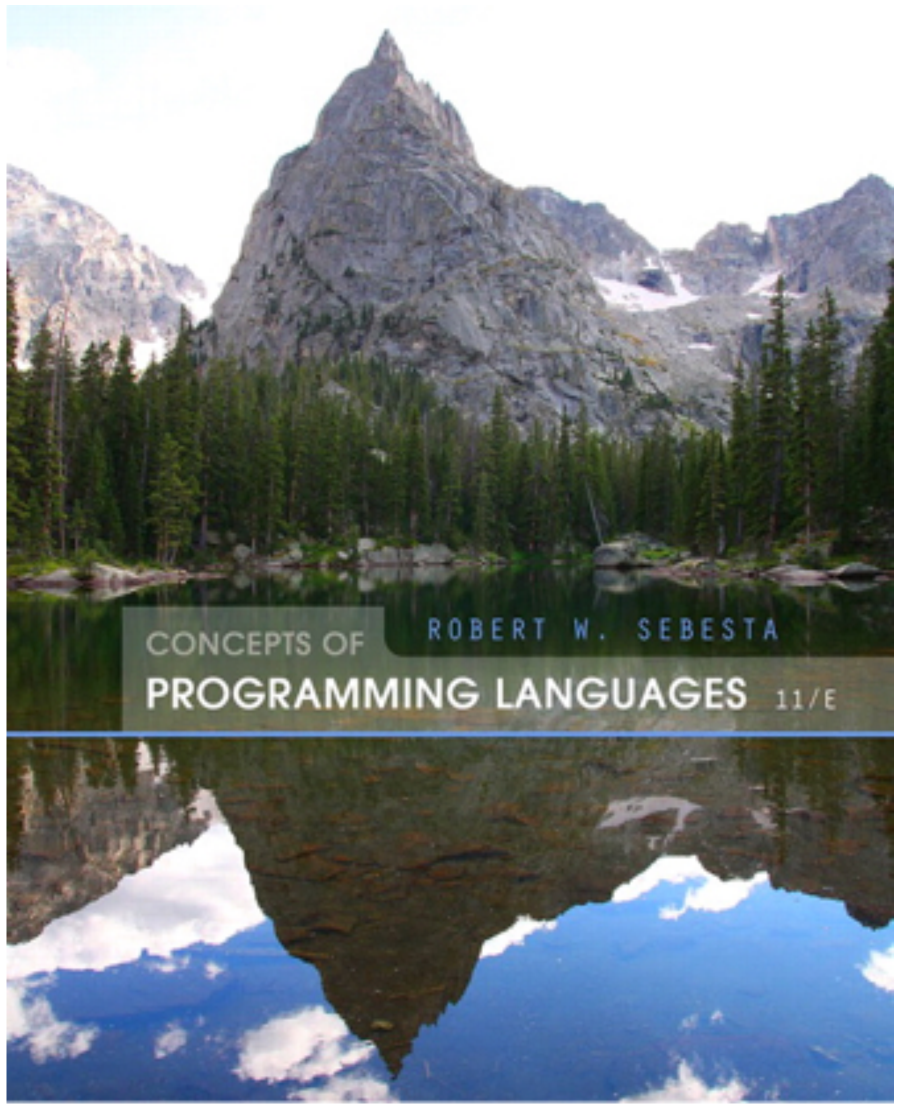
Programming Languages, Summary

CSC419; Odelia Schwartz



leftoversalad.com

Know your programming languages:

Python

R

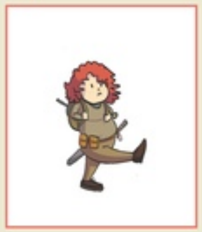
Java

JavaScript

PHP

Haskell

Perl



C

C++

C#

Ruby

Assembly

Erlang

Elixir

Go

Rust

MATLAB

Fortran

Lisp (dialects)

Brainfuck

Special thanks: @ScudoCorp, @wycats,G. Kiczales, S. Sever

Reasons for studying concepts of PL

- Increased ability to express ideas
- Improved background for choosing appropriate languages (when you open your startup… when solving particular problems)
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Better use of languages that are already known
- Overall advancement of computing

Programming Domains

- Scientific applications
  - Large numbers of floating point computations; use of arrays
  - Fortran (more recently though not stressed in book: Matlab, Python)
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated; use of linked lists
  - LISP

Language Categories

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Include languages that support object-oriented programming
  - Include scripting languages
  - Include the visual languages
  - Examples: Ada, C, Java, Perl, JavaScript, Ruby, Visual BASIC .NET, C++, Python, …
- Functional
  - Main means of making computations is by applying functions to given parameters
  - In pure languages, no side effects
  - Examples: LISP, Scheme, ML, Haskell

Language Categories (2)

- Logic
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog

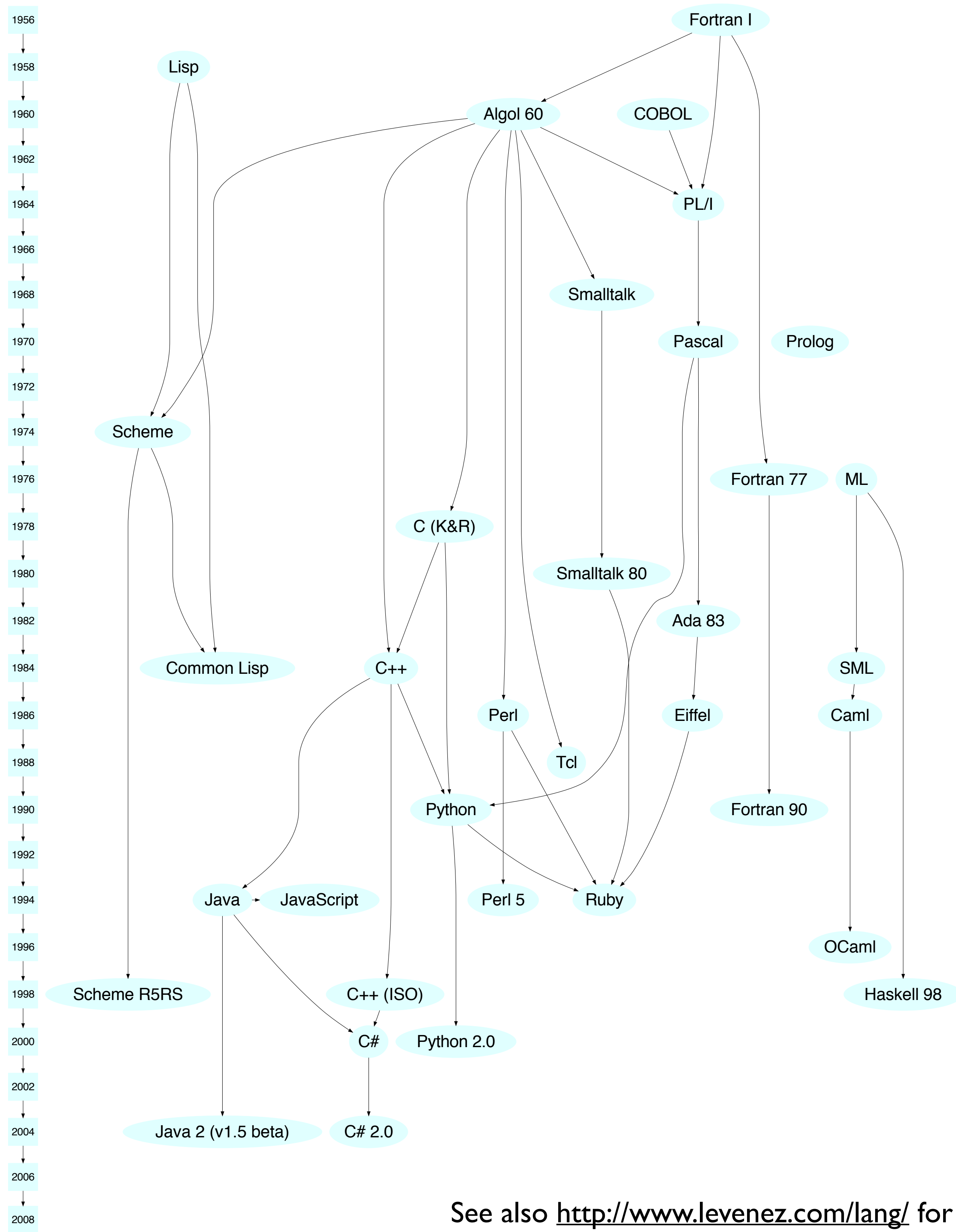
Language Evaluation Criteria

- Readability: the ease with which programs can be read and understood
- Writability: the ease with which a language can be used to create programs
- Reliability: conformance to specifications (i.e., performs to its specifications)
- Cost: the ultimate total cost

Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient
- We discussed Von Neumann bottleneck

Evolution of the Major Programming Languages
(light version)



See also http://www.levenez.com/lang/ for a complete list.

Describing Syntax and Semantics

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
- Three primary methods of semantics description
  - Operation, axiomatic, denotational

Example Grammar for small language

```
<program> → begin <stmt_list> end
 <stmt_list> → <stmt>
             | <stmt> ; <stmt_list>
 <stmt> → <var> = <expression>
 <var> → a | b | c
 <expression> → <var> + <var>
              | <var> - <var>
              | <var>
```

Example derivation

<program> => begin <stmt_list> end

- We'll derive A = B + C; B = C with this grammar
- A derivation is a repeated application of rules, starting with the start symbol (in this case program)
- => reads "derives"

Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end
=> begin A = <var> + <var> ; <stmt_list> end
=> begin A = B + <var> ; <stmt_list> end
=> begin A = B + C ; <stmt_list> end
=> begin A = B + C ; <stmt> end
=> begin A = B + C ; <var> = <expression> end
=> begin A = B + C ; B = <expression> end
=> begin A = B + C ; B = <var> end
=> begin A = B + C ; B = C end

Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end
=> begin A = <var> + <var> ; <stmt_list> end
=> begin A = B + <var> ; <stmt_list> end
=> begin A = B + C ; <stmt_list> end
=> begin A = B + C ; <stmt> end
=> begin A = B + C ; <var> = <expression> end
=> begin A = B + C ; B = <expression> end
=> begin A = B + C ; B = <var> end
=> begin A = B + C ; B = C end

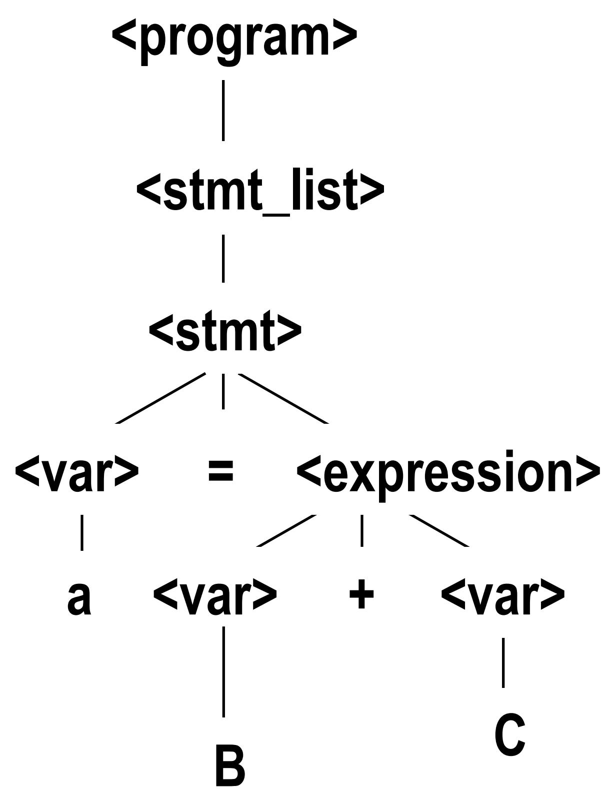
We derived leftmost; could have also done rightmost

Derivations

- Every string of symbols in a derivation is called a sentential form
- A sentence is a sentential form that has only terminal symbols
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded; similarly, rightmost derivation.
- A derivation may be neither leftmost nor rightmost

Parse Tree

- A hierarchical representation of a derivation



Ambiguity in Grammars

- A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees
- Problematic for compilers since parse tree, and therefore meaning of the structure, cannot be determined uniquely

Chapter 5: Names, Bindings, Type Checking, and Scopes

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors

Static and Dynamic Binding

- A binding is static if it first occurs before run time and remains unchanged throughout program execution.
- A binding is dynamic if it first occurs during execution or can change during execution of the program

Categories of Variables by Lifetimes

- Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ `static` variables

Categories of Variables by Lifetimes

- Stack-dynamic--Storage bindings are created for variables when their declaration statements are elaborated.

  (A declaration is elaborated when the executable code associated with it is executed)

  –local variables in C subprograms and Java methods

Categories of Variables by Lifetimes

- Explicit heap-dynamic -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
- Referenced only through pointers or references, e.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java

Categories of Variables by Lifetimes

- Implicit heap-dynamic--Allocation and deallocation caused by assignment statements
  - Ex: JavaScript and PHP; Python

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- Type checking is the activity of ensuring that the operands of an operator are of compatible types
- A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
    - This automatic conversion is called a coercion.
- A type error is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is strongly typed if type errors are always detected
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada
- Languages with a lot of coercion (C/C++), are less reliable than those with little coercion (Ada), and those with no coercion (ML).

Variable Attributes: Scope

- The scope of a variable is the range of statements over which it is visible
- The nonlocal variables of a program unit are those that are visible but not declared there
- The scope rules of a language determine how references to names are associated with variables
- Static and dynamic scope

Referencing Environments

- The referencing environment of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is active if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Union Types
- Pointer and Reference Types
- Note comparisons across languages

Chapter 7 Topics

- Introduction
- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Arithmetic Expressions: Design Issues

- Design issues for arithmetic expressions
  - Operator precedence rules?
  - Operator associativity rules?
  - Order of operand evaluation?
  - Operand evaluation side effects?
  - Operator overloading?
  - Type mixing in expressions?

Chapter 15
Functional Programming Languages

- Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution instead of imperative features such as variables and assignments
- Pure functional languages have no side effects
- LISP began as a purely functional language and later included imperative features
- Scheme is a relatively simple dialect of LISP that uses static scoping exclusively
- We also talked about increasing interest of functional in imperative, and some functional capabilities in imperative languages (such as in Python)

Functional Programming Languages

- ML is a static-scoped and strongly typed functional language which includes type inference, strong typing, and no type coercions; includes lists and list operations.
- Haskell is a lazy functional language supporting infinite lists; it is purely functional, unlike ML.
- Purely functional languages have advantages over imperative alternatives, but their lower efficiency on existing machine architectures has prevented them from enjoying widespread use
- We focused on some programming examples in Scheme (and also showed main concepts of ML/ Haskell)

Chapter 16: Logic Programming Languages

- Symbolic logic provides basis for logic programming
- Logic programs should be nonprocedural
- Prolog statements are facts, rules, or goals
- Resolution is the primary activity of a Prolog interpreter
- Although there are a number of drawbacks with the current state of logic programming it has been used in a number of areas
- We focused on some programming in Prolog

List processing capabilities

- We saw this in functional languages (e.g., Scheme, ML, Haskell)
- We also saw this in logical languages (Prolog)
- Comparison of similarities and differences in list processing capabilities