Lecture 1: Overview

CSC 131
Spring, 2019

Kim Bruce

Do Languages Matter?

- Why choose C vs C++ vs Java vs Python ...
- What criteria to decide?
- Scenarios:
  - iOS app
  - Android App
  - Web App
  - Mac App
  - Windows app
  - System software
  - Scientific App
  - Scripting

Do Languages Matter?

- Impact on programming practice
- SIGPLAN Education Board documents

Provide Abstractions

- Data Abstractions:
  - Basic data types: ints, reals, bools, chars, pointers
  - Structured: arrays, structs (records), objects
  - Units: Support for ADT's, modules, packages
- Control Abstractions:
  - Basic: assignment, goto, sequencing
  - Structured: if...then...else, loops, functions
  - Parallel: concurrent tasks, threads, message-passing

PL's & Software Development

- Development process:
  - requirements
  - specification
  - implementation
  - certification or validation
  - maintenance
- Evaluate languages based on goals

Goals of Some older PL's

- Languages & their goals:
  - BASIC - quick development of interactive programs
  - Pascal - instruction
  - C - low-level systems programming
  - FORTRAN, Matlab - number-crunching scientific
- What about large-scale programs?
  - Ada, Modula-2, object-oriented languages

PL Choice

- Languages designed to support specific software methodologies.
- Language affect way people think about programming process.
- Hard for people to change languages if requires different way of thinking about process.
  - Easier to make switch when younger!

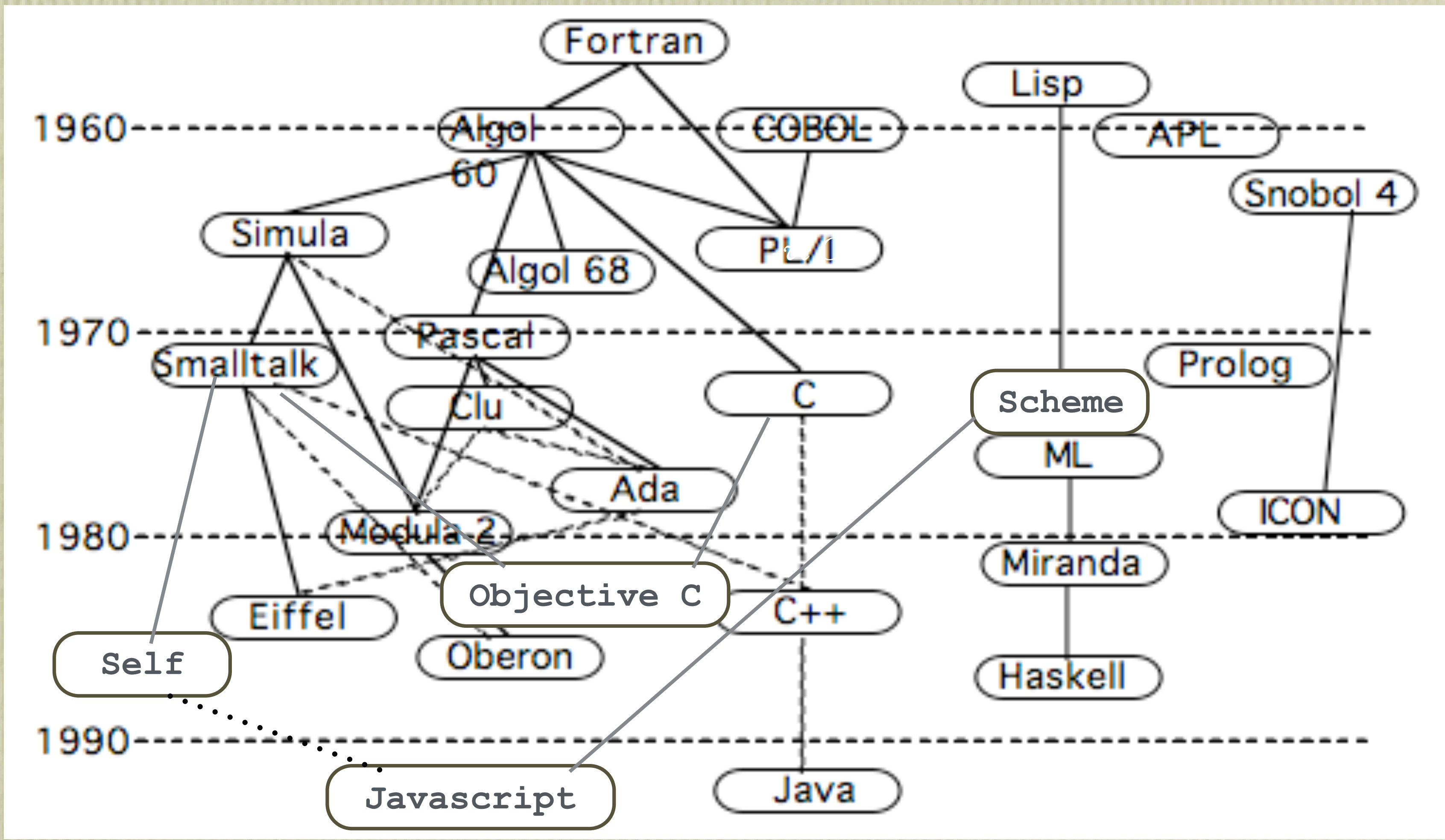
Paradigms
or whatever you want to call them

- Not crisp boundaries
  - Procedural
  - Functional
  - Logic or Constraint-programming
  - Object-oriented

History of PL's

- Machine language
  - ⇒ Assembly language
  - ⇒ High-level language
- Single highly-trained programmer
  - ⇒ Teams of programmers

History of PLs



Newer: Scala, Dart, Rust, NewSpeak, Swift, Grace, Pyret

Extreme Languages

- APL *(Used at Pomona in 1970's)*
  - Everything is a vector
  - SD←((+/((X - AV←(T←+/X)÷ρX)*2))÷ρX)*0.5
  - calculates average (AV) and standard deviation of X
- COBOL
  - Calculate largest number

```
WORKING-STORAGE SECTION.
  77 A PIC 9(4).
  77 B PIC 9(4).
  77 C PIC 9(4).
  77 LARGE PIC 9(4).
\PROCEDURE DIVISION.
  ACCEPT-PARA.
    DISPLAY "ENTER THREE NUMBERS".
    ACCEPT A.
    ACCEPT B.
    ACCEPT C.
  COMPUTE-PARA.
    IF A>B AND A>C THEN MOVE A TO LARGE.
    IF B>C AND B>A THEN MOVE B TO LARGE.
    IF C>B AND C>A THEN MOVE C TO LARGE.
  DISPLAY-PARA.
    DISPLAY "LARGEST NUMBER=" LARGE.
STOP RUN.
```

Course Goals

- Upon completion of course should be able to:
  - Quickly learn programming languages, & how to apply them to effectively solve programming problems.
  - Rigorously specify, analyze, & reason about the behavior of a software system using a formally defined model of the system's behavior.
  - Realize a precisely specified model by correctly implementing it as a program, set of program components, or a programming language.

Course Goals

- *Plus*:
  - Understand the principal underlying differences in program languages, why those differences occur, and how that affects the semantics of the languages.
  - Understand contemporary trends in the design of programming languages.
  - Understand the run-time behavior of programs, especially as it relates to memory management using the run-time stack and heap.

Administrivia

- Web page at
  - http://www.cs.pomona.edu/classes/cs131/
- Text by Mitchell:
  - Free!
  - Use some revised chapters: Haskell instead of SML
- If needed, get account from Corey LeBlanc

Administrivia

- Homework
  - Generally due every week on Thursday night.
    - Posted on Friday
  - All homework must be turned in electronically
    - Use LaTeX'ed, but can scan in pictures
    - *... but must be legible!!*

On-Line Discussions

- Will be on Piazza
- You will receive an invitation later this week.
  - Do not throw it away!
- You can ask and answer questions on-line.
  - TA's and I will monitor and respond.

Course Outline

- Functional programming (Haskell)
  - Good example of lazy functional language
  - use in implementing parsers, interpreters, etc.
- Lambda calculus
  - Simple model of language, easier to work on theory
- Implementing parsers/interpreters

Course Outline (continued)

- Run-time behavior of programs
  - Memory management
- Types and control constructs
- Data abstraction and modules
- Object-oriented languages
- Parallelism/Concurrency

Computability

- Halting Problem in your favorite language:
  - There is no program H that will, for any other program P, always accurately determine whether or not P will halt.
- Rice's Theorem: Any interesting question about programs is undecidable. (Syntax questions aren't interesting.)
- This will place limits on static checking of programs (e.g., type-checking)

Infinity

- How many programs can be written in Java
  - Countably infinite
- How many functions are there from Strings to Strings?
  - Uncountably infinite
  - So most functions are not computable!

Haskell

According to Larry Wall (designer of PERL): ... a language by geniuses for geniuses

He's wrong — at least about the latter part
though you might agree when we talk about monads

Haskell 98

- Purely functional
- Functions are first-class values
- Statically scoped
- Strong, static typing via type inference
  - Type-safe
- Parametric polymorphism
- Type classes

Haskell (cont)

- Rich type system including support for ADT's
- Non-strict (lazy) evaluation
- Imperative features emulated using monads.
- Garbage collection
- Compiled or interpreted.
- Named after Haskell Curry -- early contributor to lambda calculus and combinatory logic

Read Haskell Tutorials

- All on links page from course web page
- I like "Learn you a Haskell for greater good"
- O'Reilly text: "Real World Haskell" free on-line
- Print Haskell cheat sheet
- Use "The Haskell platform", available at
  - http://www.haskell.org/

Using GHC

- to enter interactive mode type: ghci
  - :load myfile.hs   -- :l also works
  - after changes type :reload
  - Control-d to exit
  - :set +t   -- prints more type info when interactive
  - "it" is result of expression
    - Evaluate "it + 1" gives one more than previous answer.

Built-in data types

- Unit has only ()
- Bool: True, False with not, &&, ||
- Int: 5, -5, with +, -, *, ^, =, /=, <, >, >=, ...
  - div, mod defined as prefix operators (`div` *infix*)
  - Int fixed size (usually 64 bits)
  - Integer gives unbounded size
- Float, Double: 3.17, 2.4e17 w/ +, -, *, /, =, <, >, >=, <=, sin, cos, log, exp, sqrt, sin, atan.

More Basic Types

list of Char

- Char: ‘n’
- String = [Char], not really primitive
  - "hello"++" there", length
  - No substring, but `isInfixOf` for all lists *Prefix op w/out ``!*
  - Also ‘isPrefixOf`, `isSuffixOf’ *import Data.List*
- Type classes (later) provide relations between classes.

Interactive Programming with ghci

- Type expressions and run-time will evaluate
- Define abbreviations with "let"
  - let double n = n + n
  - let seven = 7
- "let" not necessary at top level in programs loaded from files

Lists

- Lists
  - [2,3,4,9,12]: [Integer]
  - [] -- empty list
  - Must be homogenous
  - Functions: length, ++, :, map, rev
    - also head, tail, *but normally don't use!*

Polymorphic Types

- [1,2,3]:: [Integer]
- ["abc", "def"]:: [[Char]], ...
- []:: [a]
- map:: (a → b) → ([a] → [b])
- *Use* :t exp *to get type of* exp

Pattern Matching

- Decompose lists:
  - [1,2,3] = 1:(2:(3:[]))
- Define functions by cases using pattern matching:

```
prod [] = 1
prod (fst:rest) = fst * (prod rest)
```

Pattern Matching

- Desugared through case expressions:
  - ```
    head' :: [a] -> a
    head' [] = error "No head for empty lists!"
    head' (x:_) = x
    ```
- equivalent to
  - ```
    head' xs = case xs of
                 [] -> error "No head for empty lists!"
                 (x:_) -> x
    ```

Type constructors

- Tuples
  - (17,"abc", True) : (Integer , [Char] , Bool)
  - fst, snd defined only on pairs
- Records exist as well

More Pattern Matching

- (x,y) = (5 `div` 2, 5 `mod` 2)
- hd:tl = [1,2,3]
- hd:_ = [4,5,6]
  - "_" is wildcard.

Static Typing

- Strongly typed via type inference
  - head:: {a} → a
    tail:: {a} → {a}
  - last {x} = x
    last (hd:tail) = last tail
- System deduces most general type, {a} -> a
  - Look at algorithm later

Static Scoping

- What is the answer?
  - `let x = 3`
  - `let g y = x + y`
  - `g 2`
  - `let x = 6`
  - `g 2`
- What is the answer in original LISP?
  - `(define x 3)`
  - `(define (g y) (+ x y))`
  - `(g 2)`
  - `(define x 6)`
  - `(g 2)`

Static Scoping

- What is the answer?
  - `let x = 3`
  - `let g y = x + y`
  - `g 2`
  - `let x = 6`
  - `g 2`

```
{
  const x = 3
  {
    g(y) = x + y
    {
      print (g 2)
      const x = 6
      {
        print (g 2)
      }
    }
  }
}
```

- What is the answer in original LISP?
  - `(define x 3)`
  - `(define (g y) (+ x y))`
  - `(g 2)`
  - `(define x 6)`
  - `(g 2)`

Local Declarations

```
roots (a,b,c) =
   let      -- indenting is significant
      disc = sqrt(b*b-4.0*a*c)
   in
      ((-b + disc)/(2.0*a),(-b - disc)/(2.0*a))

*Main> roots(1,5,6)
(-2.0,-3.0)
```

or

```
roots' (a,b,c) = ((-b + disc)/(2.0*a),
                  (-b - disc)/(2.0*a))
   where disc = sqrt(b*b-4.0*a*c)
```

Anonymous functions

- dble x = x + x
- *abbreviates*
- dble = \x -> x + x

Defining New Types

- Type abbreviations
  - type Point = (Integer, Integer)
  - type Pair a = (a,a)
- data definitions
  - create new type with constructors as tags.
  - generative
- data Color = Red | Green | Blue

  See more complex examples later

Type Classes Intro

- Specify an interface:
  - ```
    class Eq a where
        (==) :: a -> a -> Bool    -- specify ops
        (/=) :: a -> a -> Bool
        x == y = not (x /= y)     -- optional implementations
        x /= y = not (x == y)
    ```
  - ```
    data TrafficLight = Red | Yellow | Green
    instance Eq TrafficLight where
        Red == Red = True
        Green == Green = True
        Yellow == Yellow = True
        _ == _ = False
    ```

Common Type Classes

- Eq, Ord, Enum, Bounded, Show, Read
  - See http://www.haskell.org/tutorial/stdclasses.html
- data defs pick up default if add to class:
  - data ... deriving (Show, Eq)
- Can redefine:
  - instance Show TrafficLight where

```
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

More Type Classes

- class (Eq a) => Num a where ...
  - instance of Num a must be Eq a
- Polymorphic function types can be prefixed w/ type classes
  - test x y = x < y *has type* (Ord a) => a -> a -> Bool
  - *Can be used w/ x, y of any Ord type.*
- *More later ...*
  - *Error messages often refer to actual parameter needing to be instance of a class -- to have an operation.*

Higher-Order Functions

- Functions that take function as parameter
  - Ex: map:: (a → b) → ([a] → [b])
- Build new control structures
  - ```
    listify oper identity [] = identity
    listify oper identity (fst:rest) =
                        oper fst (listify oper identity rest)
    ```
  - ```
    sum' = listify (+) 0
    mult' = listify (*) 1
    and' = listify (&&) True
    or' = listify (||) False
    ```

Exercise

- Is listify left or right associative?
  - What is listify (-) 0 [3,2,1]? 2 or -6 or 0 or ???
- How can we change definition to associate the other way?

See built-in foldl and foldr

Quicksort

```
partition (pivot, []) = ([],[])
partition (pivot, first : others) =
   let
      (smalls, bigs) = partition(pivot, others)
   in
      if first < pivot
         then (first:smalls, bigs)
         else (smalls, first:bigs)
```

Type is:

```
partition :: (Ord a) => (a, [a]) -> ([a], [a])
```

Quicksort

```
qsort [] = []
qsort [singleton] = [singleton]
qsort (first:rest) =
   let
       (smalls, bigs) = partition(first,rest)
   in
       qsort(smalls) ++ [first] ++ qsort(bigs)
```

Type is:

```
qsort :: (Ord t) => [t] -> [t]
```

Quicksort - parametrically

```
partition (pivot, []) lThan = ([],[])
partition (pivot, first : others) lThan =
  let
    (smalls, bigs) = partition(pivot, others) lThan
  in
    if (lThan first pivot)
      then (first:smalls, bigs)
      else (smalls, first:bigs)

partition ::
        (t, [a]) -> (a -> t -> Bool) -> ([a], [a])

*Main> partition(6,[8,4,6,3])(>)
```

Quicksort

```
qsort [] lt = []
qsort [singleton] lt = [singleton]
qsort (first:rest) lt =
   let
      (smalls, bigs) = partition (first,rest) lt
   in
      qsort smalls lt ++ [first]
                      ++ qsort bigs lt

qsort :: [a] -> (a -> a -> Bool) -> [a]

*Main> qsort [33,66,32,87,999,2](>)
[999,87,66,33,32,2]
```

Recursive Datatype Examples

- data IntTree = Leaf Integer |
  Interior (IntTree,IntTree)
  deriving Show
  - Example values: Leaf 3, Interior(Leaf 4,Leaf -5), ...
- data Tree a = Niltree |
  Maketree (a, Tree a, Tree a)

Binary Search Using Trees

```
insert new Niltree = Maketree(new,Niltree,Niltree)
insert new (Maketree (root,l,r)) =
  if new < root
     then Maketree (root,(insert new l),r)
     else Maketree (root,l,(insert new r))

buildtree [] = Niltree
buildtree (fst : rest) =
                insert fst (buildtree rest)
```

Binary Search Tree

```
find elt Niltree = False
find elt (Maketree (root,left,right)) =
  if elt == root
     then True
     else if elt < root then find elt left
     else find elt right      -- elt > root

bsearch elt list = find elt (buildtree list)
```

Haskell is Lazy!

Lazy vs. Eager Evaluation

- Eager: Evaluate operand, substitute operand value in for formal parameter, and evaluate.
- Lazy: Substitute operand for formal parameter and evaluate body, evaluating operand only when needed.
  - Each actual parameter evaluated either not at all or only once! (Essentially cache answer once computed)
  - Like left-most outermost, but more efficient

Lazy evaluation

- Compute f(1/0,17) where f(x,y) = y
- Computing head(qsort[5000,4999..1]) is faster than qsort[5000,4999..1]
- Compare time of computations of:
  - fib 32
  - dble (fib 32) where dble x = x + x
- Computations based on *graph reduction*
  - *like tree rewriting, except w/computation graphs - sharing*

Lazy Lists

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

complexity $O(fib\ n) \sim O(2^n)$

```
fibList = f 1 1
  where f a b = a : f b (a+b)
fastFib n = fibList!!n
```

complexity $O(n)$

```
fibs = 1:1:[ a+b | (a,b) <- zip fibs (tail fibs)]

primes = sieve [ 2.. ]
       where
         sieve (p:x) = p :
              sieve [ n | n <- x, n `mod` p > 0]
```

Call-by-need

- Efficient implementation of call-by-name (Algol 60)
- If purely functional language then may evaluate expression at most once, because can never change.
- Hence graph instead of tree works!
  - dble(fib 32)