

# CS655 class notes

Raphael Finkel

October 13, 2009

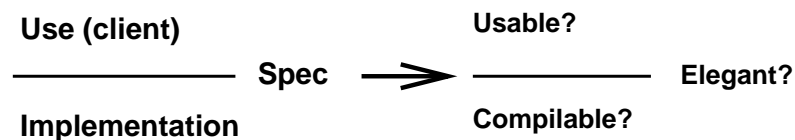
## 1 Intro

Lecture 1, 8/27/2009

- Handout 1 — My names
  - Mr. / Dr. / Professor / —
  - Raphael / Rafi / Refoyl
  - Finkel / Goldstein
- Plagiarism — read aloud from handout 1
- Assignments on web and in handout 1.
- E-mail list: cs655001@cs.uky.edu; instructor uses to reach students.
- All students will have MultiLab accounts, although you may use any computer you like to do assignments.
- textbook — all homework comes from here
- First assignment (Chapter 1, due Thursday)

## 2 Software tools

A programming language is an example of a **software tool**.



### 3 McLennan's Principles (elicit first)

### 4 Algol-like languages: review

- First generation: Fortran
  - constructs based on hardware
  - lexical: linear list of statements
  - control: sequence, **goto**, **do**, subroutines using reference parameter mode
  - data: arithmetic, including complex; arrays with a 3-d limit
  - name: separate scopes; common area
- Second generation: Algol 60
  - lexical: free format; keywords
  - control: nested; **if**, **while**, **for**, but baroque; subroutines with value and name parameter modes.
  - data: generalized arrays, but no complex
  - name: nested scopes with inheritance and local override
- Third generation: Pascal (return to simplicity)
  - data: user-defined types; records, enumerations, pointers
  - control: subroutines with value and reference parameter modes; **case** statement
- Fourth generation: Ada (abstract data types)
  - lexical: bracketed syntax
  - data: modules with controlled export; generic modules
  - control: concurrency with rendezvous
- Fifth generation: Other directions
  - dataflow
  - functional. We will study ML and Lisp.
  - object-oriented. We will study Smalltalk.
  - declarative (logic). We will study Prolog.

## 5 Block structure

- Introduced in Algol.
- A block is a nestable name scope.
- Identifiers can be local, nonlocal, or global with respect to a block.
- Nonlocal identifiers: the language must define whether to
  - inherit (typically allowed if there is no conflict)
  - override (typically true if there is a conflict)
  - require explicit import and export
- At **elaboration time**, constants get values, dynamic-sized types are bound to their size, space is allocated for variables.
- Definition: the **non-local referencing environment (NLRE)** of a procedure or block of code is the binding of non-local identifiers (typically variables, but also constants, types, procedures, and labels) to values.
- **Deep binding**: The NLRE of P is determined (bound) at the time that P is elaborated (and is the RE of the elaborating scope).
- **Shallow binding**: The NLRE of P is determined at the time that P is invoked (and is the RE of the calling scope).
- Adequately difficult example: book 24:21

## 6 Theme: binding time

- There is a range from early to late.
  - language-definition time (example: the fact that constants exist)
  - compile time (example: values of constants in Pascal) We call compile-time bindings **static**.
  - link time (example: version of `printf` in C)
  - elaboration time (example: value of **final int** in Java)
  - statement-execution time (example: value of **int** variable) We call execution-time bindings **dynamic**.
- Early binding is most efficient.
- Late binding is most capable.

## 7 Imperative languages

- Imperative languages involve statements that modify the current state by changing the values of variables.
- A **variable** is an identifier bound (usually statically) to a type, having a value that can change over time. The **L-value** of a variable is the use of a variable on the left side of an assignment (think of “address”); the **R-value** of a variable is its use on the right side (think of “current value”).
- A **type** is a set of values, associated (mostly statically) with operations defined on those values. Type **conversion** means expressing a value of one type as a value of another type.
  - **coercion**: implicit conversion
  - **cast**: explicit conversion
  - **non-converting cast**: rarely needed. **qua** operator of Wisconsin Modula, **reinterpret\_cast<>** of C++.
- An **operation** is a function or an operator symbol as shorthand. It can be heterogeneous.
  - operators have **arity** (example: unary, binary), precedence, associativity
  - operators may be infix (+), prefix (unary -), postfix (->)
  - operators may have short-circuit (lazy) semantics
- An operation is **overloaded** if its identifier or operator symbol has multiple visible definitions. Overloading is resolved (usually statically) by arity, operand types, and return type. Overloading resolution can be exponentially expensive.
- A **primitive type** has no separately accessible components. Examples: integer, character, real, Boolean.
- A **structured type** has separately accessible components. Examples: pointer (dereference), record (field select), array (subscript), disjoint union (variant select). An **associative array** is an array whose index type is string.
- A **constant** is like a variable, but it has no L-value and an unchanging R-value.

## 8 Parameters

- Nomenclature
  - **Formal parameter:** the identifier that the procedure uses to refer to the parameter; it is elaborated when the procedure is invoked.
  - **Actual parameter:** the expression that computes the value of the parameter; it is evaluated in the environment of the caller.
  - **Linkage:** The machine-oriented mechanism by which the caller A causes control to jump to the called procedure B, including initializing B's stack frame, passing parameters, passing results back to A, and reclaiming B's stack frame when it has completed.
- Parameter-passing modes
  - **value mode:** The formal has its own L-value, initialized to the R-value of the actual. The language design may restrict the formal to read-only use. Value mode is the only mode available in C.
  - **result mode:** The formal has its own L-value, not initialized. When B returns, the formal's L-value is copied back to the actual (which must have an L-value, so the actual cannot be an arbitrary expression). Result mode was introduced in Algol-W.
  - **value-result mode:** The formal has its own L-value, initialized to the R-value of the actual. As B returns, its value is copied back to the actual (which must have an L-value). Value-result mode was introduced in Algol-W.
  - **reference mode:** The formal has the same L-value as the actual (which must have an L-value, which might be a temporary location). The language may allow the programmer to specify read-only use of the formal parameter. Reference mode is the only mode available in Fortran.
  - **name mode:** All accesses to the formal parameter re-evaluate the actual (either for L-value or R-value, depending on the access to the formal). This evaluation is in the RE of the caller, typically by means of a compiled procedure called a **thunk**. Name mode was invented for Algol-60 and never used again.

- **macro mode:** The formal parameter is expanded as needed to the text of the actual parameter, which by itself need not be syntactically complete. No modern language uses this mode.

## 9 Iterators

Lecture 2, 9/1/2009

- Iterators allow us to generalize **for** loops.
  - The control variable of the **for** loop ranges over a set of values generated piecemeal by an **iterator**. [book 39:9-10](#).
  - The iterator is like a procedure, taking parameters and returning values of a specified type.
  - The iterator uses a **yield** statement to return a value, but it maintains its RE (and its program counter) in order to continue on demand from the **for** loop.
  - A useful language-supplied iterator is `int upto(low, high)`, which yields all the values in the specified range.
- Iterators are especially useful for generating combinatorial structures.
  - Algorithm for generating all binary trees of  $n$  nodes: [book 41:11](#)
  - Same thing in Python:

```
def binGen(size):
    if size > 0:
        for root in range(size):
            for left in binGen(root):
                for right in binGen(size - root - 1):
                    yield("cons(" + left + "," + right + ")")
    else:
        yield "-"

for aTree in binGen(3):
    print aTree
```

- Trace of `binGen(3)`. [Lecture 3, 9/8/2009](#)
- Another example: yield all nodes in a tree (in pseudo-Python)

```
def treeNodes(tree):
    if tree != null:
        for element in treeNodes(tree.left):
            yield element
        yield tree.value
        for element in treeNodes(tree.right):
            yield element
```

- Wouldn't it be nice to have a **yieldall** construct:

```
def treeNodes(tree):
    if tree != null:
        yieldall treeNodes(tree.left):
        yield tree.value
        yieldall treeNodes(tree.right):
```

This construct might be able to use shortcuts to improve efficiency.

## 10 Macro package to embed iterators in C

- Macros are **IterSTART**, **IterFOR**, **IterDONE**, **IterSUB**, **IterYIELD**.
- Usage: [book 48:14](#)
- Implementation
  - set `jump` and `longjmp` for linkage between **for** and the controlling iterator, between **yield** and its controlled loop.
  - Padding between stack frames to let `longjmp( )` be called without harming frames higher on the stack. Three integers is enough in Linux on an i686.
  - A `Helper` routine to actually call the iterator and act as padding.
  - The top frame must be willing to assist in creating new frames.

## 11 General coroutines — Simula 67

- Problem: binary-tree node-equality test in symmetric order
- Solution
  - Independently advance in each tree [book 38:8](#)

- [Lecture 4, 9/10/2009](#)
- Each coroutine has its own stack; main has its own stack.
- All the stacks are joined via static-chain pointers into a cactus stack.
- A scope must not exit until all its children exit, or we must use a reference count. This is an example of the dangling NLRE problem.
- Syntax (Simula 67)
  - Records have initialization code that may execute **detach**.
  - Another coroutine may resume it via explicit **call**.
- Ole-Johann Dahl, designer of Simula 67, got the Turing award in 2001.

## 12 Power loops

- How can you get a dynamic amount of **for**-loop nesting?
- Application:  $n$  queens [book 57:29](#)
- Usual solution: single **for** loop with a recursive call.
- Cannot use that solution in Fortran, which does not allow recursion.
- Solution: Power loops. [book 57:28](#)
- Implementation: Only needs branches, no recursion. [book 59:31](#)
- How general is this facility?
- Do power loops violate principle 20?

## 13 IO

- Attempt to strip a programming language of all non-essential elements.
- What's left: **goto** with parameters, hence formal-actual bindings.
- Parameters can be integer, anonymous function, or continuation.
- A function is passed by **closure**: pointer to code, pointer to NLRE.

- A **continuation** is a function whose parameters have already been bound. It is passed by a closure along with the parameters.
- Examples `book 50:15 and following`
- `Lecture 5, 9/17/2009`

## 14 Dimensions

- `Lecture 6, 9/22/2009`
- `book 70:7`
- Applies to reals.
- Can be embedded into a strong type system.

## 15 ML

- ML is **functional**, but we are particularly interested in its type system, with these important aspects.
  - The language is **strongly typed**.
  - The compiler **infers** the types of identifiers if possible.
  - **Higher-order types** are easily represented.
  - Types can be **polymorphic**, that is, expressed with respect to type identifiers.
- Examples, starting on `book 81:13`
- `Lecture 7, 9/24/2009`
- `Lecture 8, 9/29/2009`
- How does currying work? `handout on currying`

## 16 Types

- `Lecture 9, 10/1/2009`
- A **type** is a property of an R-value or of an identifier that can hold R-values.

- The property consists of a set of values.
- **Strong typing** means the compiler
  - knows the type of every R-value and identifier
  - enforces type compatibility on assignment and formal-actual binding
    - **compatible** means type equivalent, a subtype, or convertible
- A **subtype** consists of a subset of the values
  - Assignment and formal-actual binding require a dynamic check.
  - Subtype examples: range of integers, subclass of class
  - `subTypeVar := baseTypeVar`
- Type equivalence
  - **structural equivalence**: expand type to a canonical string representation; equivalence is string equality.
    - lax: ignore field names, ignore array bounds, ignore index type, flatten records.
    - pointers require that we handle recursive types (and still build finite strings)
  - **name equivalence**: reduce a type to an instance of a type constructor
    - **type constructors**: **array**, **pointerTo**, **enum**, **struct**, **derived**.
    - lax (**declaration equivalence**): multiple uses of one type constructor are equivalent

## 17 What does polymorphic mean?

People use the term **polymorphic** to mean various features.

- Static procedure overloading with compile-time resolution (Ada, Java). Here, the type of the procedure (its **signature**) determines whether it is a viable candidate for resolving the overloading.
- Dynamic method binding (Java, Smalltalk, C++ deferred binding). Here, the dynamic type (class) of the value (object) determines which procedure (method) to invoke.

- Types described by type identifiers (perhaps with the compiler inferring the types of values) (ML). Here, the dynamic type constraints on the parameter and return value determine the effective type of the function.
- Passing an ADT as a parameter (Russell).
- Generic packages (Ada), templates (C++).

## 18 Unusual first-class values

- Lecture 10, 10/8/2009
- A **first-class value** can be returned from a function. In languages with variables, a first-class value can be stored in a variable.
- A **second-class value** can be an actual parameter.
- A **third-class value** can be used “in its ordinary way”.
- Labels and procedures
  - Usually third class values; the “ordinary way” is in a **goto** statement or a procedure-call statement.
  - They could be second class. They must be passed as a closure. For a label, the closure includes the RE of its target, so the stack can be unwound to the right place. For a procedure, the closure includes its NLRE to resolve non-local references.
  - To make them first class, we still need to build a closure, which we can then store in a variable. But the lifetime of that variable might exceed the life of the RE stored in the closure. This is the **dangling-NLRE problem**. book 76:11
- To resolve the dangling-NLRE problem
  - Let it happen and call the program “erroneous”.
  - Prevent it from happening by restricting the language.
    - Don’t let labels or procedures be first-class: Pascal
    - Don’t let scopes nest, so there is no need for closures (for procedures; labels are still problematic): C
    - Only allow top-level procedures (or labels) be first-class: Modula-2

- Let it happen and make it work: allocate all frames from the heap and use explicit release (reference counts suffice).

## 19 Can types be second or first class?

- Letting a type be second-class is a step toward polymorphism.
  - Second-class types allow generic packages (Ada), templates (C++).
  - However, we only allow types to be passed at compile time during generic-package instantiation.
  - One can restrict the range of the actual type (the “type of the type”) by a Java-like interface.
- An attempt at first-class types: **dynamic**. book 112:68 Actually, this attempt is an attempt to extend strong typing to dynamic types; it is not the same as making types first-class.
- Another attempt at first-class types: Russell.
  - But what Russell calls a “type” we would call an ADT (abstract data type), which is something like a Java class.
  - So a Russell actually introduces dynamic class definitions.
  - We will see dynamic class definitions in Smalltalk later.

## 20 Haskell

- Haskell is a functional programming language much like ML, with additional features.
- Haskell uses indentation for grouping, much like Python. This design makes programs compact, but harder to read (if routines are long) and very difficult for blind programmers to construct.
- All functions are fully curried.
- Haskell evaluates all expressions **lazily**. Therefore, one can express infinite lists:

```
naturalNumbers = [0 ..]  
ones = 1 : ones
```

- Haskell allows lists to be built using **comprehensions**, which are like sets in Zermelo-Fränkel set theory.

```
squares :: [Int]
squares = [n*n | n <- [1 .. 5] ]
fermat :: [Int]
fermat = [(a, b, c, n) | a<-[3..], b<-[3..],
                       c<-[3..], n<-[3..],
                       ((a^n) + (b^n) == (c^n))]
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (a:rest) =
  quicksort [b | b <- rest, b <= a] ++
  [a] ++
  quicksort [b | b <- rest, b > a]
```

- List comprehensions give us an easy way to implement Lisp's `mapcar` function:

```
mapcar (list, function) = [function a | a <- list]
```

In fact, we can generally avoid using `mapcar` entirely.

- One can combine lazy evaluation, comprehension, and infinite lists to gain memoization (dynamic programming)

```
cache :: [Int]
cache = [fib x | x <- [0 ..]]
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = cache!!(n-1) + cache!!(n-2)
```

- Haskell has **interfaces**, much like Java, which are like types that provide certain operations.

## 21 Lisp

- Lecture 12, 10/13/2007
- Lisp has been very influential and was widely used in AI.

- Lisp is **homoiconic**: programs and data structures have the same form, so one can execute data and one can manipulate program.
- Lisp is **functional** (functions have no side effects, and there are no variables)
  - We need to ignore **set**, **rplaca**, **rplacd**, and a few other non-functional features.
  - **defun** does change the context; introducing new functions always has a side effect.
  - Because it is functional, Lisp allows lazy and eager evaluation.
- Examples (online)
- The metacircular interpreter book 135:31 ff