CS 326
Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Control Flow

Language Specification

- General issues in the design and implementation of a language:
  - Syntax and semantics
  - Naming, scopes and bindings
  - Control flow
  - Data types
  - Subroutines

Control Flow

- Establishes the order in which instructions are executed
- Mechanisms for specifying control flow:
  - Sequencing - execution usually corresponds to the order of statements in the program
  - Selection - depending on a run-time condition, a choice is made among several statements or expressions
  - Iteration - a fragment of code is executed repeatedly, either a certain number of times, or until a run-time condition is satisfied
  - Procedural abstraction - a fragment of code is encapsulated so that it can be treated as a unit, subject to parameterization

Control Flow

- Mechanisms for specifying control flow (cont.):
  - Recursion - an expression is defined in terms of (simpler versions of) itself
  - Concurrency - several code fragments are executed "at the same time", either in parallel on separate processors, or interleaved on the same processor
  - Nondeterminacy - the ordering is deliberately left unspecified (any order is correct)

Control Flow

- These mechanisms are found in most programming languages
- Their relative importance varies considerably:

- Imperative languages
  - sequencing is essential (emphasis on assigning values to variables)
  - heavy use of iteration
- Functional languages
  - sequencing has minor role (emphasis on evaluation of expressions that return a value, without side-effects)
  - heavy use of recursion
- Logic languages
  - hide the issue of control flow entirely

Sequencing

- In Scheme:

```
(define (f x)
  (+ x 1)
  (+ x 2))
```

- What is the effect of the (+ x 1) expression?
  - None, it has no side effects and the function returns the value of (+ x 2)

```
(define (f x)
  (display x)
  (+ x 2))
```

- What is the effect of the (display x) expression?
  - Now it matters, as (display x) has the side effect of displaying x

- In general, sequencing is relevant only when side effects are present

Sequencing

- Why is it good to have no side effects?

- Ensure that functions are idempotent:
  - Can call a function repeatedly with same parameters → same result
  - The moment when a function is called does not affect the surrounding code

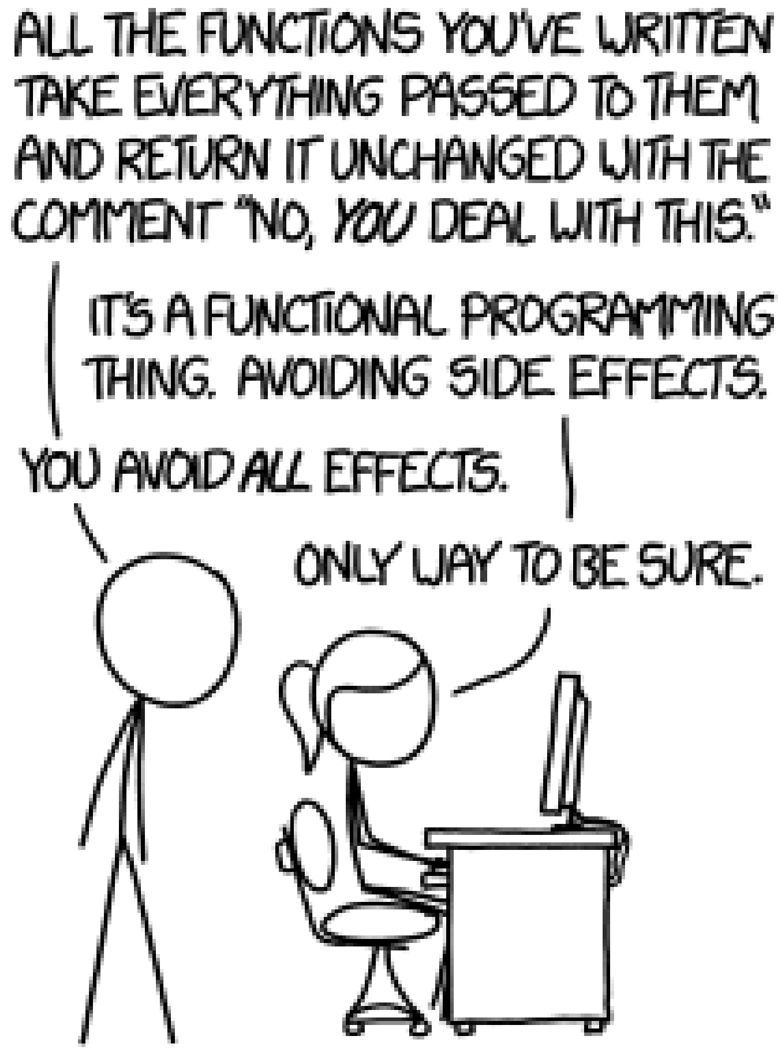
Are these equivalent?

```
a = f(b);        c = d;
c = d;           a = f(b);
```

No, if f changes d

- Easier to check the program correctness
- Easier code improvement – safe rearrangement of expressions

Side Effects



Sequencing

- In imperative languages:

- List of statements – introduced by begin...end or {...} delimiters
- Such a list:
  - is called a compound statement (block)
  - defines a local scope for variables declared in the block
  - may have a return value – last statement in list

- Still desire for lack of side-effects:
  - Euclid and Turing – functions are not allowed side-effects, only procedures can
  - Ada – functions can change global variables, but not their own parameters

Expressions and Statements

- Expressions – generate a value
- Statements – just have side effects (through assignments)

- Statement-oriented languages
  - Distinguish between statements and expressions
  - Statements are the building blocks of programs
  - Variables are assigned values generated by expressions
  - What a statement returns is not important (not defined)
  - Examples: most imperative languages
- Expression-oriented languages
  - No distinction, everything is an expression and must return a value
  - Examples: Algol 68, functional languages (Lisp, Scheme, ML)

- C is somewhere in between

Expressions and Statements

- Example in Algol-68:

```
begin
    a := if b < c then d else e;
    x := begin f(y); g(z) end;
    g(d);
    p := q := r;
    2 + 3;
end
```

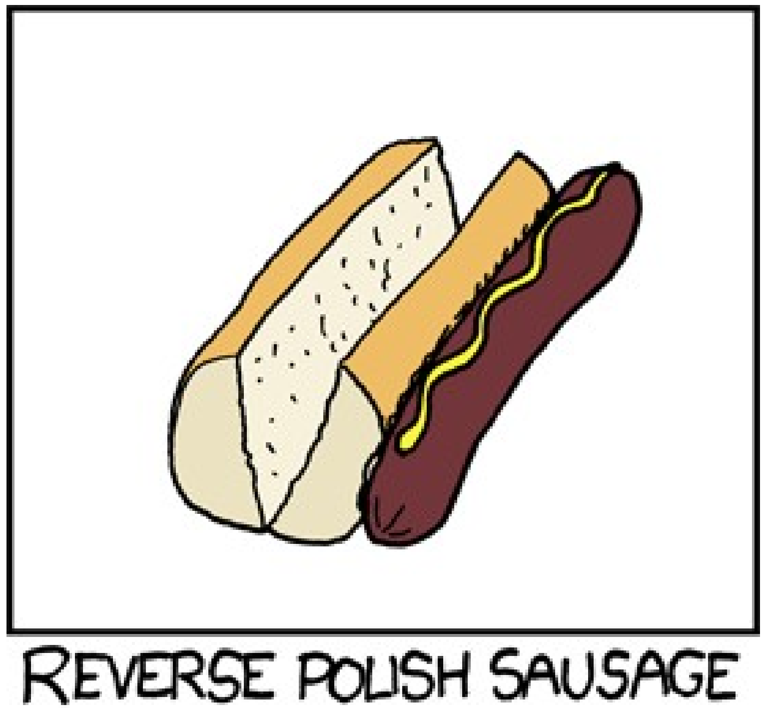
- Value of the if...then...else expression – the then part or the else part, depending on the condition
  - a is assigned d or e
- Value of the begin...end expression – the last expression in sequence
  - x is assigned g(z)
  - the value returned by f(y) is ignored
  - the value returned by g(d) is ignored
  - the outer begin...end expression returns 5
- Value of the assignment – the value that is assigned
  - the assignment q := r returns r, which is assigned to p

Expression Evaluation

- Expression
  - either a simple object (constant or variable) or an operator applied on a collection of operands, each of which being an expression
  - it results in a value
- Operators can be +, -, *, /, etc or function names
- Operands can be simple objects (constants or variables), or expressions

- Expressions in "pure" form (in "pure" functional languages) have no side effects - referential transparency
- Notation
  - prefix
  - infix
  - postfix

RPS



Expression Evaluation

- Algol-family languages (Pascal, C) – distinguish between function calls and built-in operators

    my_func (a, b)   - prefix notation for function calls

    a + b   - infix notation for binary operators

    -c - prefix notation for unary operators

    a := if b <> 0 then c else d   - in Algol, if...then...else is a three- operand infix operator

- Lisp-family languages –no distinctions, Cambridge Polish notation everywhere (prefix, with parentheses around expression)

    (my_func a b)

    (+ a b)

    (- c)

- C++ – define operators as shorthand notations for function calls

    a + b   ⇔a.operator+(b)

- Postscript, Forth – use mostly postfix notation

Precedence and Associativity

- How is the following expression evaluated?

  9-3*2+1

- Precedence - rules to specify that some operators group "more tightly" than others, in the absence of parentheses (* and / have higher precedence that + and -)
- Associativity - rules to specify whether sequences of operators of equal precedence group to the right or left (arithmetic operators have left associativity)

- These issues arise only for infix notation, not for prefix or postfix

Precedence and Associativity

- Precedence:
  - C – 15 levels of precedence - too many to remember
    - Does + have higher precedence than >> ?
  - Pascal – 3 levels of precedence - too few for good semantics
    - Precedence is not specified for some operators

      if A < B and C < D then (* ouch *)
    - This condition evaluates to ((A < B) and C) < D
  - APL – all operators have equal precedence, parentheses must be used
- Associativity:
  - In general, associativity is towards left. When is it towards right?
    - In assignments: a = b = c + d

Assignments

- What is a variable?
  - It actually depends on the context in which it appears
- In C, what is the meaning of a in the following:

```
d = a;          // the value of a
a = b + c;      // the location (address) of a
```

- Expressions that denote locations → l-values
- Expressions that denote values → r-values

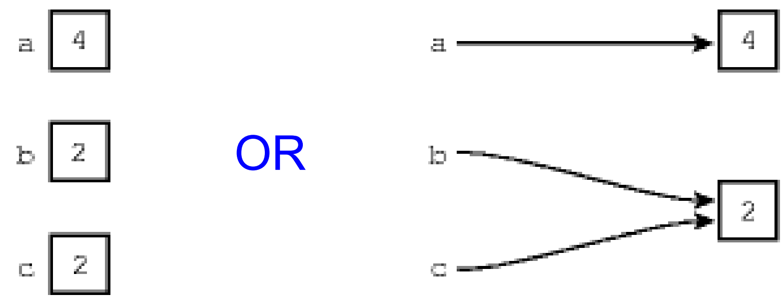
a	has both r-value and l-value
5+1	only r-value
NULL	only r-value (it is a constant)
b[i]	both
p->s	both
f(a)	has r-value, unless it doesn't return anything (void)
	may have l-value, if it returns an address (pointer)

Assignments

- Consider the code:

```
b := 2;
c := b;
a := b + c;
```

- Implementation:



- Value model (left)
  - Variable – a named container for a value
  - Examples: C, Pascal, Ada
- Reference model (right)
  - Variable - a named reference (pointer) to a value
  - Examples: Lisp, Scheme, ML (functional languages), Clu, Smalltalk
  - Important to distinguish between:
    - variables that refer to the same object (eq?)
    - variables that refer to different objects, with same values (equal?)

Assignment vs. Initialization

- What is the difference between:

```
int a = 1;          AND     int a;
                            a = 1;
```

  - In the first case, if a is statically allocated, the value 1 is placed into location a at compile time
  - In the second case, the assignment will incur execution cost at run-time

- What is the difference between:

```
void f (int x)              AND         void f (int x)
{                                       {
  int a = 1;                              int a;
}                                         a = 1;
                                        }
```

  - No difference, the value 1 is placed into a at runtime anyway, as the memory location for a is determined at run-time (on the stack)

Assignment vs. Initialization

- Why is initialization also useful?
  - To prevent using variables before they are assigned a value -> initialize them when are first declared

- Can initialization be done automatically (with some default value) when the programmer doesn't do it?
  - Statically allocated variables – in C they are guaranteed to be filled with zero bits, if not otherwise initialized
  - Dynamically allocated variables (on stack or heap) – expensive to initialize automatically, because it cannot be done at compile time

Assignment vs. Initialization

- Can the use of uninitialized variables be detected at run-time (dynamic semantic error)?

- Uninitialized floating-point variables
  - can be filled with a NaN ("not a number") value
  - any use of NaN in computation → signal error

- For other types, where all possible values are legitimate
  - need to use extra space for a flag
  - too expensive

Assignment vs. Initialization

- Particular interest in C++

```
class T
{
   char * s;
   int  N;                          // number of elements
   ...
};
```

- For an object containing (as a data member) a variable length array:
  - assignment – the assignment operator must generally deallocate the old space and allocate new space
  - initialization – the constructor must simply allocate space

Combination Assignment Operators

- Update a variable:

  a = a + 1;

  b.c[3].d = b.c[3].d * e;

- Why is this not desirable?
  - Hard to write and to read
  - Redundant address calculations
  - Address calculation may have side effects:

    a[f(x)] = a[f(x)] + 3;

    If f has side effects (don't want them twice), need to rewrite as:

    j = f(x);

    a[j] = a[j] + 3;

Combination Assignment Operators

- To update a variable - use combination assignment operators (in C):

```
a += 1;
b.c[3].d *= e;
```

- C also provides prefix and postfix increment and decrement operations:

```
int i = 5;
A[++i] = b;        // i becomes 6, A[6] is assigned value b
int k = 5;
A[k--] = b;        // A[5] is assigned value b, k becomes 4
```

- Pointer arithmetic in C:

```
int * p;
...
p += 3;            // actually changes p to point 3*sizeof(int) bytes higher
                   in memory
```

Ordering within Expressions

- Precedence and associativity
  - Define the order in which operators are applied
  - Do not specify the order of evaluating operands

```
a - f(b) - c*d          // is f(b) or c*d evaluated first?
g (a, f(b), c*d)        // is f(b) or c*d evaluated first?
```

- Why is evaluation order important?
  - Side effects – f may modify the values of c or d
  - Code improvement:

```
a*b + f(c)        // a*b would need a register to save the result
                  // better to evaluate f(c) first, so that f has all registers available
```

Ordering within Expressions

- Importance of code improvement → most languages impose no order of evaluation
  - the compiler can choose whatever ordering produces faster code
  - exception: Java - always left-to-right evaluation
- Rearranging order of expressions (in Fortran):

a = b + c d = c + e + b	Produce code equivalent to:	a = b + c d = a + e

- Can generate problems – computer arithmetic:
  - If b, c, d are all positive and close to the maximum value that can be represented:

```
b - c + d        // OK
b + d - c        // arithmetic overflow
```

Short-Circuit Evaluation

- Short-circuit evaluation
  - used in the evaluation of Boolean expressions
  - evaluate only as much as needed to compute the value of an expression
- Examples:

```
if (b != 0 && a/b == c) ...
if (p && p->foo) ...
if (i >= 0 && i < N_MAX && A[i] > x) ...
```

- C – short-circuit evaluation
- Pascal – only regular evaluation of Boolean expressions
- Clu – both:

and and or	- regular evaluation
cand and cor	- short-circuit evaluation

Structured and Unstructured Flow

- Control flow in assembly language - conditional and unconditional jumps
- Early imperative languages (Fortran, Cobol, PL/I) - mimic this approach:

```
        if A .lt. B goto 10
        ...
10: ...
```

- Late 1960s, 1970s - debate on merits and evils of goto
  - Dijkstra article: "GOTO Statement Considered Harmful"
  - Rubin article: "'GOTO Considered Harmful' Considered Harmful"
- Ada – allows goto in limited contexts
- Fortran 90, C++ – allows goto just for compatibility
- Java – does not allow it, but keeps goto as a keyword

Structured and Unstructured Flow

- Abandonment of goto → apparition of structured programming
  - instead of labels and jumps - lexically nested blocks
  - selection through if...then...else
  - iteration through for, while
- Remaining cases when goto would be useful:
- Mid-loop exit

```
        while true do
        begin
                    readln (line);
                    if all_blanks (line) then goto 100;
                    consume_line (line)
        end;
     100: ...
```

- C provides break to do this

Structured and Unstructured Flow

- Mid-loop continue

```
while not eof do
begin
   readln (line);
   if all_blanks (line) then goto 100;
   consume_line (line)
      100:
end;
```

- C provides continue to do this

Structured and Unstructured Flow

- Early return from subroutines

```
procedure consume_line (var line: string);
...
begin
   ...
   (*if the rest of line is a comment, ignore it *)
   if line[i] = '%' then goto 100;
   ...
      100:
end;
```

- C provides return to do this

Structured and Unstructured Flow

- Backing out of deeply nested blocks
  - recovery from errors
  - such conditions are called exceptions
- If implemented with goto out of subroutines
  - need to "repair" the stack of each current subroutine invocation
- Some languages (Clu, Ada, C++, Java, Common Lisp) provide an exception handling mechanism to do this
  - in C++: use throw and catch

Announcements

- Readings
  - Rest of Chapter 6
- Homework
  - HW 4 out – due on March 19
  - Submission
    - at the beginning of class
    - with a title page: Name, Class, Assignment #, Date
    - preferably typed

Selection

- Use the if...then...else notation introduced in Algol 60:

  if condition then statement
  else if condition then statement
  ...
  else if condition then statement

- In Algol 60 and Pascal – only one statement is allowed (or a compound statement using begin...end)

Selection

- Ambiguity – with what if does else associate?

```
if a = b then
   if c = d then
      statement1
   else
      statement2
```

  - Pascal – "disambiguating rule“: else associates with the last unmatched if

Selection

- Scheme – ambiguity solved by parentheses

```
(if (= a b)
        x
        y)       ; only one expression allowed in each arm

(cond
   ((= a b)
        p
        q        ; several expressions allowed in each arm
        r)       ; the value of last one is returned
   ((= a c)
        s
        t)
   (else
        u
        v))
```

Short-Circuited Conditions

- How does the compiler generate code for an if...then...else statement?
- Source code:

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

- Without short-circuit evaluation - evaluate entire Boolean expression, then jump

- Target code:

```
        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
        goto L3
L2: else_clause
L3:
```

Short-Circuited Conditions

- Same example:

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

- With short-circuit evaluation - evaluate only as much as needed in order to jump

- Target code:

```
        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:
```

Case/Switch Statements

- Alternative syntax for a special case of selection (from a set of discrete constants)
- Example (Modula-2):

```
CASE expr of
    1:        clause_A
|   2, 7:     clause_B
|   3..5:     clause_C
|   10:       clause_D
    ELSE      clause_E
END
```

- Specify values on each arm - they must be discrete and disjoint:
  - constants (1)
  - enumerations of constants (2, 7)
  - ranges of constants (3..5)

- Implementation
  - sequential testing (similar to if...then...else)
  - jump table (compute an address to jump in a single instruction)

Case/Switch Statements

```
CASE expr of
     1:         clause_A
|    2, 7:      clause_B
|    3..5:      clause_C
|    10:        clause_D
     ELSE       clause_E
END
```

- Jump table implementation:

```
     goto L6      -- go to address computation
L1: clause_A
     goto L7
L2: clause_B
     goto L7
L3: clause_C
     goto L7
L4: clause_D
     goto L7
L5: clause_E
     goto L7
L6:
L7:
```

```
T:  &L1                    -- expr is 1
    &L2                    -- expr is 2
    &L3                    -- expr is 3
    &L3                    -- expr is 4
    &L3                    -- expr is 5
    &L5                    -- expr is 6
    &L2                    -- expr is 7
    &L5                    -- expr is 8
    &L5                    -- expr is 9
    &L4                    -- expr is 10
L6: r1 := expr
    if r1 < 1 goto L5
    if r1 > 10 goto L5
    r1 -:= 1
    r1 := T[r1]
    goto *r1
L7:
```

Case/Switch Statements

- How efficient is the jump table implementation?
  - Time efficiency - always
  - Space efficiency
    - yes, when the overall range is small and dense
    - otherwise, better use sequential testing
- Variations across languages:
  - no ranges allowed in case arms (Pascal, C)
  - computed goto (Fortran)

    ```
    goto (15, 100, 150, 200), I
                          -- if I is 1, jump to 15; if I is 2, jump to 100...
    ```
  - array of labels (Algol 60)

    ```
    switch S := L15, L100, L150, L200;
    I = ...
    goto S[I];
    ```
  - ability to "fall-through" case arms (C) - must use break

Case/Switch Statements

- "Falling-through" in C:

```
switch (expr)
{
   case 1:   clause_A
                              break;
   case 2:
   case 7:   clause_B
                              break;
   case 3:
   case 4:
   case 5:   clause_C
                              break;
   case 10: clause_D
                              break;
   default:   clause_E
                              break;
}
```

Iteration

- Implemented as loops
- Usually executed for side-effects

- Mechanisms
  - Enumeration-controlled loops (for) - executed once for every value in a given finite set
  - Logically-controlled loops (while) - executed until some Boolean condition changes

Enumeration-Controlled Loops

- The number of iterations is known, and should not change during the loop
- Example (Modula-2):

```
FOR i:= first TO last BY step DO
   ...
END
```

- Issues:
  - Changes to the loop index (i), step or bounds (first and last)
  - Empty bounds
  - Direction of step
  - Jumps in and out of the loop

Enumeration-Controlled Loops

- Changes to the loop index, step or bounds
  - Prohibited in most languages (Algol 68, Pascal, Ada...)
  - The bounds and step are evaluated exactly once, before first iteration
- Pascal → nothing is allowed to "threaten" the index variable:
  - assign to it
  - pass it to a subroutine by reference
  - read it from file

Enumeration-Controlled Loops

- Empty bounds
  - Need to test termination condition before first iteration
  - If empty bounds, do not execute loop

```
FOR i:= first TO last BY step DO
   ...
END
```

- Target code:

```
      r1 := first
      r2 := step
      r3 := last
L1:   if r1 > r3 goto L2
      ...                          -- loop body; use r1 for i
      r1 := r1 + r2
      goto L1
L2:
```

Enumeration-Controlled Loops

- Alternative target code:

```
    r1 := first
    r2 := step
    r3 := last
    goto L2
L1: ...                          -- loop body; use r1 for i
    r1 := r1 + r2
L2: if r1 <= r3 goto L1
```

- Assumption (in both variants) - step is positive

Enumeration-Controlled Loops

- Direction of step
  - If the step is negative, need to generate different code
  - Problem - at compile time, direction may not be known
- Solutions
  - Require programmer to declare direction
    - Pascal: for i:= 10 downto 1 do ...
    - Ada: for i in reverse 1..10 do ...
  - Require step to be a compile-time constant (Modula-2, Modula-3)
  - First compute the number of iterations N (always $N \geq 0$), then execute the loop N times (Fortran)

Enumeration-Controlled Loops

- General implementation by using iteration count (Fortran):

```
    r1 := first
    r2 := step
    r3 := max(⌊(last-first+step)/step⌋,0)          -- iteration count N
    if r3 <= 0 goto L2
L1: ...                                             -- loop body; use r1 for i
    r1 := r1 + r2
    r3 := r3 - 1
    if r3 > 0 goto L1
L2:
```

- Works for any step direction

Enumeration-Controlled Loops

- Jumping in and out of the loop
  - difficult to implement
  - difficult to understand
- Gotos out of the loop
  - relatively clean
  - alternatives in structured languages - break in C
- Gotos that jump in the loop from outside
  - issues - what is the index, what are the bounds, etc.
  - prohibited in almost every language

Loops

- C
  - provides for, while and do loops
  - all are logically-controlled
  - for is just a more compact alternative to while loops
    - number of iterations is not known in advance
    - can change index, bounds, step within loop
    - programmer responsible for overflows

Logically-Controlled Loops

- Simpler that enumeration-controlled loops

  while condition do statement

- Approaches
  - Test before each iteration (most common, while in C)
  - Test after each iteration (do in C)
  - Mid-loop test and exit (in Modula-1):

```
loop
    statement_list
when condition exit
    statement_list
when condition exit
    ...
end
```

Recursion

- Equally powerful to iteration
- Any iterative algorithm can be rewritten recursively and vice-versa
  - No special syntax required
  - Fundamental to functional languages (Lisp, Scheme)
- "Naive" implementation of recursion is less efficient that iteration
  - overhead due to function calls - stack maintenance
- Efficient implementation – tail recursion

Tail Recursion

- Compute greatest common divisor:

```
(define gcd (lambda (a b)
   (cond ((= a b) a)
         ((< a b) (gcd a (- b a)))
         ((> a b) (gcd (- a b) b)))))
```

- The function is tail recursive
  - no additional computation follows the recursive call
  - returns what the recursive call returns
  - can reuse the memory space of current iteration for next one (no stack allocation)

- The compiler will "rewrite" as:

```
gcd (a b)
start:
   if a = b
      return a
   if a < b
      b := b - a
      goto start
   if a > b
      a := a - b
      goto start
```

Tail Recursion

- Changes to a function that is not tail recursive, to create tail recursion:

$$\sum_{i=low}^{high} f(i)$$

- Non tail recursive:

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

- Make it tail recursive:

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

- Need to call it initially with:

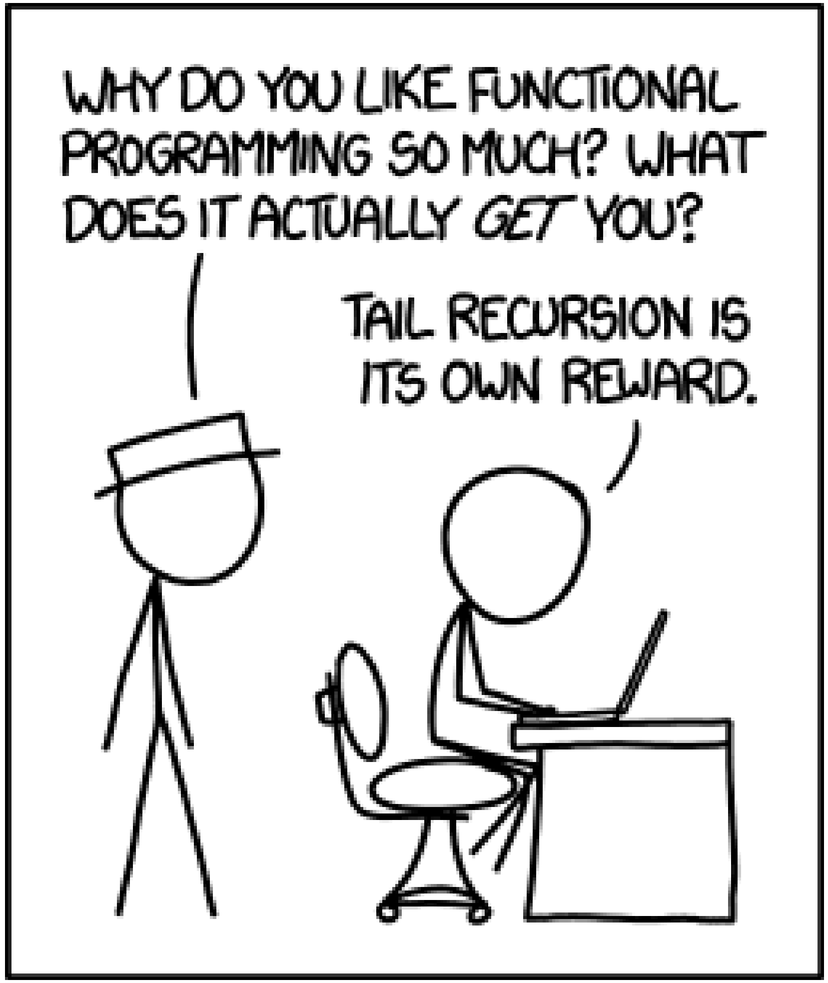
```
(sum f low high 0)
```

Tail Recursion

- Add a wrapper function, that does the initial call:

```
(define summation (lambda (f low high)
  (letrec ((sum (lambda (f low high subtotal)
                  (if (= low high)
                      (+ subtotal (f low))
                      (sum f (+ low 1) high (+ subtotal (f low)))))))
     (sum f low high 0))))
```

Tail Recursion



Evaluation of Function Arguments

- When are the arguments evaluated?
  - Before being passed to the function (applicative-order evaluation)
    - in most languages
    - safer, more clear
  - Pass a representation of unevaluated parameters to the function; evaluate them only when needed (normal-order evaluation)
    - typical for macros
    - can be faster
- Normal-order evaluation example (C) - check if n is divisible by a:

```
#define DIVIDES(n,a)      (!((n) % (a)))
```

- When used – textual substitution:

```
DIVIDES (x, y+z)      =>      (!((x) % (y+z)))
```

Evaluation of Function Arguments

- Normal-order evaluation - may have unexpected effects:

```
#define MAX(a,b)    ((a) > (b) ? (a) : (b))
```

  - What happens if we use MAX (x++, y++) ?
    - side-effects (increments) happen more than once

```
#define SWAP(a,b) { int t = (a); (a) = (b); (b) = t; }
```

  - What happens if we use SWAP (x, t) ?
    - obtain { int t = (x); (x) = (t); (t) = t; }
    - simple text substitutions, no naming and scope rules

- In C++ - avoid these problems by using functions
- Best compromise - inline functions
  - have the semantics of regular functions
  - if possible, the compiler expands the function definition at the point of call (similar to the macros)

Announcements

- Readings
  - Rest of Chapter 6