (d ** e))) / f)
  - » Fortran does this last option
- or something else?

Precedence & Associativity

- Precedence specify that some operators group more tightly and others
  - » Richness of rules across languages varies (overview next slide)
- Associativity rules specify that sequences of operators of *equal* precedence groups either left or right (or up or down? for a weird language of your own creation)
  - » Associatively rules are somewhat uniform across languages but there are variations

<table>
<tr><th>Fortran</th><th>Pascal</th><th>C</th><th>Ada</th></tr>
<tr><td></td><td></td><td>++, -- (post-inc., dec.)</td><td></td></tr>
<tr><td>**</td><td>not</td><td>++, -- (pre-inc., dec.),
+, - (unary),
&, * (address, contents of),
!, ~ (logical, bit-wise not)</td><td>abs (absolute value),
not, **</td></tr>
<tr><td>*, /</td><td>*, /,
div, mod, and</td><td>* (binary), /,
% (modulo division)</td><td>*, /, mod, rem</td></tr>
<tr><td>+, - (unary
and binary)</td><td>+, - (unary and
binary), or</td><td>+, - (binary)</td><td>+, - (unary)</td></tr>
<tr><td></td><td></td><td><<, >>
(left and right bit shift)</td><td>+, - (binary),
& (concatenation)</td></tr>
<tr><td>.eq., .ne., .lt.,
.le., .gt., .ge.
(comparisons)</td><td><, <=, >, >=,
=, <>, IN</td><td><, <=, >, >=
(inequality tests)</td><td>=, /= , <, <=, >, >=</td></tr>
<tr><td>.not.</td><td></td><td>==, != (equality tests)</td><td></td></tr>
<tr><td></td><td></td><td>& (bit-wise and)</td><td></td></tr>
<tr><td></td><td></td><td>^ (bit-wise exclusive or)</td><td></td></tr>
<tr><td></td><td></td><td>| (bit-wise inclusive or)</td><td></td></tr>
<tr><td>.and.</td><td></td><td>&& (logical and)</td><td>and, or, xor
(logical operators)</td></tr>
<tr><td>.or.</td><td></td><td>|| (logical or)</td><td></td></tr>
<tr><td>.eqv., .neqv.
(logical comparisons)</td><td></td><td>?: (if ... then ... else)</td><td></td></tr>
<tr><td></td><td></td><td>=, +=, -=, *=, /=, %=,
>>=, <<=, &=, ^=, |=
(assignment)</td><td></td></tr>
<tr><td></td><td></td><td>, (sequencing)</td><td></td></tr>
</table>

Figure 6.1: **Operator precedence levels in Fortran, P** operators at the top of the figure group most tightly.

Example Precedence:

- if A < B and C < D then K = 5

How would Pascal evaluate this?

- A < (B and C) < D

Precedence

- Most languages avoid this problem by giving arithmetic operators higher precedence than relational (comparison) operators.

Precedence: Rule of Thumb

- C has 15 levels - too many to remember
- Pascal has 3 levels - too few for good semantics
- Fortran has 8
- Ada has 6
  - » Note: Ada puts *and*, *or* at same level
- Lesson: when unsure (e.g., programmer using many languages, better to circumvent precedence and use parentheses!

Associativity Example

- Basic operators almost always left to right
  - » 9-3-2 = (9-3)-2 = 4 (left right)
  - » 9-3-2 = 9-(3-2) = 8 (right-left)
- Exponential operator: **
  - » right-left (as do mathematics) in Fortran
    - – 4**3**2 = 4**(3**2) = 262,144
  - » Language syntax requires parenthesized form (Ada)
- Assignment '=' in expressions associates: right-left
  - » a = b = a + c => a = (b = (a+c))
    - – assigns (a+c) to b then assigns the same value to a

Side Effects & Idempotent Functions

Side Effects – a function has a side effect if it influences subsequent computation in any way other than by returning a value. A side effect occurs when a function changes the environment in which it exists

Idempotent – an idempotent function is one that if called again with the same parameters, will always give the same result

Referentially transparent - Expressions in a purely functional language are referentially transparent, their value depends only on the referencing environment.

Imperative programming – "Programming with side effects" (programming in terms of statements, state).

Side Effects

- Assignment statements provide the ultimate example of side effects
  - they change the value of a variable
  - Fundamental in the von Neumann model of computation.
- Several languages *outlaw side effects* for functions (these languages are called *single assignment* languages)
  - » easier to prove things about programs
  - » closer to Mathematical intuition
  - » easier to optimize
  - » (often) easier to understand
- But side effects can be nice: consider - `rand()`
  - Needs to have a side effect, or else the same random number every time it is called.

Side Effects (cont)

- Side effects are a particular problem if they affect state used *in other parts* of the expression in which a function call appears:
- Example:
  - » `a - f(b) - c * d  /* f(b) may affect 'd' */`
  - » What is evaluated first:

```
a – f(b)
c * d
```


Ordering within Expressions

- **Another Example:**
  - » f(a, g(b), c) which parameter is evaluated first?
- **Why is it important:**
  - » Side-effects:
    - if g(b) modifies a or c then the values passed into f will depend on the order that parameters are evaluated
  - » Code improvements:
    - a = B[i]
    - c = a * b + d * 3
      - Note: precedence or associativity does not say if we evaluated a*b or d*3 first.
    - Evaluate: d*3 first, so the previous load (slow) of B[i] from memory occurs in parallel of a doing something different, i.e. computing d*3.

Evaluation of Operands and Side Effects

```
int x = 0;

int foo()
    {
    x += 5;
    return x;
    }

  int a = foo() + x + foo();
```

What is the value of a?

a = 5 + x + foo()

Re-ordering using mathematical properties

- Commutative
  - » (a+b) = (b+a)
- Associative
  - » (a+b) + c = a + (b + c)
- Distributive
  - » a * (b + c) = a * b + a * c.

Mathematical Identities

Example:

a = b + c

d = c + e + b

Re-order to:

a = b + c

d = b + c + e (already evaluated b+c (it is a))

Mathematical Identities

- **Problem:** Computer has limited precision
  - » associativity (known to be dangerous)
    `(a + b) + c`
    works if `a~=maxint` and `b~=minint` and `c<0`
    `a + (b + c)` does not

Expression vs. Statement Orientation

- **Statements :**
  - » executed solely for their side effects and
  - » return no useful value
  - » most imperative languages
  - » time dependent
- **Expressions :**
  - » may or may not have side effects
  - » always produces a value and
  - » functional languages (Lisp, Scheme, ML)
  - » time less
- C kinda halfway in-between (distinguishes)
  - » allows expression to appear instead of statement

Assignment

- statement (or expression) *executed for its side effect*
- assignment operators (`+=`, `-=`, etc)
  - » handy
  - » avoid redundant work (or need for optimization)
    - – No need for redundant address calculations (guaranteed)
  - » perform side effects exactly once (avoids pecularities)
    - – A[f(i)] = A[f(i)] + 1 (f(i) may have a side effect x 2).
    - – A[f(i)] += 1

References and Values

- Assignment seems straightforward
- Semantic differences depending if languages uses a
  - » A reference model
  - » or value model of variables.
- Impact on programs that use pointers (we will talk more about pointers after Spring break).

Value Model

a [4]

- *Variable is a named container for a value*
- left-hand side of expressions denote "locations" and are referenced as l-values
- right-hand side of expressions denote "values" and are referred to as r-values
- Expressions can be either an l-value or an r-value depending on context:
  - » `2 + 3 = a`
  - » `a = 2 + 3`
  - » `( f(x)+3 ) -> b[c] = 2 /* l-value expression */`
  - » `k = (f(x)+3)->b[c]`

```
Example languages who use
value model: C and C++
```


Value Model: Example

a [4]

a [4]

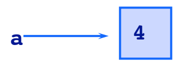
b [2]

c [2]

```
b = 2
c = b
a = b + c
```

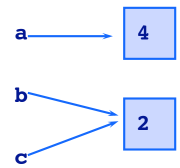
1. Put the value 2 in b
2. Copy value of b into c
3. Read b and c and put result in a

Reference Model



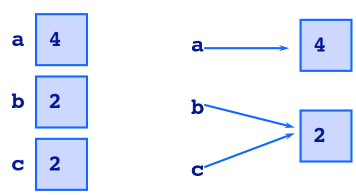
- *Variable is a named reference to a value*
- Every "value" is a l-value (location)
  - » Only one '4', variables points to the '4', Above the variable a points to '4'
- To get a "value" (r-value) need to dereference it to obtain value that it contain (points to).
  - » Most languages this dereferencing is automatic, e.g., Clue. In ML need to explicit dereference it.
  - » Indirection for accesses (however most compiler use multiple copies of objects to speed things up).

Reference Model




```
b = 2
c = b
a = b + c
```

1. Let b refer to 2
2. Let c also refer to 2
3. Pass these references to '+'
4. Let a refer to the result, namely 4




```
b = 2
c = b
a = b + c
```

- *Value model: any integer value can contain the value 2*
- *Reference model: only one 2 (if variable on right side, need to dereference to get actual value).*

Value/Variable Model Implications

- Reference model need to distinguish between variables that
  - » refer to the same object and variables that
  - » point to different objects but that have the same "value" (happens to be equal)
- LISP provided two notions of equality to distinguish between the two.

Value versus Reference Models

- Value-oriented languages
  - » C, Pascal, Ada
- Reference-oriented languages
  - » most functional languages (Lisp, Scheme, ML)
  - » Clu, Smalltalk
- Algol-68 kinda halfway in-between
- Java deliberately in-between, uses both:
  - » Value model for built-in types (int, double)
  - » Reference model for user-defined types (objects)
- C# and Eiffel allow programmer choose model for user defined types.

Orthogonality

- Features that can be used in any combination (no redundancy)
  - » Meaning makes sense
  - » Meaning is consistent

```
if (if b != 0 then a/b == c else false) then ...
if (if f then true else messy()) then ...
```

- Algol makes orthogonality a principal design goal.

Initialization

Motivation:

- Improve execution time: Statically allocated variables (by compiler)
  - » e.g. reduce cost of assignment statement at run time.
- Avoid (weird) errors of evaluating variables with no initial value

Approach:

- Pascal has no initialization facility (assign)
- C/C++ initializes static variables to 0 by default
- Usage of non-initialized variables may cause a hardware interrupt (implemented by "initializing" value to NaN)
- Constructor: automatic initialization at run-time

Control Flow

(Really)

Structured vs. Unstructured Control Flow

Structured Programming – hot programming trend in the 1970's

- Top down design
- Modularization of code
- Structured types
- Descriptive variable names
- Extensive commenting
- After Algol 60, most languages had: if…then…else, while loops

Don't need to use `goto's` …

Types of Control Flow

- Sequencing -- statements executed (evaluated) in a specified order
  - » Imperative language - very important
  - » Functional - doesn't matter as much (emphasizes evaluation of expression, de emphasize or eliminates statements, e.g. assignment statements)
- Selection -- Choice among two or more
  - – Deemphasized in logical languages
- Iteration -- Repeating structure
  - – emphasized in imperative languages
- Procedural abstraction
- Recursion, requires stack
- Concurrency executing statements at the same time
- Non-determinacy -- unspecified order

Sequencing

- Simple idea
  - » Statements executes one after another
  - » Very imperative, von-Neuman
  - » Controls order in which side effects occur
- Statement blocks
  - » groups multiple statement together into one statement
  - » Examples:
    - – {} in C, C++ and Java
    - – begin/end in Algol, Pascal and Modula
- Basic block
  - » Block where the only control flow allowed is sequencing

Selection

`if` statements

- `if` *condition* `then` *statement* `else` *statement*
  - Nested if statements have a dangling else problem

Dangling `else` Problem

```
if … then
    if … then
    else …
```

- or

```
if … then
    if then …
else …
```

- Which one does the else map to?

Dangling `else` Problem

- ALGOL:
  - does not allow "then if"
  - statement has to be different than another `if` statement (can be another block, that contains an `if`)
- Pascal:
  - `else` associates with closest unmatched `then`
- Perl:
  - Has a separate `elsif` keyword (in addition to `else` and `if`)
  - "`else if`" will cause an error

Strict vs short-circuit evaluation of conditions

- strict
  - Evaluate all operands before applying operators
    - Pascal
- short-circuit
  - Skip operand evaluation when possible
  - Evaluation order important
    - if operand-evaluation has side effects (seen)
    - if programmer knows that some operands can be computed more quickly than others
  - Examples
    - || and && in C++ and Java
      - always use short-circuit evaluation
    - then if and or else in Ada
      - language supports both strict and short-circuit, programmer decides: use *`and, or`* for strict evaluation

Short Circuiting

- C++

```
p = my_list;
while( p && p->key != val )
      p = p->next;
```

- Pascal

```
p := my_list;
while( p <> nil) and ( p^.key <> val ) do
      p := p^.next
```

Ouch!

Short Circuit Jump Code

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

- Usually purpose of condition is to *create a branch instruction* to various locations not a value to be stored.

No Short Circuiting (Pascal)

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

```
     r1 := A        -- load
     r2 := B
     r1 := r1 > r2
     r2 := C
     r3 := D
     r2 := r2 > r3
     r1 := r1 & r2
     r2 := E
     r3 := F
     r2 := r2 $<>$ r3
     r1 := r1 $|$ r2
     if r1 = 0 goto L2
L1: then_clause   -- label not actually used
    goto L3
L2: else_clause
L3:
```

- root would name r1 as the register containing the expression value

Short Circuiting

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

```
    r1 := A
    r2 := B
    if r1 <= r2 goto L4
    r1 := C
    r2 := D
    if r1 > r2 goto L1
L4: r1 := E
    r2 := F
    if r1 = r2 goto L2
L1: then_clause
    goto L3
L2: else_clause
L3:
```

- Inherited attributes of the conditions root would indicate that control should "fall through" to L1 if the condition is true, or branch to L2 if false.
- Value of 'final' expression never in a register rather its value is implicit in the control flow.

Implications

- Short-circuiting
  - » Can avoid out of bound errors
  - » Can lead to more efficient code
  - » Not all code is guaranteed to be evaluated
- Strict
  - » Not good when code has build in side effects

Case/Switch Statements

- Alternative to nested if…then…else blocks

```
j := … (* potentially complicated expression *)
IF j = 1 THEN clause_A
ELSEIF j IN 2,7 THEN clause_B
ELSEIF j IN 3..5 THEN clause_C
ELSEIF (j = 10) THEN clause_D
ELSE clause_E
END
```

```
CASE … (* potentially complicated expression *) of
         1:      clause_A
      | 2, 7:     clause B
      | 3..5:    clause C
      | 10:           clause D
         ELSE    clause E
END
```

Principal motivation of case statement is to generate efficient target code not syntactic elegance.

Implementation of Case Statements

- If…then…else

```
    r1 := ...
    if r1 <> 1 goto L1
    clause_A
    goto L6
L1: if r1 = 2 goto L2
    if r1 <> 7 goto L3
L2: clause_B
    goto L6
L3: if r1 < 3 goto L4
    if r1 > 5 goto L4
    clause_C
    goto L6
L4: if r1 <> 10 goto L5
    clause_D
    goto L6
L5: clause_E
L6:
```

- Case (uses jump table)

```
T:  &L1          -- tested expression = 1
    &L2
    &L3
    &L3
    &L3
    &L5
    &L2
    &L5
    &L5
    &L4          -- tested expression = 10
L6: r1 := ...    -- calculate tested expression
    if r1 < 1 goto L5
    if r1 > 10 goto L5     -- L5 is the "else" arm
    r1 -:= 1
    r2 := T[r1]
    goto *r2
L7:
```


Case vs Switch

- `Switch` is in C, C++, and Java
  - » Unique syntax
  - » Use `break` statements, otherwise statements fall through to the next case
- `Case` is used in most other languages
  - » Can have ranges and lists
  - » Some languages do not have `default` clauses
    - – Pascal

Origin of Case Statements

- Descended from the computed `goto` of Fortran

```
goto (15, 100, 150, 200), J

if J is 1, then it jumps to label 15
if J is 4, then it jumps to label 200
if J is not 1, 2, 3, or 4, then the
  statement does         nothing
```


Iteration

Iteration

- More prevalent in imperative languages
- Takes the form of loops
  - » Iteration of loops used for their side effects
    - – Modification of variables

Iteration

Two (2) kinds of iterative loops:

- enumeration controlled: Executed once for every value in a given finite set (iterations known before iteration begins)
- logically-controlled: Executed until some condition changes value

Enumeration-Controlled Loop

- Early Fortran:

```
     do 10 i = 1, 50, 2
       . . .
  10: continue
```

- Equivalent?

```
  10: i = 1
      . . .
      i = i + 2
      if i <= 50 goto 10
```


Issue #1

- Can the step size/bounds be:
  - » Positive/negative ?
  - » An expression ?
  - » Of type Real ?

Issue #2

- Changes to loop indices or bounds
  - » Prohibited to varying degrees
  - » Algol 68, Pascal, Ada, Fortran 77/90
    - – Prohibit changes to the index within loop
    - – Evaluate bound once (1) before iteration

Changes to loop indices or bounds

- A statement is said to threaten an index variable if
  - » Assigns to it
  - » Passes it to a subroutine
  - » Reads it from a file
  - » Is a structure that contains a statement that threatens it

Issue #3

- Test terminating condition before first iteration
- Example:

```
for i := first to last by step do
        …
end
```

```
   r1  := first
   r2  := step
   r3  := last
L1:  if r1 > r3 goto L2
     …
   r1  := r1 + r2
   goto L1
L2
```

```
   r1  := first
   r2  := step
   r3  := last
L1:   …
   r1  := r1 + r2
   goto L1
L2: if r1 < r3 goto L1
```


Issue #4

- Access to index outside loop
  - » undefined
    - – Fortran IV, Pascal
  - » most recent value
    - – Fortran 77, Algol 60
  - » index is a local variable of loop
    - – Algol 68, Ada

Issue #5

- Jumps
  - » Restrictions on entering loop from outside
    - – Algol 60 and Fortran 77 and most of their descendents prevent the use of gotos to jump into a loop.
  - » “exit” or “continue” used for loop escape

Summary Issues

- step: size (pos/neg), expression, type
- changes to indices or bounds within loop
- test termination condition before first iteration of loop
- scope of control variable (access outside loop)
  - » value of index after the loop

Logically Controlled Loops

```
while condition do statement
```

- Advantages of `for loop` over `while loop`
  - » Compactness
  - » Clarity
  - » All code affecting flow control is localized in header

Logically Controlled Loops

- Where to test termination condition?
  - » pre-test (while)
  - » post-test (repeat)
  - » mid-test (when)
    - – one-and-a-half loops (loop with exit, mid-test)

```
loop:
   statement list
when condition exit
   statement list
when condition exit
end loop
```


C's for loop

- C's for loop
  - » Logically controlled
    - – Any enumeration-controlled loop can be written as a logically-controlled loop

```
for( i = first; i <= last; I += step )
      {
      }
```

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


C's for loop

- Places additional responsibility on the programmer
  - » Effect of overflow on testing of termination condition
  - » Index and variable in termination condition can be changed
    - – By body of loop
    - – By subroutines the loop calls

Combination Loops

- Combination of enumeration and logically controlled loops
- Algol 60's for loop

For_stmt -> for id := for_list do stmt

*For_list -> enumerator (, enumerator)**

Enumerator -> expr

-> expr step expr until expr

-> expr while condition

Algol 60's for loop

- Examples: (all equivalent)

```
for i := 1, 3, 7, 9 do…
for i := 1 step 2 until 10 do …
for i := 1, i + 2 while i < 10 do …
```

- Problems
  - » Repeated evaluation of bounds
  - » Hard to understand

Iterators: HW - Read in Textbook

- True Iterators
- Iterator Objects
- Iterating with first-class functions
- Iterating without iterators

Recursion

Recursive Computation

- Decompose problem into smaller problems by calling itself
- Base case- when the function does not call itself any longer; no base case, no return value
- Problem must always get smaller and approach the base case

Recursive Computation

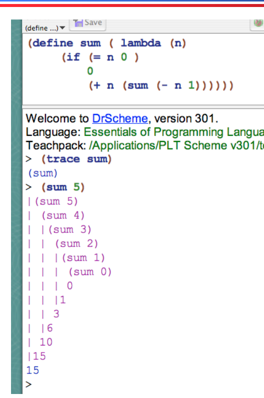
- No side effects
- Requires no special syntax
- Can be implemented in most programming languages; need to permit functions to call themselves or other functions that call them in return.
- Some languages don't permit recursion: Fortran 77

Tracing a Recursive Function

```
(define sum (lambda(n)
  (if (= n 0)
      0
      (+ n (sum (- n 1))))))
```


Tracing a Recursive Function

```
>(trace sum)
#<unspecified> >
>(sum 5)
"CALLED" sum 5
 "CALLED" sum 4
  "CALLED" sum 3
   "CALLED" sum 2
    "CALLED" sum 1
     "CALLED" sum 0
     "RETURNED" sum 0
    "RETURNED" sum 1
   "RETURNED" sum 3
  "RETURNED" sum 6
 "RETURNED" sum 10
"RETURNED" sum 15
15
```



Embedded vs. Tail Recursion

Analogy: You've been asked to measure the distance between UGA and Georgia Tech

Embedded:

1. Check to see if you're there yet
2. If not, take a step, put a mark on a piece of paper to keep count, restart the problem
3. When you're there, count up all the marks

Tail:

1. Write down how many steps you're taken so far as a running total
2. When you get to Georgia Tech, the answer is already there; no counting!

Recursion

- Tail recursion: No computation follows recursive call

```
/* assume a, b > 0 */
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);
  }
```


Which is Better?

- Tail.
- Additional computation never follows a recursive call; the return value is simply whatever the recursive call returns
- The compiler can reuse space belonging to the current iteration when it makes the recursive call
- Dynamically allocated stack space is unnecessary

- Any logically controlled iterative algorithm can be rewritten as a recursive algorithm and vice versa
- Iteration: repeated modification of variables (imperative languages)
  - » Uses a repetition structure(`for`, `while`)
  - » Terminates when loop continuation condition fails
- Recursion: does not change variables (functional languages)
  - » Uses a selection structure (`if`, `if/else`, or `switch/case`)
  - » Terminates when a base case is recognized

Tail Recursion Example

```
/* assume a, b > 0 */
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);
  }
```

```
/* assume a, b > 0 */
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 tail recursive?

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

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

Last one: Note that it passes along an accumulator.

Recursion

- equally powerful to iteration
- mechanical transformations back and forth
- often more intuitive (sometimes less)
- *naïve* implementation less efficient
  - » no special syntax required
  - » fundamental to functional languages like Scheme

Expression Evaluation: Short Circuiting

- Consider `(a < b) && (b < c)`:
  - » If `a >= b` there is no point evaluating whether `b < c` because `(a < b) && (b < c)` is automatically false
- Other similar situations

```
if (b != 0 && a/b == c) ...
if (*p && p->foo) ...
if (f || messy()) ...
```