

Control Flow

Chapter 6



Control Flow

- Basic paradigms for control flow:
 - Sequencing
 - Selection
 - Iteration
 - Procedural Abstraction
 - Recursion
 - Concurrency
 - Exception Handling
 - Nondeterminacy



Control Flow

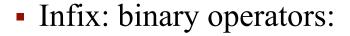
- Sequencing:
 - major role in imperative languages
 - minor role in functional languages
- Recursion
 - major role in functional languages
 - less important in imperative languages (iteration)
- Logic programming
 - no control flow at all
 - programmer specifies a set of rules
 - the implementation finds the order to apply the rules







- Expression: operands and operators
- Operator
 - function: a + b means +(a, b)
 - Ada: a+b is short for "+" (a,b)
 - C++: a+b is short for a.operator+(b)
- Notation
 - prefix + a b or +(a, b) or (+ a b)
 - *infix* a + b
 - postfix a b +
- Infix: common notation; easy to work with
- Pre/Postfix: precedence/associativity not needed



$$a + b$$

• Prefix: unary operators, function calls (with parentheses)

$$-4$$
, f(a, b)

• Scheme: prefix always – *Cambridge Polish* notation

Postfix: Pascal dereferencing ^, C post in/decrement

Ternary operators: C++ conditional operator '?:'

$$(a > b)$$
 ? a : b



- Precedence, associativity
- Fortran example: a + b * c ** d ** e / f
- Precedence levels
- C, C++, Java, C#: too many levels to remember (15)
- Pascal: too few for good semantics
 if A < B and C < D then ... means
 if A < (B and C) < D then ...</pre>
- Fortran has 8 levels
- Ada has 6 (it puts and & or at same level)
- Associativity: usually left associative
 - Right associative; C: a = b = c means a = (b = c)
- Lesson: when unsure, use parentheses!

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



- *Side Effect*:
 - any effect other than returning a value to surrounding context
 - essential in imperative programming
 - computing by side effects
 - (pure) functional languages: no side effects
 - same value returned by an expression at any point in time
- Value vs Reference
 - d = a value of a
 - a = b + c location of a
 - Value model: a variable is a named container for a value
 - C, Pascal, Ada
 - Reference model: a variable is a named reference to a value
 - Scheme, Lisp, Python, Clu





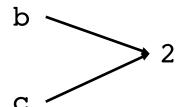
• Example:

- Pascal (value model):
 - any variable can contain value 2
- Clu (reference model):
 - there is only one 2

- value model
 - a [4]
 - b 2
 - c 2

reference model







- Value vs Reference
- Java: in-between
 - built-in types value model
 - user-defined types reference model
 - drawback: built-in types cannot be passed when user-defined is expected – wrapping is used (boxing)
- C#: user can choose
 - class reference
 - struct value
- Important to distinguish between variables referring to:
 - the same object or
 - different objects whose values happen to be equal
 - Scheme, Lisp provide several notions of "equality"



Subroutines: Parameter Passing



- Call by value: pass the value
 - C, C++, Pascal, Java, C#
- Call by reference: pass the address
 - Fortran, C++, C (pointers)
- Call by sharing:
 - Java, C#, Python, Scheme
- Call by *name*: direct substitution; evaluated each time it is needed
 - Algol 60, Simula
- Call by *need*: call by name with memoization
 - Haskell, R

Short-circuiting

Short-circuiting

- if a > b then the second part does not matter
- *Short-circuit evaluation*: evaluate only what is needed
 - Lazy evaluation
 - can save time:

```
if (unlikely_cond && expensive_cond) ...
```

- Semantics change:
 - Avoiding out-of-bounds indices:

if
$$(i \ge 0 \&\& i < MAX \&\& A[i] > foo) ...$$

Avoiding division by zero:

if
$$(d == 0 \mid \mid n/d < threshold) ...$$



Short-circuiting: example

• C list searching:

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

Pascal does not have short circuit:

```
p := my_list;
still_searching := true;
while still_searching do
    if p = nil then
        still_searching := false
    else if p^.key = val then
        still_searching := false
    else p := p^.next;
```

- Sometimes side effects are desired
 - C has also non-short-circuit: &,

Short-circuiting: implementation

if ((A > B) and (C > D)) or $(E \neq F)$ then then_clause else else clause

Without short circuit

L3:

```
r1 := A -- load
    r2 := B
    r1 := r1 > r2
    r2 := C
   r3 := D
    r2 := r2 > r3
    r1 := r1 & r2
    r2 := E
    r3 := F
    r2 := r2 \neq r3
    r1 := r1 | r2
    if r1 = 0 goto L2
L1: then clause -- (L1 unused)
    goto L3
L2: else clause
```

• With short circuit (jump code)

L2: else clause

L3:

```
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
```



Iteration

- Arbitrary complexity of programs:
 - Iteration for, while, ...
 - Recursion
- Iterate over collections
 - Iterator objects:
 - C++, Java, Euclid
 - True iterators:
 - Python, C#, Ruby, Clu
 - First-class functions
 - Scheme, Smalltalk



Iteration

Python – user-defined iterator

```
class PowTwo:
    def init (self, max = 0):
        self.max = max
    def iter (self):
        self.n = 0
        return self
    def next (self):
        if self.n < self.max:</pre>
            result = 2 ** self.n
            self.n += 1
           return result
        else:
            raise StopIteration
```



Iteration

Python – user-defined iterator

```
a = PowTwo(3)
i = iter(a)
print(next(i)) # 1
print(next(i)) # 2
print(next(i)) # 4
print(next(i)) # raises StopIteration
```



True iterators



• Example – Python:

```
for i in range(first, last, step):
...
```

- range built-in iterator
- use a call to a yield statement
- like return but control goes back to iterator after each iteration
- the iterator continues where it left off
- yield separate thread of control
 - its own program counter
 - execution interleaved with that of the for loop



True iterators

Python generator – much simpler

```
def PowTwoGen(max = 0):
    while n < max:
        yield 2 ** n
        n += 1
a = PowTwoGen(3)
print(next(a)) # 1
print(next(a)) # 2
print(next(a)) # 4
print(next(a)) # raises StopIteration
```



True iterators

Python generator: can generate infinite stream

```
def all even():
   while True:
       yield n
       n += 2
print(next(a)) # 0
print(next(a)) # 2
print(next(a)) # 4
print(next(a)) # 6
print(next(a)) # 8
print(next(a)) # 10
```



First-class functions



Iteration with first-class functions

```
(define uptoby
  (lambda (low high step f)
    (if (<= low high)</pre>
        (begin
           (f low)
           (uptoby (+ low step) high step f))
        '())))
(let ((sum 0))
  (uptoby 1 100 2
           (lambda (i)
             (set! sum (+ sum i))))
  sum)
                                      2500
```



- Recursion vs Iteration efficiency
- naïve implementation of recursion is less efficient
 - time and space needed for subroutine calls
- the language can generate fast code for recursion
- Tail recursion
 - no computation after the recursive call
 - as fast as iteration

```
int gcd(int a, int b) {  /* assume a,b > 0 */
   if (a == b) return a;
   else if (a > b) return gcd(a-b, b);
   else return gcd(a, b-a);
}
```





- Tail recursion
 - can be implemented without the stack allocations
 - a good compiler can recast the recursive function as:

```
int gcd(int a, int b) {  /* assume a,b > 0 */
start:
   if (a == b) return a;
   else if (a > b) { a = a-b; goto start; }
   else { b = b-a; goto start; }
}
```



- Scheme
- Recursive summation

- Scheme
- Tail recursive summation

Eliminate st (subtotal)

```
(define sum3
  (lambda (f low high)
      (sum2 f low high 0)))
```



- Careless recursion can be very bad
- Exponential

```
def fib1(n):
    if n == 0 or n == 1:
        return 1
    return fib1(n-1) + fib1(n-2)
```

Linear

```
def fib2(n):
    f1 = f2 = 1
    for i in range(n-1):
        f1, f2 = f2, f1 + f2
    return f2
```



- Evaluation order (of subroutine arguments)
- Applicative: evaluate before passing
 - used by most languages
- Normal-order: pass unevaluated; evaluate when needed
 - lazy evaluation
 - short-circuit evaluation
 - macros
 - Scheme: used for infinite data structures
 - lazy data structures





- Example: Scheme lazy (infinite) data structures
 - delay a promise
 - force forces evaluation

```
(define naturals
  (letrec ((next (lambda (n) (cons n (delay (next
(+n 1))))))
    (next 1)))
(define head car)
(define tail (lambda (stream) (force (cdr
stream))))
(head naturals)
(head (tail naturals))
                              => 2
(head (tail (tail naturals)))
                               => 3
```