Chapter 6: Flow Control

長庚大學資訊工程學系　陳仁暉　助理教授
Tel: (03) 211-8800 Ext: 5990
Email: jhchen@mail.cgu.edu.tw
URL: http://www.csie.cgu.edu.tw/~jhchen

Control flow

- Ordering information is fundamental to imperative programming
- Categories for ordering instructions
  - sequencing
  - selection
  - iteration
  - procedural abstraction (Ch. 8)
  - recursion
  - concurrency (Ch. 12)
  - nondeterminacy

Chapter contents

- Expression evaluation
  - syntactic form
  - precedence & associativity of operators
  - order of evaluation of operands
  - semantics of assignment statement
- Structured & unstructured control flow
  - goto-statement
- Sequencing, selection, iteration, recursion, nd

Expressions

An expression generally consists of

- Simple object (or atomic)
  - (named or literals) variables and constants
- Structured
  - function applied on arguments
    - arguments are expressions
  - word *operator* is commonly used for functions with a special (operator) syntax
    - arguments of operators are called *operands*
    - syntactic sugar (Ada, C++)

‘fixity’ of operators

- Prefix: operator is *before* its operands
  - `-x`
  - `(+ 1 2)`
- Infix: operator is *among* (between) its operands
  - `x - y`
  - `myBox: displayOn: myScreen at: 100@50`
  - `a = b != 0 ? a/b : 0;`
- Postfix: operator is *after* its operands
  - `p^, i++`

Precedence & Associativity

- What is an operand of what?
  - people don’t want to put parentheses around every subexpression
  - what if no parentheses?
  - **`a + b * c ** d ** e / f`**
  - note: this is a problem only with infix operators
- Precedence rules
  - tell how tightly operands bind their arguments (e.g., * and +)
- Associativity rules
  - tell whether a sequence of operators (of equal precedence) groups operands to the left or to the right
  - ‘left-to-right’ grouping: a – b – c → ((a – b) – c)
  - ‘right-to-left’ grouping: a ** b ** c → (a ** (b ** c))

Precedence rules

- Given an expression
  - find the operator with highest precedence
  - assign to it the left & right operand
  - x + y * z → x + (y*z)
- C, C++, Java have too many precedences
  - Fig. 6.1 shows only a fragment
  - best to parenthesize properly because nobody remembers them
- Pascal has too few precedences
  - IF a < b AND c < d THEN ... is parsed to IF a < (b AND c) < d THEN ...
    - syntactic error unless a,b,c,d are all Booleans
    - and if they are, this is most probably not what the programmer had in mind
  - → most languages give higher precedence to arithmetic operators and lower to boolean operators, e.g., C/C++, Java
- No precedences at all: APL, Smalltalk

Associativity rules

- Operators are commonly ‘left-to-right’ associative
  - i.e. they form groups to the left
  - a – b – c – d → (((a – b) – c) – d)
- But exceptions exist (right-to-left)
  - exponentiation: a ** b ** c ** d → (a ** (b ** (c ** d)))
    - Ada: ** ‘does not associate’
  - assignment expressions (a = b = c + d)
- Recommendation
  - precedence and associativity vary much from one language to another →
  - programmer should voluntarily use parentheses

Assignments

- Purely functional languages
  - computation = expression evaluation
  - effect of *any* expression = value of that expression
- Imperative languages
  - computation = ordered series of changes to the variables in computer memory
  - changes are made by *assignments* → evaluation of expressions may have *side effects*
  - both *statements* and expressions
- Side effect
  - Definition: any other way than returning a value that influences the subsequent computation
  - purely functional languages have no side effects
    - expressions always return the same value for same binding environment
    - the *time* of the evaluation has no effect on the result
    - Expressions in a purely functional language are said to be *referentially transparent*

What is a variable?

- There are differences in the semantics of assignment statement
  - often ‘invisible’, but have a major impact when pointers are used
- C examples
  - **`d = a;`** **`a`** refers to the *value* of a
  - **`a = b+c;`** **`a`** refers to the *address* of a
- Variable is a ‘named container for a value’
  - *value model* of variables
  - ‘left-hand-side’: address, l-value, location of the container
  - ‘right-hand-side’: value, r-value, contents of the container
- In general
  - any expression that yields a location has an l-value
  - and an expression that yields a value has an r-value

Expressions and l-values

- All expressions are not l-values
  - not all values are locations
  - not all names are variables
- Examples
  - **`2+3:=a`** doesn't make much sense
  - not even **`a:=2+3`** if **`a`** is a constant
- Not all l-values are simply names
  - legal C statement: **`(f(a)+3)->b[c] = 2;`**
  - in C++ one can return a *reference* (to a structure) and write simply **`g(a).b[c] = 2;`**

Reference model (of variables)

- Variables are named *references* to values
  - not containers
  - Figure 6.2
- Unique objects (e.g. for all integer values)?
  - in reference model b & c refer to the same object
    - necessity to decide identity?
  - in Clu, integers are *immutable*
    - value 2 never changes
    - it doesn't matter whether we compare 2 'copies of 2' or 2 references to the 'unique 2'
    - most implementations of languages using the reference model have adopted the 'copy approach' for efficiency reasons (for immutable types)
  - → different definitions for 'being equal'
- *Dereferencing*
  - process of obtaining the referred r-value
  - required when context expects an r-value
  - automatic in most languages, explicit in ML
- Java: value model for built-in types, reference model for classes

Orthogonality

- Name originates from linear algebra
  - orthogonal set of vectors → none of the members depends on the others, all are required to define the vector space
- Principal design goal of Algol 68
  - language features = orthogonal set
    - can be used in any combination
    - all combinations make sense
    - features always mean the same (no matter the context)
  - e.g. Algol-68 was *expression-oriented*
    - no notion of a statement, just use expressions without their value
    - ‘statements’ can appear as expressions
- C: intermediate approach
  - expression can appear in statement context
  - sequencing and selection expressions (to use statements in expression context)

Assignments in expressions

- Value of an assignment: right-hand-side
- May lead to confusion
  - different assignment & equality operators
    - Algol 60, Pascal: a := b assign (a = b: equality)
    - C, C++, Java: a = b assign (a == b: equality)
  - further confusion for C
    - lacking a boolean type → integer used instead
      - 0: false, all other values: true
    - both if (a = b) and if (a == b) are legal
  - C++ has bool but it coerces (a = b) to bool
    - automatically for numeric, pointer & enumeration types!
  - Java (finally) disallows use of int in boolean context

Initialization

- Imperative languages
  - already have a construct to specify variable values (assignment statement)
  - → not all have a special ‘initial value’ construct
- Why should such a thing be useful?
  - static variables can be initialized at compile-time
    - saves time (recompiled)
    - in reference model also the values of stack/heap variables (the actual references are created at run time)
  - common error: use of an uninitialized variable
    - program is still buggy but at least errors are systematic
    - ‘uninitialization’ may be caused also by *other* reasons
      - dangling pointers
      - updates to the tag-field of a variant type

Initialization choices

- Initialization as assignment
  - Pascal extensions allow initializations of simple types at variable declaration
  - C, Ada: *aggregate* expressions to initialize even structured types at compile-time
- Default values
  - C initializes all static data to null/0 values
- Constructors
  - A constructor routine of a class is automatically called when an object of that class is created
  - C++ allows to define own assignment statement for classes
    - e.g. variable-length strings
- Incorrect values (causing a dynamic semantic error)
  - bugs are FOUND (legal default values may mask them)
  - IEEE NaN (not a number) constant is catched by hardware (fast to check at runtime)

Catching uninitialized variables

- In general, an expensive operation
  - for many types, all bit patterns are legal
  - → must extend data with an explicit (boolean) tag field
    - set ‘uninitialized’ at elaboration time
    - set ‘initialized’ at each assignment
  - run time checks at each use
- Note
  - any potential error that depends on run-time flow
    - e.g. using an uninitialized value
  - is provably impossible to detect at compile-time in general
  - but can be catched in some restricted cases
    - ‘straight-line’ code, e.g., a = 3 + 1;
    - Java: precise definition of ‘definite assignment’ (each possible path must assign a value)

Assignment operators

- Updating variables is *very* common in imperative languages
  - → ‘update statement’ x := x + b is common
  - cumbersome to red/write if ‘x’ is complex
    - are the both sides really the same?
  - redundant address calculations for ‘x’
    - address calculations may have side-effects!
- Assignment operators answer to all these
  - e.g. x += b
  - self-clear whether both sides same because only one side
  - address is computed only once
  - note: C has 10 assignment operators (one for each operator)

C & post increment/decrement operations

- Adjust the value of x by one
  - ‘special case of a special case’
  - still occurs very often
  - C applies it also to pointers
    - *p++ = *q++
    - +/-1 = relative to the size of the pointed structure
- ++i (pre-increment)
  - i = i+1, value = i
  - equal to i += 1
- i++ (post-increment)
  - value = i, i = i+1
  - equal to (temp = i, i+=1, temp)

Simultaneous Assignment

- a, b := c, d (in CLu, ML, and Perl)
  - *not* the sequencing operator of C
  - value-based model
    - variable *tuple* (multiway) a, b is assigned the value tuple c, d
  - reference model
    - reference tuple a, b is assigned another reference tuple
      - references to the values of c & d
    - a, b := b, a
    - a, b, c := foo(d, e, f)
    - ML & Haskell: pattern matching (generalization of tuples)

Evaluation order in expressions

- Precedence & associativity
  - tell which operator is applied to what operands
  - does not tell in what order operands are evaluated
    - a – f(b) – c * d : is a – f(b) evaluated before c*d?
    - f(a, g(b), c): is g(b) evaluated before c?
- Why does the order matter?
  - ***Side effects***
    - consider a – f(b) – c * d when f(b) modifies d
  - ***Code improvement***
    - register allocation
      - a * b + f(c) (p.263)
      - call f(c) first to avoid saving a*b into memory
    - instruction scheduling
      - a := B[i]; c := a*2 + d*3;
      - evaluate d*3 before a*2 (loading a takes 2 machine cycles, can do d*3 while waiting)

Ordering & language implementations

- Leave the order to the compiler to decide
  - many implementations explicitly state that the order is *undefined*
- Left-to-right evaluation (Java)
- Allow (even larger scale) rearranging
  - commutative, associative, distributive operations
    - use of these may lead to the invention of common subexpressions (and code improvements)
  - unfortunately computers do not follow mathematics
    - limited range → overflows
    - limited precision → ‘absorption’ of small values
    - note: some languages have ‘integers of infinite size’
- Want a certain order?
  - use parentheses in operator expressions
  - no way to affect argument evaluation in subroutine calls
    - better not write programs where this order matters

Short-circuit evaluation

- Special property of Boolean expressions
  - the whole expression has not to be computed in order to determine its value
    - (a < b) AND (b < c). If a >= b → no need to check b < c
    - similarly for (a > b) OR (b > c). If a > b
- Benefits
  - can save execution time
  - most important: changes the semantics of Boolean expressions
  - examples
    - traversing a list (dereferencing a null pointer)
    - indexing an array (index out of bounds)
    - division (by zero)
  - full evaluation would lead to a runtime error
- Drawbacks
  - sometimes we really want the full evaluation (side effects)

Short-circuit & implementations

- Always full evaluation
- Always partial evaluation
- Own operators for full & partial evaluation
  - Clu: and, or, cand, cor
  - Ada: and, or, and then, or else
  - C: &, |, &&, ||
- Note
  - if the expression is used to control program execution (if-statement, while-loop)
  - then we don't necessarily need the value at all (only want to direct the program)

Structured & unstructured flow

- Jumps in assembly languages
  - only way to redirect program execution
  - → goto statement of Fortran (and other early languages)
- Goto considered harmful
  - hot issue in 1960/70s
  - most modern languages
    - do not have jump statements at all
    - or implement it only in some restricted form

Structured programming

- Emphasizes
  - top-down design (i.e., progressive refinement)
  - modularization of code
  - structured types (records, sets, pointers, multi-dimensional arrays)
  - descriptive variables and constant names
  - extensive commenting conventions
  - especially *structured control-flow constructs*
- Most structures were invented in Algol 60
  - case-statement in Algol W

Are gotos needed?

- Special situations where
  - control should be redirected in a way that is hard (or impossible) to catch using structured constructs
  - but which can easily be implemented with jump statements
- Mid-loop exit & continue
  - goto out of loop/end of loop
  - → own control structures
- Early returns from subroutines
  - goto return address
  - → return statement
- Errors and exceptions
  - non-local goto & *unwinding* (of subroutine stack and register values)
  - nonlocal gotos are a 'maintenance nightmare'
  - → exception handling mechanisms

Continuations

- Generalization of the ‘non-local goto’
  - in low-level terms
    - code address (to continue execution from)
    - referencing environment (to restore)
    - quite a lot like a 1st class subroutine
  - in high-level terms
    - *context* in which the execution may continue
  - all non-local jumps are continuations
- Scheme language (successor of LISP)
  - continuations are 1st class data objects
  - programmer can design own control structures (both good and bad ones)
  - implemented using the ‘heap frame’ idea

Sequencing

- Central to imperative programming
  - control the order in which side effects occur
- *Compound statement*
  - list of statements enclosed in 'statement parentheses'
    - begin – end
    - { - }
  - can be used 'as a single statement'
- Block
  - compound statement with a set of declarations
- Value of a (compound) statement?
  - usually the value of its final element

Side-effects: good or evil?

- Side-effect freedom
  - functions will always return same values for same inputs
  - expressions return the same value independent of the execution order of subexpressions
  - easier to
    - reason about programs (e.g. show correctness)
    - improve compiled code
- Side-effects are desirable in *some* computations
  - pseudo-random number generator (remembers the ‘seed’)
- Language design
  - Euclid, Turing: no side-effects in functions
  - Ada: functions can change only static or global variables
  - most: no restrictions at all

Selection

- IF-THEN-ELSE
  - Algol 60: if ... then ... else if ... else
  - most languages contain some variant of this
- Language design
  - *one* statement after then/else (Pascal, Algol 60)
  - → nested IFs cause ‘dangling else’ problem
    - Algol 60: statement after ‘then’ must begin with something else than ‘if’ (e.g. ‘begin’)
    - Pascal: closest unmatched then
  - statement list with a terminating keyword
    - special elsif or elif keyword
    - to keep terminators from piling up at the end of a nested list

Selection & short-circuit evaluation

- The actual value of the ‘control expression’ is not usually of interest
  - only the selection (of program flow) itself
  - most machines contain conditional jump/branch instructions that directly implement some simple comparisons
  - → compile *jump code* for expressions in selection statements (and logically controlled loops)
- Example on page 273
  - full evaluation (r1 will contain the value)
  - short-circuit: execution is shorter & faster
    - Notice: the value can still be generated if it is required somewhere
    - (value is obvious after the selection)

case/switch statements

- Alternative syntax for a *special case* of nested if-then-else
  - each condition compares
    - the *same* integer (or enumerated type) expression
    - against a different compile-time *constant*
  - Modula-2 example on p. 275
- Corresponding case-statement
  - starts with the controlling expression
  - each conditional part becomes an *arm* of the case-statement
  - each constant value becomes a *case label*, which must be
    - type compatible with the tested expression
    - disjoint

Why case/switch is useful?

- Syntactic elegance?
- Allows efficient target code to be generated
  - examples on p. 276 (if-then & corresponding case)
  - case statement can *compute* the jump address in a single instruction
- *Jump table* implementation
  - table containing arm addresses
    - one entry for each value between the lowest and highest case label value
  - use the case expression as an index to this table
  - additional check for table bounds

Alternative case implementations

- Jump table
  - very fast
  - space-efficient when
    - the set of case labels is dense
    - the range of case labels is small
- Sequential testing
  - useful when the number of case arms is small
- Hash tables
  - useful when then range of label values is large
  - requires a separate entry for each possible value → can get large
- Binary search structures
  - implement label *intervals* and search on them
- Notable: Good compiler must be able to make the correct decisions and use the appropriate implementation

case & language design

- Varying syntactic details
  - ranges allowed in label lists?
    - may require binary search
    - Pascal, C: not allowed
  - arm: single statement or statement list
  - action to take if no label matches
    - do nothing (C, Fortran)
    - crash (Pascal, Modula: runtime error)
    - use a default arm
      - keywords: else, otherwise, default
      - Ada: required by compiler (unless all labels are covered)
- Historical ancestors
  - Fortran: computed goto
  - Algol 60: switch = array of labels
  - Algol 68: array of statements (orthogonality!)

C switch

- Each possible value must have its own label
- Need to simulate label lists
  - allow empty arms and
  - let control fall through all ‘empty arms’ to the common statement list
- But control ‘falls through’ any arm!
  - each arm must be terminated with an explicit break-statement
  - but of course nothing forces one to write them → ‘smart’ programming tricks
  - leads to difficult bugs
- C++ and Java proudly follow the tradition

Iteration

- Allows the repeated execution of some set of operations
  - usually takes a form of (control flow) *loops*
  - loops are executed because of the side effects they cause
  - without iteration (and recursion) computers would be useless!
- Two principal loop varieties
  - difference: the mechanism used to decide how many times they iterate
  - *enumeration-controlled* (definite iteration)
    - execute once for each element in some (finite) set
    - number of iterations is known before the loop is executed
  - *logically controlled* (indefinite iteration)
    - execute until some Boolean condition changes its value
  - usually distinct in languages (exception: Algol 60)

Enumeration-controlled loops

- Fortran DO-loop (p.280)
  - **`do 10 i = 1, 10, 2`**
  - **`10:`** label at the *last* statement of the loop body
    - usually contains the **`continue`** (no-op) statement
  - **`i:`** *index* variable of the loop
    - **`1:`** initial value of **`i`**
    - **`10:`** the maximum value **`i`** may take
    - **`2:`** the amount by which **`i`** is increased in each iteration
      - updates are executed *after* the loop body is executed
  - easy and efficient compilation

Minor problems with Fortran DO

- Loop bounds must be positive integer variables/constants
  - Fortran 77: integer & real expressions
  - Fortran 90 took reals away (precision difficulties)
- Typing errors are easy to make
  - DO 5 I = 1,25 is a for-loop
  - DO 5 I = 1.25 is an assignment (to DO5I)
    - pre-90 Fortrans ignore blank spaces
  - claim: NASA Mariner 1 space probe was lost because of this
  - Fortran 77: additional comma before variable name

Major problems with Fortran DO

- statements in the loop body may change the loop index
  - number of iterations is not known
  - hard-to-find bug or a hard-to-read code
- gotos are allowed into & out of the loop
  - jump in without initializing loop counter?
- value of the loop counter after termination?
  - implementation-dependent
  - expected value: L + ((U-L) div S) + 1) * S i.e. the first value that exceeds the bound U
  - arithmetic overflow possible if U is large
    - → negative value & infinite iteration (or run-time exception)
    - more complex code to check for overflow? → index may contain its last in-bounds value after termination
- bounds tested *after* the loop is executed
  - → at least one iteration no matter what the bounds are

Language design issues with for-do -loops

- Can the loop index or loop bounds be modified in the loop
  - if so, what is the effect?
  - in general: is the enumeration always the same?
- upper bound < lower bound?
- value of loop index after termination?
- can one jump into/out of loops

Commonly used implementation decisions

- Prohibit changes to loop indices/bounds
  - and good so
  - bounds are evaluated only once (later changes have no effect)
- Bounds are checked *before* the first iteration
  - takes care of ‘empty bounds’
  - compiled code is longer but more intuitive (p. 283)
  - improved version: only one branch

Negative / unknown steps

- IF the step is a variable THEN the direction of the iteration is not known at compile-time
- Naive implementation
  - test sign & provide 2 tests (for both cases)
- Direction required by language design
  - Ada, Pascal: downto, reverse
  - Modula-2: step must be a compile-time constant
- use *iteration count* instead of index variable to control termination
  - compute count from given bounds & step (p. 284)
  - avoids sign test and arithmetic overflow issues!
  - most modern processors have instructions for ‘decrement-test-branch’
  - sometimes the index variable can be eliminated in code improvement

Loop index value after loop

- Leave undefined (Pascal)
- 'Most recently assigned' (Fortran 77, Algol 60)
  - normal termination: first value exceeding bound
  - overflows & subrange types: 'first value exceeding' may be incorrect or illegal
- 'Last one that was valid'
  - compiled code is slower
  - necessary if overflow is a danger
- Avoid the issue altogether (Ada, C++)
  - make index a variable *local* to the loop
  - for-statement declares the index (type induced from bounds)
  - not visible after → value can not be even accessed
  - not visible before → no danger of overwriting an old value

Combination loops

- Algol 60 'overkill' (p. 286)
  - index values defined by a sequence of enumerators (value, range, while-expr)
  - each expression is *re-evaluated* at the top of the loop
    - otherwise the while-form would be quite useless
    - leads to hard-to-understand programs
- C for-statement
  - equivalent to a special kind of a while-loop
    - control information collected to the header
  - everything is on programmer's responsibility
    - overflow checking, side effects
  - note: any expression (including empty & expression list) is allowed

Iterators

- Iterating over something else than an arithmetic sequence?
- In general, iterate over the elements of any well-defined set
  - e.g. nodes of a binary tree in pre-order
  - some languages support this by design (Clu, Icon)
  - some by library classes (Java, C++)

Clu iterator

- See Figure 6.5 (p.288)
- Programmer can write own iterators
  - special kind of a subroutine (co-routine)
  - invocation: **for** e **in** iterator_call(args) **do**
  - iterator may *yield* results several times and return once
  - implementation: store also ‘call address’ to stack frame
- for-loops (iterator invocations) may be nested
- iterators can be recursive (see Fig. 6.6)

Icon generator

- deeply embedded to the semantics of the language
  - generator can be used in any context that accepts an expression (p. 290)
- can be used to implement backtracking search
  - tests succeed or fail (instead of being true/false)
  - if a failing test contains a generator, Icon tries it again (and tests the next value)
  - *reversible assignments* <- are restored to previous values when backtracking
- built-in generators
  - to .. by for arithmetic enumeration
  - find, upto for string manipulation
- user-defined generators
  - any subroutine using *suspend* instead of *return*

Enumerating without iterators...

- Implementation of iterators
  - involves some special implementation problems
  - requires jumping back and forth in the subroutine stack (see 8.6.3)
  - → not implemented in most languages
- Similar effect through programming conventions
  - Fig. 6.7 (p.291): C 'iterator' for the elements of a binary tree
  - note: a recursive implementation could be better
  - data structure
    - holding all information that an iterator would hold automatically
  - interface routines
    - create/destroy iterator
    - test whether it is empty
    - get next element if there is some

...Enumerating without iterators

- Euclid generators (p. 293)
  - for-loop = interface to a generator module
  - generator = module which exports
    - variables value & stop
    - function Next
  - create/destroy: module initialization
  - all interface calls are made automatically
- C++
  - use container classes & inheritance
  - by heavily overloading (!=, ++, ->) and using constructor and destructor, one gets ‘almost a for-loop’
- Java
  - implement Enumeration interface (p. 294)

Logically controlled loops

- Not that many semantic subtleties
- Only real question:
  - where in the body of the loop
  - the terminating condition is tested?
- Most common approach: before each iteration
  - Algol W & Pascal: WHILE condition DO statement
  - most successors of Pascal:
    - DO starts a statement list
    - loop ends with some terminating keyword
- Languages without them?
  - pre-90 Fortrans: simulate (negate test & jump over if true)
  - Algol 60: ‘dummy enumeration’ combined with the actual loop

Post-test loops

- REPEAT statement UNTIL condition
  - iteration continues until condition comes true
- Eliminates code duplication if we know that the body is executed at least once
- This happens especially when the body has to be executed in order to compute the termination condition
- Note: do-while of C works in the ‘other direction’
  - iteration continues as long as the condition holds

Mid-test loops

- Can be simulated with conditional gotos
- Modula-1
  - WHEN condition EXIT
  - a loop may contain any number of these
  - part of the loop syntax
  - must be at the top level of the loop
- Modula-2
  - simple EXIT statement
  - can appear anywhere, typically after some IF
  - compiler must check that EXITs are inside some loop
  - C **`break`** works in a similar manner

Multi-level exits

- Nested loops → exit all / some of them?
  - with ‘standard’ exit one must introduce auxiliary boolean variables & add conditional statements
- Ada
  - loops can be named
  - EXIT can specify which (named) loop it breaks
  - Java has adopted a similar mechanism

Recursion

- No special syntax required
  - possible in any language that allows subroutines to call themselves
- Recursion or iteration? (Which one is better?)
  - Imperative language: based on side-effects → iterate
  - functional/logic language: ‘pure’ → recur
  - choice is quite often only a matter of taste
    - sum: iterate, GCD: recur (p. 297)
    - but also the opposite is possible (p. 298)

Tail-recursion optimization

- Common argument: iteration is faster than recursion
  - makes sense, because a function call must allocate space from stack etc, whereas iteration only jumps
  - but good compilers can transform recursion automatically into iteration!
- Tail-recursive functions
  - recursive call is the *last* action the function makes
  - i.e. no computation follows the call
  - → function returns whatever the recursive call returns
- Tail-recursion optimization (TRO)
  - dynamic stack allocation is unnecessary → the recursive call can *reuse* the frame of the caller
  - recursive calls become jumps to the start of the routine

Generalized TRO

- Apply TRO even in non-tail cases
  - make the code following the recursive call a *continuation*
  - pass the continuation as an extra parameter
  - execute continuations after ‘termination’
- Programming tricks
  - transform non-TR functions to TR ones with helper routines
  - well-known in ‘functional programming community’
  - use of accumulators (summation, p. 299)
    - works when a binary function is known to be associative

Recursion 'algorithmically inferior'?

- Example: Fibonacci numbers (p. 300)
  - defined via mathematical recurrence formula
  - → leads directly to a naive O(n^2) recursive implementation
  - one can easily write an O(n) iterative program
  - a skillful programmer can do the same even with recursion
    - helper routine: remember the previously computed 2 numbers
    - simulation of iteration via recursion?
    - YES but WITHOUT side-effects!
- What about the 'non-skillful' ones?
  - define iterative constructs as syntactic sugar for tail recursion
  - programmer writes for-loops, compiler takes care of the 'skill'
  - special mechanisms needed in order to refer to the 'old' values of variables (from the previous iteration)
  - Sisal example code on p. 301 is still side-effect free

Nondeterminacy (nd)

- Note: we skip 6.6.2
  - I really do not understand why it is located here, perhaps we return to it in chapter 8
- Nondeterministic choice (ch 6.7)
  - choice between alternatives is deliberately unspecified
    - e.g. evaluation order of subexpressions
    - note: choice = iteration control → nd iteration
  - *guarded commands*
    - notation for nd selection & nd iteration
    - Dijkstra -75
    - most current implementations follow this notation

ND selection

- max(a,b): nd choice when a=b
  - different imperative implementations make different choices
  - in practice this does not matter → special notation to point this out?
- Guarded selection
  - combination of guarded commands
  - guard: logical expression
  - guard + statement = guarded command
  - statement may be executed if the guard is true
    - nd choice when several are

ND iteration

- Perform a loop around guarded commands
  - none of guards true → terminate
  - otherwise make an nd choice
  - e.g. Euclid’s gcd algorithm
- ND choice is not only esthetics!
  - some concurrent programs really need it
  - correctness of execution depends on a truly nd choice
  - example in Figure 6.9

Implementing an ND choice

- If-then-elseif
  - always favors earlier requests
  - → some requests may have to wait forever
- Keep guarded commands in a circular list
  - guards are checked in the list order
  - always continue from the one succeeding the previously chosen guard
  - works well in *most* cases
    - example of bad performance on page 307
    - A can be chosen only at odd iterations of the loop
    - imagine A(), B(), C() always succeed
    - B → C → B → C → ...

Fair nd choice

- ND choices should have a guarantee of *fairness*
- What is fair?
  - no true guard can be always skipped
  - no guard that is true infinitely often can be always skipped
  - any guard that is true infinitely often is chosen infinitely often
    - satisfied if the choice is truly random
    - i.e. implementation must use some good pseudo-random number generator
    - but these are computationally expensive to use
- In practice
  - circular list
  - rough random numbers from the cpu clock
- Guards & side effects? (full/partial evaluation of guards)

Summary

- Expression evaluation
  - l- and r- values
  - variable models (value/reference)
  - precedence, associativity, ordering
  - shortcircuit evaluation & implementation
- Principal forms of control
  - sequencing, selection, iteration
  - recursion, nondeterminacy

Evolution of control constructs?

- Clearly some has happened
  - just compare Fortran with Ada
- Goals that are driving the evolution
  - humans: ease of programming, semantic elegance
  - machines: ease of implementation, efficiency
  - sometimes contradictory, sometimes complementary

Ease and efficiency

- Both goals satisfied → go implement it
  - short-circuit evaluation
    - cleaner semantics (null pointer check & dereferencing)
    - fast implementation (jump code)
  - index variables local in for-loops
    - value after iteration is not a problem
    - no need to check for arithmetic overflow
- Improvement worth a small run-time cost
  - midtest loops (need more branch instructions)
  - iterators
  - large payback in decreased programming work

...Ease and efficiency

- Compilers are better
  - some ‘costly’ constructs are now possible
  - e.g. label ranges in case statements (needs binary search)
  - note: also programmer may help the compiler (e.g. pointing out common subexpressions)
- Some constructs are still too expensive
  - lazy evaluation, continuations, truly nd choice
  - used only in special cases

Programming conventions

- Use of a ‘primitive’ language does not imply that the programs are primitive, too
- No short-circuit evaluation
  - use nested selection statements
- No iterators
  - use a bunch of subroutines with same functionality
- No midtest loops
  - auxiliary Boolean variables & nested selections
- No modules
  - use consistent naming of subroutines
- etc etc