Chapter 6 -- Control Flow

- order of operations as a program executes
  - sequential (sequencing)
  - unstructured (e.g. goto, typically in assembly)
  - selection (aka alternation)
  - iteration
  - subprogram/procedural (Chapt 9)
  - recursion
  - concurrency/parallel (Chapt 13)
  - exception handling (Chapt 13)
  - speculation (Chapt 13)
  - nondeterminacy

- Expression Evaluation
  - operators (e.g. +, - %, ...)
  - operands (aka arguments)
  - Notations:
    - prefix: Op a b or op(a,b) or (op a b)
    - infix: a Op b
    - postfix: a b Op

Control Flow (page 2)

- Expression Evaluation (cont)
  - Parenthesis group operators and operands
    - Lisp: (* (+ 1 3) 2)
    - ML: max (2+3) 4 ;;
    - Smalltalk(Mixfix): myBox displayOn: myScreen at: 100@50
  - Precedence and Associativity
    - a + b * c ^ d ^ e / f ? order of operations?
    - results differ based on order (precedence)
    - a + b - c + b - d ? order of operations?
    - results can differ based on order (associativity)
  - Expression issues:
    - Pascal: if a < b and c < d then ...
      - is: a < (b and c) < d
    - Fortran: 4**3**2
      - 262144 -- right association
    - Ada: 4**3**2
      - syntax error -- no association, must provide ()s
- Standard math associativity:
  - +, -, *, /, % -- left to right
  - ** (or ^) -- right to left

Precedence from book

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (abs not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mo
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (un
		<<, >> (left and right bit shift)	+, - (bin & (conca
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /= , <
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	

Control flow (page 4)

- Some languages allow programmer precedence and associativity
  - Haskell: infixr 8 ^
  - right-to-left, next highest precedence
  - infixl, infix, precedence levels from 0 to 9 (highest)
- Assignment operator in expressions (like C: a = b = c+d;
  - typically right to left
- assignment produces a side effect vs pure expression based language
- pure functional have no side effects, e.g. an expression will always generate the same value
  - referentially transparent.
- in an imperative language, variables can change value so an expression may have different values
  - computation by side effect

Languages in different classes

- Pure Functional: Haskell, Miranda, and some other obscure languages
- Mostly Functional: ML, Lisp, Erlang, and a bunch of other languages
- Mostly Imperative with functional features: C#, Scala, Python, Ruby, ...

References and Values

- Assignment appears simple ... but there are some issues
  - L-value vs R-value
  - a = b; // a is an L-value, b is an R-value
  - a = b + c; // a is an L-value, b is an R-value
  - lvalue = rvalue;
  - a complicated expression can be an l-value: (f(a)+3)->b[c] = 2; // in C
  - context defines l-value/r-value: t = a; a = b; b = t; // C
  - Differences:
    - Pascal & Clu: b := 2; c := b; a := b + c;
    - Pascal -- box model -- copies data (value model)
    - Clu -- pointer model -- copies pointers (reference modes)
  - Java uses value model for built-in types and reference model for classes
  - C# allows choice for user defined, class is reference, struct is a value
  - Reference model -- r-value needs a dereference, either implicit or explicit
    - Has issues if built-in types are value, classes are reference
    - Java: wrapper class to insert integers into hashtable collection object ...
      - recent C# and Java 5+ do automatic boxing

Orthogonality of features ...

- features can be used in "any" combination
- algol 68 -- designed for orthogonality
  - every statement has a value (no "void" functions)
  - a := if b < c then d else e;
  - a := begin f(b); g(c); end;
  - g(d); 2+3;
- C not quite similar but both allow assignment in expressions
  - C’s problem: = vs ==
  - many bugs due to this feature
  - Some languages use := for assignment to avoid this
  - (My ATL/X for CSCI 450 uses <-- for assignment.)
- Issue for imperative languages ... they depend on side effects
  - Often update variables
  - a <-- a + 1
  - b.c[i].d <-- b.c[i].d * f;
  - a[f(i)] <-- a[f(i)] + 1; vs j <-- f(i); a[j] = a[j] + 1;
  - algol 68, C ,... use OP=, like a+= 1;
  - prefix/postfix increment/decrement ++A, B--
    - need a proper definition of sequence of operations (precedence)
    - *p++ = *q++;

Side effects and assignment

- Clu, ML, Perl, Python, ...   a, b = c, d;
  - swap: a, b = b, a; (order of operations again important)
  - a, b, c = f(d, e, f); // returns a tuple

- Initialization
  - default value vs none
    - C global variables default value.
    - Different languages provide different rules
  - aggregate initialization -- many languages
  - floating point NaN value?
  - Issue of catching use of uninitialized variables for other types
  - Compiler static checking of uninitialized variable use
  - Many Object-oriented languages have constructors (static and dynamic objects)

- Operator ordering --
  - precedence, associativity -- some ordering specified ... but
  - a - f(b) - c * d
    - what if f(b) sets a? c*d before a-f(b)?
  - function call: f(a, g(b), h(c))
  - How does optimization change this?
  - Many languages leave ordering as "undefined"
  - But may be require compiler to obey parenthesis ...

More ordering

- Some languages allow for re-ordering based on math

  a = b + c;   d = b + r + c; AS a = b + c; d = a + r;

- Issue with: a - c + d where a + d overflows
  - Should it be checked or not? Depends on language
- reordering an numeric stability .. real numbers
  - adding small numbers first may change result.
- short circuit evaluation
  - a && f(b) -- does f(b) have needed side effects?
  - p = list; while (p && p->key != val) p = p->next;
    - doesn't work in Pascal ... uses full evaluation
  - can be used to avoid out of bounds also ...
  - Some languages offer: "or" and "or else", "and", "and then" (Ada)

Control Flow

- Assembly language ... conditional and unconditional jumps -- unstructured
- Early languages: unstructured also

  if (A .lt. B) goto 10 ! Fortran (and basic also)

  ...

  10: code

  - Dijkstra -- "Goto Letter" CACM 11, 3 (March 1968) 147-148
- Alternatives: "structured programming", modular development, stepwise refinement
  - sequencing, selection, iteration
  - Done by algol 60 first: if/then/else, for, while
  - Case/switch in Algol W
  - repeat/until
- Goto mostly limited to inside a function / not in the language
- Functions/procedures have returns
  - return multi levels in nested routines?
  - historic languages allowed gotos to scope visible labels (Algol 60, PL/1, Pascal)
  - would require unwinding of the stack
  - How about passing in a label and being able to go to that one?
  - C "library solution" setjump/longjump.

Errors and other exceptions

- Deep return more often happens in an error condition
- Some languages provide an exception mechanism
  - error return via different path
  - needs to unwind stack
  - try {..} catch ... typical
  - more later
- Continuations
  - generalization of return to a reference environment
  - Scheme and Ruby do it, implemented with a closure
  - Simple Ruby: cont.ruby (src)
  - Complex Ruby: cont2.ruby (src)
    - See why a Fiber is preferred (light weight cooperative concurrency)
  - Can build many things with this:
    - gotos, midloop exits, multilevel returns, exceptions, call-by-name parameters, coroutines (Fibers)

Sequencing

- controlling the order of execution (imperative, assignments)
- a ; b; c;
  - a is done before b, b before c. issue: subprogram with a side effect
  - block of code: can be in begin/end or {...} aka "compound statement"
  - algol 68 and others, value of compound statement is last "statement"
  - Common Lisp, choice by programmer
  - sequencing is "useless" if no side effects may occur
  - if functions can’t have side effects (Euclid, Turing) sequencing may be changed

Selection

- if ... then ... else
  - dangling else (in some languages, Algol 60, Pascal, ...)
  - some languages have elsif keyword
  - lisp has similar

```
(cond
   ((= A B)
     (...))
   ((= A C)
     (...))
   ((= A D)
     (...))
   (T
     (...)))
```

Selection (Page 2)

- Chort-circuit conditions
  - Some language implicit (C) and some explicit (Ada)
- case/switch statements (aka computed goto)
  - replaces if/then/elsif/elsif/elsif/.../else
- Various versions
  - single value/single statement
  - multiple value/single statement
  - multiple value/break
  - default / otherwise
  - range cases
  - implementation varies: if/then, jump tables, combo
    - added to allow jump tables, faster implementation

Iteration

- Way to perform similar operations (so is recursion)
  - allows for more than fixed sized tasks
  - imperative tends to use iteration, functional tends to use recursion
- Loops are typically executed for their side effects.
- Two primary types: logic controlled, enumeration controlled

Iteration (page 2)

- Logic:
  - while e do s;   // may never execute s
  - repeat s while e; Or repeat s until e; // always executes s
- Enumeration:
  - Python: for e in mycollection (mycollection iterable type)
  - Fortran 90: do i = 1, 10, 2 .... enddo
  - Algol 60: for i := 1 step 2 until 10 do ...
  - number of times through loop defined at compile time
  - Pascal: for V := e1 to / downto e2 do s
    - e1 and e2 evaluated once before loop, not fixed at compile time
    - setp is only by one or by -1
    - for x in set_expr do s
  - infinite loop problem with overflow
  - Note: C is logic: for (e1; e2; e3) s -> e1; while(e2) { s; e3 }
- Issues:
  - loop entry, loop exit?
    - Fortran jump to label, exit via break/exit
  - can loop body modify "control variable"
  - can loop body modify termination condition
  - is control variable available outside loop
  - restarting the loop out of order, continue

Iteration (page 3)

- Iterators
  - general form: function that yeilds many values, one for each "call"
  - Can also generate a collection
    - Python: for i in range(first, last, step)
  - chapel iterator:

```
iter fib(n: int) {
  var (current, next) = (0, 1);
  for 1..n {
    yield current;
    (current, next) = (next, current + next);
  }
}
```

- use:

```
write("First few Fib numbers are: ");
for iv in fib(10) do
   write(iv, ", ");
writeln("...");
```

- Collections can be data structures ... not as obvious how to iterate
  - book shows a python binary tree class with an iterator
  - also shows a java one, different techniques

Iteration (page 4)

- Object iterators have a class initialization and a next method
  - object keeps state
- C++ 11 allows ++ operator on an iterator as the next with iterator as a pointer
- read book on scheme, ruby and smalltalk iterators
- regular functions can simulate iterators
- Other looping constructs
  - for (;;) { ... } // the "official" way to do an infinite loop in C, exit via "break"
  - ada: label: loop ..... exit label when ... end loop label;

Recursion

- Some languages (Fortran 77, early basic, ...) do not allow recursion
- Some functional languages do not allow iteration
- Most languages now have both iteration and recursion
- Use may mostly be matter of taste
  - and how the problem is presented:
    - sum 1 <= i <= 10 f(i)
    - gcd(a,b):
      - a, if a = b
      - gcd(a-b,b), if a > b
      - gcd(a, b-a), if b > a

Recursion (page 2)

- Can use either for these problems
- recursion is often the default for functional languages ... no variables need to be set
- Standard recursion
  - evaluate arguments at call
  - new activation record (on stack)
- tail recursion can be easily converted into iteration
- gcd example:
  - recursive:

```
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);
}
```

  - compiler done iterative solution removing tail recursion

```
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; }
}
```

Recursion (page 3)

- Programmer can write code that uses tail recursion to allow compiler to use iteration
- Recursive solutions are not necessarily algorithmically inferior
  - Some recursive functions cost a lot: e.g. fib()

```
fib x: if x == 0 || x == 1 return 1;
        else return fib(x-1) + fib(x-2);
```

  - exponential solution when sequential possible (see chapel iterator)
  - But, it is possible to do with tail recursion (which is O(n))

Parameters: Applicative- and Normal-Order evaluation

- assumption: parameters (arguments) are evaluated before passing to a subprogram
  - Applicative-order ...
- Normal-Order: passing some representation of the argument for later evaluation
  - Macros do Normal-Order
  - Short Circuit boolean evaluation is also Normal-Order
    - Only evaluated if needed
  - name parameters: passes two representations: lvalue thunk, rvalue thunk
  - some language designers ignore beneficial semantics due to "implementation cost"
  - better languages may trade a bit of speed for better semantics
  - Haskell and Miranda are side-effect free and use normal-order (lazy) evaluation for all parameters

Lazy Evaluation

- Most imperative languages use applicative-order
- In some cases, normal-order can lead to faster code
  - in some cases (like short circuit code) will never evaluate an argument
- Haskell uses normal-order by default
- Scheme has functions called delay and force
  - implements lazy evaluation
    - with no side effect, same as normal-order
    - keeps track of which arguments have been evaluated
    - if needed more than once, evaluates only once
    - A delayed expression is sometimes called a promise
    - lazy data structure -- fleshed out on demand
- Algol 60 subroutine headers indicate type of parameter Applicative or normal

Nondeterminacy and other flow control

- some languages allow a method of non-determinacy
- Chapel: foreach i in 1 ... 100 ; s
  - Sequential chapel -- not known, s must work for any order
  - Parallel chapel -- s can be done in parallel, even on different machines
  - foreach l in locales ; f()
    - runs f() in parallel on all machines