CS655 class notes

Raphael Finkel

September 9, 2019

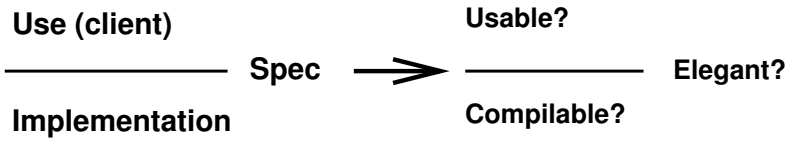
1 Intro

Class 1, 8/26/2019

- Handout 1 — My names
  - Mr. / Dr. / Professor / —
  - Raphael / Rafi / Refoyl
  - Finkel / Goldstein
- Extra 5 minutes on every class? What is a good ending time?
- 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. But your programs must run on MultiLab computers, because that's how they will be graded.
- textbook — all homework comes from here
- Oral assignments are end-of-chapter assignments. Assignment for Chapter 1 exercises (Friday).

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
  - Class 2, 8/28,2019
  - 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
  - name: 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 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.

Class 3, 8/30/2019 Exercises from Chapter 1.

6 Block structure

- Class 4, 9/4/2019
- 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

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
  - Class 5, 9/6/2019
  - **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. For instance, say we have four versions of +, depending on whether they take integers/floats and whether they return integers/floats. Then how do you resolve (a + b) + (c + d)?
- A **primitive type** (or **basic 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. In Java, it's denoted by the modifier **`final`**.

8 Iterators

- 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, using "generators":

```
1 def binGen(size):
2  if size > 0:
3   for root in range(size):
4    for left in binGen(root):
```

```
5      for right in binGen(size - root - 1):
6       yield("cons(" + left + "," + right + ")")
7   else:
8    yield "-"
9
10 for aTree in binGen(3):
11  print aTree
```

- Class 6, 9/9/2019
- Trace of binGen(3).

- Another example: yield all nodes in a tree (in pseudo-Python)

```
1 def treeNodes(tree):
2   if tree != null:
3     for element in treeNodes(tree.left):
4       yield element
5     yield tree.value
6     for element in treeNodes(tree.right):
7       yield element
```

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

```
1 def treeNodes(tree):
2   if tree != null:
3     yieldall treeNodes(tree.left):
4     yield tree.value
5     yieldall treeNodes(tree.right):
```

This construct might be able to use shortcuts to improve efficiency. JavaScript actually has it: `yield*`. It **delegates** the yielding to another iterator.

- Another example: all combinations $C(n, k)$:

```
1 def comb(n,k,start):
2   if k == 0:
3     yield ""
4   elif k+start <= n:
5     for rest in comb(n,k-1,start+1):
6       yield str(start+1) + "," + rest
7     for rest in comb(n,k,start+1):
8       yield rest
```

```
 9
10 for result in comb(6,3,0):
11   print result
```

9 Macro package to embed iterators in C

- Macros are **IterSTART**, **IterFOR**, **IterDONE**, **IterSUB**, **IterYIELD**.
- Usage: book 48:14
- Implementation
  - `setjump` 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.

10 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?