CSC 533: Organization of Programming Languages

Spring 2008

See online syllabus at:
http://dave-reed.com/csc533
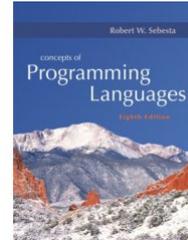
Course goals:

- understand issues in designing, implementing, and evaluating programming languages
- appreciate strengths and tradeoffs of different programming paradigms
- working knowledge of Java, C/C++ & Scheme

Why are there different programming languages?

in theory, all programming languages are equivalent
- compiled/interpreted into basic machine operations
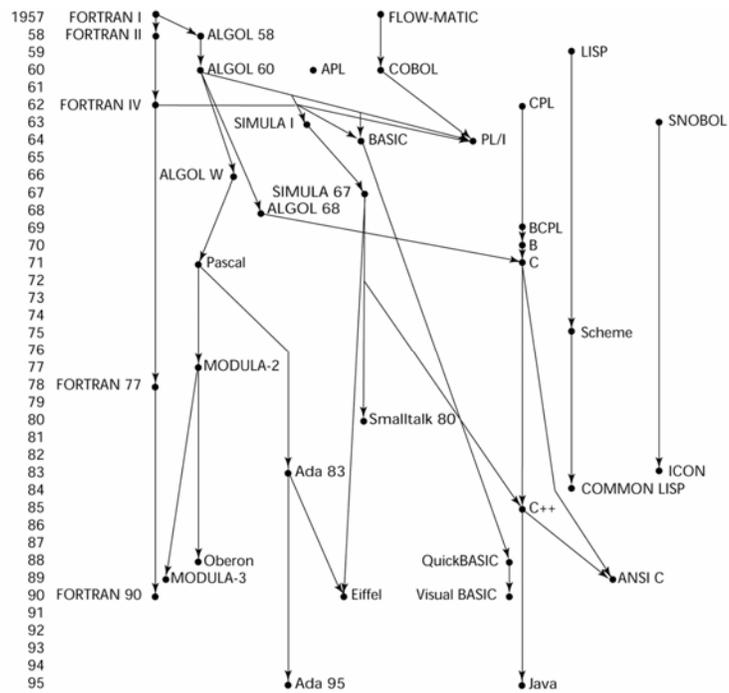- Church-Turing thesis applies

Why are there different natural languages?

in practice, different languages provide distinct voices
- different cultures (application domains)
- different primitive concepts (operations)
- different ways of thinking about the world (perspectives)

Family tree of high-level languages

1957 FORTRAN I
58 FORTRAN II
ALGOL 58
FLOW-MATIC
59
60 ALGOL 60 · APL COBOL LISP
61
62 FORTRAN IV
63 SIMULA I CPL SNOBOL
64 BASIC PL/I
65
66 ALGOL W
67 SIMULA 67
68 ALGOL 68
69 BCPL
70 B
71 Pascal C
72
73
74
75 Scheme
76
77 MODULA-2
78 FORTRAN 77
79
80 Smalltalk 80
81
82
83 Ada 83
84 ICON
85 C++ COMMON LISP
86
87
88 Oberon QuickBASIC
89 MODULA-3 ANSI C
90 FORTRAN 90 Eiffel Visual BASIC
91
92
93
94
95 Ada 95 Java

3

Programming paradigms

similarly, different problem-solving approaches (paradigms) exist and are better suited to different types of tasks

imperative approaches: programs specify sequences of state changes
  block-structured: subroutines & nested scopes (Pascal, C)
  object-based: interacting objects (Ada, Modula)
  object-oriented: objects + inheritance (C++, Java, Smalltalk)
  distributed: concurrency (Occam, Linda)

declarative approaches: programs specify what is to be computed
  functional: functions in lambda calculus (LISP/Scheme, ML)
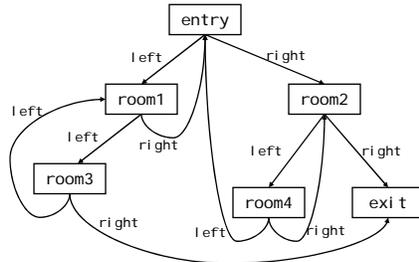 *logic*: relations in predicate calculus (Prolog)
  database: info records and relations (SQL)

4

2

Example: HW1

consider a maze of rooms, with each room connected to others by 2 doors
- note: doors are not necessarily bidirectional

can navigate the maze by making a series of left or right door choices

e.g., starting in entry, left-left path:

entry → room1 → room3

e.g., starting in room2, left-right path:

room2 → room4 → room2

you are to write a Java program that reads in a maze structure from a file, stores it, and repeatedly prompts the user for a room & path
- should then report the room where the player would end up by following the path
- file structure? internal structure? I/O?

5

Example: HW1 in LISP/Scheme

for comparison purposes, HW1 is ideal for Scheme
- Scheme is a functional programming language
- it is symbolic, can represent words and text as easily as numbers
- it has primitives for manipulating lists and structures
- recursion is natural and efficient

```
(define ROOMS
  '((entry room1 room2)
    (room1 room3 entry)
    (room2 room4 entry)
    (room3 room1 exit)
    (room4 entry room2)
    (exit exit exit)))


(define (navigate loc path room-map)
  (cond ((null? path) loc)
        ((equal? (car path) 'L)
         (navigate (cadr (assoc loc room-map)) (cdr path) room-map))
        ((equal? (car path) 'R)
         (navigate (caddr (assoc loc room-map)) (cdr path) room-map))
        (else 'ILLEGAL-PATH)))
```

can compactly represent the maze as a list

can write the basic code in 7 lines (ignoring file I/O)

6

3

Why study programming languages?

increased capacity to express ideas
- broader perspective on programming and problem solving, new paradigms

improved background for choosing appropriate languages
- know tradeoffs between different languages
- simplify programming task by choosing best-suited language

increased ability to learn new languages
- as with natural languages, 2nd languages is hardest to learn
- languages come and go, must be able to adapt/adjust

better understanding of the significance of implementation
- can use language more intelligently if understand implementation
- also aids in identifying bugs

increased ability to design/implement new languages

7

How do we judge a programming language?

readability
in software engineering, maintenance cost/time far exceeds development cost/time
→ want code to be easy to understand, modify, and maintain

- *simplicity:* language should be as small as possible, avoid redundant features
  - ➤ C++ is pretty complex, e.g., `x++;`   `++x;`   `x +=1;`   `x = x + 1;`
  - ➤ Java slightly better (some features removed); Scheme is very simple

- *orthogonality:* small set of primitive constructs, can be combined independently and uniformly (i.e., very few special cases)
  - ➤ C++ is OK but many exceptions, e.g., functions can return structs, not arrays
  - ➤ Java slightly better; Scheme is highly orthogonal

- *natural control and data structures:* provide useful, high-level abstractions
  - ➤ C++ is good but does include some tricky ones, e.g., `:?`, `goto`
  - ➤ Java comparable (but no `goto`); Scheme is limited (e.g., recursion for looping)

- *simple and unambiguous syntax:* intended form of code is clear to reader
  - ➤ C++ not so good, e.g., overall complexity, dangling else, overused `static`
  - ➤ Java slightly better; Scheme syntax is simple and clear

8

4

How do we judge a programming language (cont.)?

writability

want to make programming as simple as possible for the programmer

- *simplicity + orthogonality + natural control & data structures + simple & unambiguous syntax*

- *support for abstraction:* need to be able to define and utilize abstractions
  - C++ is good, e.g., support for functions, libraries, classes
  - Java is comparable; Scheme is OK, but more tedious

- *expressivity:* language provides convenient ways of specifying computations
  - C++ is good, e.g., if & switch, while & do-while & for, bitwise operators, …
  - Java is slightly less (removes low-level); Scheme is not very expressive (few control structures)

note: readability & writability are often at odds

e.g., more control structures can simplify programmer's task, but make code harder to read and maintain (more to know, more redundancy)

Common LISP vs. Scheme

9

How do we judge a programming language (cont.)?

reliability

want to ensure that a program performs to its specifications under all conditions
  - →want to build in strong error checking and recovery capabilities
  - →also want to help the programmer to avoid errors

- *readability + writability*

- *type checking:* identify errors either at compile-time or during execution
  - C++ is pretty good, e.g., most types checked at compile time, some loopholes
  - Java is slightly better; Scheme is dynamic, must do checking during execution

- *exception handling:* ability to intercept and recover from errors in code
  - C++ is OK (try/catch, but not always used)
  - Java is slightly better (libraries require exception handling), Scheme is more awkward

- *memory management:* control memory accessing, allocation/deallocation, aliasing
  - C++ is *dangerous*, e.g., can access specific addresses, must deallocate memory
  - Java is better (memory mgmt is automatic); Scheme handles all transparently

10