C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg.

- Bjarne Stroustrup

CSE341
Programming Languages

Lecture 1.2 – September 21, 2016

Programming Paradigms

Largely adapted from V. Shmatikov, J. Mitchell and R.W. Sebesta

Today

- Language Evaluation
- Programming Paradigms
- Compilation Process

Evaluating Programming Languages

Language Evaluation Criteria

- Readability: Most important
  - Overall simplicity
    - Too many features is bad
    - Multiplicity of features is bad
  - Orthogonality
    - Makes the language easy to learn and read
    - Meaning is context independent
  - Control statements
  - Data type and structures
  - Syntax considerations

Language Evaluation Criteria

- Writability:
  - Simplicity and orthogonality
  - Support for abstraction
  - Expressivity

Support for Abstraction

- Data
  - Programmer-defined types and classes
  - Class libraries
- Procedural
  - Programmer-defined functions
  - Standard function libraries

Reliability

- Program behavior is the same on different platforms
  - E.g., early versions of Fortran
- Type errors are detected
  - E.g., C vs. ML
- Semantic errors are properly trapped
  - E.g., C vs. C++
- Memory leaks are prevented
  - E.g., C vs. Java

What Does This Mean?

```
*p++ = *q++
```

Does this mean…	… or	… or
`*p = *q;`	`*p = *q;`	`tp = p;`
`++p;`	`++q;`	`++p;`
`++q;`	`++p;`	`tq = q;`
		`++q;`
		`*tp = *tq;`

Orthogonality

- A language is orthogonal if its features are built upon a small, mutually independent set of primitive operations.
- Fewer exceptional rules = conceptual simplicity
  - E.g., restricting types of arguments to a function
- Tradeoffs with efficiency

Efficient Implementation

- Embedded systems
  - Real-time responsiveness (e.g., navigation)
  - Failures of early Ada implementations
- Web applications
  - Responsiveness to users (e.g., Google search)
- Corporate database applications
  - Efficient search and updating
- AI applications
  - Modeling human behaviors

Language Evaluation Criteria

- Reliability:
  - Type checking
  - Exception handling
  - Aliasing
  - Readability and writability

Language Evaluation Criteria

- Cost:
  - Programmer training
  - Software creation
  - Compilation
  - Execution
  - Compiler cost
  - Poor reliability
  - Maintenance

Language Evaluation Criteria

- Others:
  - Portability
  - Generality
  - Well-definedness

Programming Paradigms

What Is a Programming Language?

- Formal notation for specifying computations, independent of a specific machine
  - Example: a factorial function takes a single non-negative integer argument and computes a positive integer result
  - Mathematically, written as factorial: Int → Int
- Set of imperative commands used to direct computer to do something useful
  - Print to an output device: printf("merhaba\n");
  - What mathematical function is "computed" by printf?

Partial and Total Functions

- Value of an expression may be undefined
  - Undefined operation, e.g., division by zero
    - 3/0 has no value
    - Implementation may halt with error condition
  - Nontermination
    - f(x) = if x=0 then 1 else f(x-2)
    - This is a partial function: not defined on all arguments
    - Cannot be detected by inspecting expression (why?)
- These two cases are “mathematically” equivalent, but operationally different (why?)

Partial and Total Functions

- Total function $f: A \to B$ is a subset $f \subseteq A \times B$ with
  - $\forall x \in A$, there is some $\mathrm{y} \in B$ with $\langle x, y \rangle \in f$ (total)
  - $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ then $\mathrm{y} = \mathrm{z}$ (single-valued)
- Partial function $f: A \to B$ is a subset $f \subseteq A \times B$ with
  - If $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ then $\mathrm{y} = \mathrm{z}$ (single-valued)
- Programs define partial functions for two reasons
  - What are these reasons?

Computability

- Function *f* is computable if some program P computes it
  - For any input *x*, the computation P(*x*) halts with output *f*(*x*)
  - Partial recursive functions: partial functions (int to int) that are computable

Halting Function

- Decide whether program halts on input
  - Given program P and input x to P,

    Halt(P,x) = yes, if P(x) halts

    no, otherwise

- Clarifications
  - Assume program P requires one string input x
  - Write P(x) for output of P when run in input x
  - Program P is string input to Halt

- Fact: There is no program for Halt

Unsolvability of the Halting Problem

- Suppose P solves variant of halting problem

  On input Q, assume P(Q) = yes, if Q(Q) halts
  no, otherwise

- Build program D

  D(Q) = run forever if Q(Q) halts
  halt if Q(Q) runs forever

- If D(D) halts, then D(D) runs forever
- If D(D) runs forever, then D(D) halts
- Contradiction! Thus P cannot exist.

Main Points About Computability

- Some functions are computable, some are not
  - Example: halting problem
- Programming language implementation
  - Can report error if program result is undefined due to an undefined basic operation (e.g., division by zero)
  - Cannot report error if program will not terminate

Computation Rules

- The factorial function type declaration does not convey how the computation is to proceed
- We also need a computation rule

  fact (0) = 1

  fact (n) = n * fact(n-1)
- This notation is more computationally oriented and can almost be executed by a machine

Factorial Functions

- C, C++, Java:
  - int fact (int n) { return (n == 0) ? 1 : n * fact (n-1); }
- Scheme:
  - (define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
- Haskell:
  - fact :: Integer->Integer
  - fact 0 = 1
  - fact n = n*fact(n-1)

Paradigms

Paradigms

- Imperative / Procedural
- Functional / Applicative
- Object-Oriented
- Concurrent
- Logic
- Scripting

- In reality, very few languages are “pure”
  – Most combine features of different paradigms

Why Paradigms?

- Paradigms emerge as the result of social processes in which people develop ideas and create principles and practices that embody those ideas
  - Thomas Kuhn. “The Structure of Scientific Revolutions.”
- Programming paradigms are the result of people’s ideas about how programs should be constructed
  - … and formal linguistic mechanisms for expressing them
  - … and software engineering principles and practices for using the resulting programming language to solve problems

Imperative

- Imperative (procedural) programs consists of actions to effect **state change**, principally through assignment operations or side effects
  - Fortran, Algol, Cobol, PL/I, Pascal, Modula-2, Ada, C
  - Why does imperative programming dominate in practice?
- OO programming is not always imperative, but most OO languages have been imperative
  - Simula, Smalltalk, C++, Modula-3, Java
  - Notable exception: CLOS (Common Lisp Object System)

Functional and Logic

- Focuses on function evaluation; avoids updates, assignment, mutable state, side effects
- Not all functional languages are "pure"
  - In practice, rely on non-pure functions for input/output and some permit assignment-like operators
  - E.g., (set! x 1) in Scheme
- Logic programming is based on predicate logic
  - Targeted at theorem-proving languages, automated reasoning, database applications
  - Declarative programming (express the logic of a computation without describing its control flow)

Concurrent and Scripting

- Concurrent programming cuts across imperative, object-oriented, and functional paradigms
- Scripting is a very “high” level of programming
  - Rapid development; glue together different programs
  - Often dynamically typed, with only int, float, string, and array as the data types; no user-defined types
  - Weakly typed: a variable ‘x’ can be assigned a value of any type at any time during execution
- Very popular in Web development
  - Especially scripting active Web pages

Unifying Concepts

- Unifying language concepts
  - Types (both built-in and user-defined)
    - Specify constraints on functions and data
    - Static vs. dynamic typing
  - Expressions (e.g., arithmetic, boolean, strings)
  - Functions/procedures
  - Commands
- We will study how these are defined syntactically, used semantically, and implemented pragmatically

Design Choices

- C: Efficient imperative programming with static types
- C++: Object-oriented programming with static types and ad hoc, subtype and parametric polymorphism
- Java: Imperative, object-oriented, and concurrent programming with static types and garbage collection
- Scheme: Lexically scoped, applicative-style recursive programming with dynamic types
- Standard ML: Practical functional programming with strict (eager) evaluation and polymorphic type inference
- Haskell: Pure functional programming with non-strict (lazy) evaluation.

Abstraction and Modularization

- Re-use, sharing, extension of code are critically important in software engineering
- Big idea: **detect errors at compile-time**, not when program is executed
- **Type** definitions and declarations
  - Define intent for both functions/procedures and data
- **Abstract data types** (ADT)
  - Access to local data only via a well-defined interface
- Lexical **scope**

Static vs. Dynamic Typing

- Static typing
  - Common in compiled languages, considered “safer”
  - Type of each variable determined at compile-time; constrains the set of values it can hold at run-time
- Dynamic typing
  - Common in interpreted languages
  - Types are associated with a variable at run-time; may change dynamically to conform to the type of the value currently referenced by the variable
  - Type errors not detected until a piece of code is executed

Why is this so Important?

- Failed launch of Ariane 5 rocket (1996)
  - \$500 million payload; \$7 billion spent on development
- Cause: software error in inertial reference system
  - Re-used Ariane 4 code, but flight path was different
  - 64-bit floating point number related to horizontal velocity converted to 16-bit signed integer; the number was larger than 32,767; inertial guidance crashed

Program Correctness

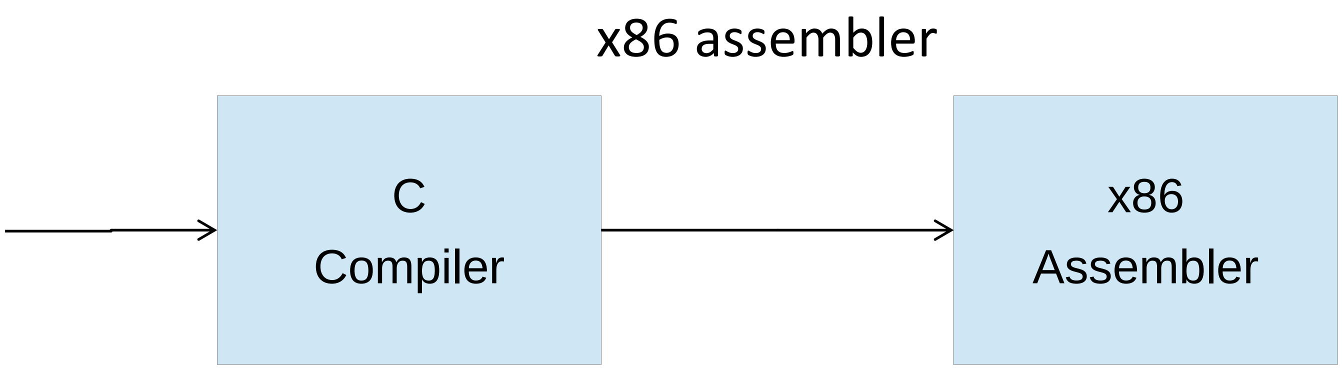
- Assert formal correctness statements about critical parts of a program and reason effectively
  - A program is intended to carry out a specific computation, but a programmer can fail to adequately address all data value ranges, input conditions, system resource constraints, memory limitations, etc.
- Language features and their interaction should be clearly specified and understandable
  - If you do not or can not clearly understand the semantics of the language, your ability to accurately predict the behavior of your program is limited

Language Translation

- **Native-code compiler**: produces machine code
  - Compiled languages: Fortran, C, C++, SML …
- **Interpreter**: translates into internal form and immediately executes (read-eval-print loop)
  - Interpreted languages: Scheme, Haskell, Python …
- **Byte-code compiler**: produces portable bytecode, which is executed on virtual machine (e.g., Java)
- Hybrid approaches
  - Source-to-source translation (early C++ to C to compile)
  - Just-in-time Java compilers convert bytecode into native machine code when first executed

Language Compilation

- **Compiler**: program that translates a source language into a target language
  - Target language is often, but not always, the assembly language for a particular machine



Checks During Compilation

- Syntactically invalid constructs
  - Invalid type conversions
  - A value is used in the “wrong” context, e.g., assigning a float to an int
- Static determination of type information is also used to generate more efficient code
  - Know what kind of values will be stored in a given memory region during program execution
- Some programmer logic errors
  - Can be subtle: if (a = b) … instead of if (a == b) …

Compilation Process

- Compilation: source code → relocatable object code (binaries)
- Linking: many relocatable binaries (modules plus libraries) → one relocatable binary (with all external references satisfied)
- Loading: relocatable → absolute binary (with all code and data references bound to the addresses occupied in memory)
- Execution: control is transferred to the first instruction of the program

Compilation Process

- At compile time, absolute addresses of variables and statement labels are not known
- In static languages (such as Fortran), absolute addresses are bound at load time
- In block-structured languages, bindings can change at run time

Phases of Compilation

- Preprocessing: conditional macro text substitution
- Lexical analysis: convert keywords, identifiers, constants into a sequence of tokens
- Syntactic analysis: check that token sequence is syntactically correct
- Generate abstract syntax trees (AST), check types
- Intermediate code generation: “walk” the ASTs and generate intermediate code
  - Apply optimizations to produce efficient code
- Final code generation: produce machine code

Language Interpretation

- Read-eval-print loop
  - Read in an expression, translate into internal form
  - Evaluate internal form
    - This requires an abstract machine and a “run-time” component (usually a compiled program that runs on the native machine)
  - Print the result of evaluation
  - Loop back to read the next expression

Bytecode Compilation

- Combine compilation with interpretation
  - Idea: remove inefficiencies of read-eval-print loop
- Bytecodes are conceptually similar to real machine opcodes, but they represent compiled instructions to a virtual machine instead of a real machine
  - Source code statically compiled into a set of bytecodes
  - Bytecode interpreter implements the virtual machine
  - In what way are bytecodes "better" then real opcodes?

Next Class

- Syntax and Semantics

- Reminders:
  - Homework 0 is assigned