CSE373: Data Structure & Algorithms
Lecture 23: Programming Languages

Aaron Bauer
Winter 2014
Choosing a Programming Language

• Most of the time you won’t have a choice about what programming language to use
  – Software is already written in a particular language
  – Platform requires a specific language (Objective-C for iOS)
  – Language required by computational tool (Mathematica, etc.)
• Still important to understand capabilities and limitations of language
• When you do get to choose, your choice can have tremendous impact
  – This is despite theoretical equivalence!
  – Turing Completeness
Turing Completeness

• A programming language is said to be **Turing complete** if it can compute every **computable function**
  – Recall the Halting Problem as a non-computable function
• In other words, every Turing complete language can approximately simulate every other Turing complete language
• Virtually every programming language you might encounter is Turing complete
  – Data or markup languages (e.g. JSON, XML, HTML) are an exception
• So a choice of language is about how computation is described, not about what it’s possible to compute
What we might want from a Language

• Readable (good syntax, intuitive semantics)
• High-level of abstraction (but still possible to access low level)
• Fast
• Good concurrency and parallelism
• Portable
• Manage side effects
• Expressive
• Make dumb things hard
• Secure
• Provably correct
• etc.
Type System

- Collection of rules to assign types to elements of the language
  - Values, variables, functions, etc.
- The goal is to reduce bugs
  - Logic errors, memory errors (maybe)
- Governed by type theory, an incredibly deep and complex topic

- The type safety of a language is the extent to which its type system prevents or discourages relevant type errors
  - Via type checking
- We’ll cover the following questions:
  - When does the type system check?
  - What does the type system check?
  - What do we have to tell the type system?
When Does It Check?

- Static type-checking (check at compile-time)
  - Based on source code (program text)
  - If program passes, it’s guaranteed to satisfy some type-safety properties on all possible inputs
  - Catches bugs early (program doesn’t have to be run)
  - Possibly better run-time performance
    - Less (or no) checking to do while program runs
    - Compiler can optimize based on type
  - Inherently conservative
    - *if* `<complex test> then <do something> else <type error>`
  - Not all useful features can be statically checked
    - Many languages use both static and dynamic checking
When Does it Check?

- Dynamic type-checking (check at run-time)
  - Performed as the program is executing
  - Often “tag” objects with their type information
  - Look up type information when performing operations
  - Possibly faster development time
    - edit-compile-test-debug cycle
  - Fewer guarantees about program correctness
What Does it Check?

- Nominal type system (name-based type system)
  - Equivalence of types based on declared type names
  - Objects are only subtypes if explicitly declared so
  - Can be statically or dynamically checked

- Structural type system (property-based type system)
  - Equivalence of types based on structure/definition
  - An element A is compatible with an element B if for each feature in B’s type, there’s an identical feature in A’s type
  - Not symmetric, subtyping handled similarly

- Duck typing
  - Type-checking only based on features actually used
  - Only generates run-time errors
How Much do we Have to Tell it?

• Type Inference
  – Automatically determining the type of an expression
  – Programmer can omit type annotations
    • Instead of (in C++)
      std::vector<int>::const_iterator itr = myvec.cbegin()
      use (in C++11)
      auto itr = myvec.cbegin()
  – Can make programming tasks easier
  – Only happens at compile-time
• Otherwise, types must be manifest (always written out)
How Flexible is it?

• Type conversion (typecasting)
  – Changing a value from one type to another, potentially changing the storage requirements
  – Reinterpreting the bit pattern of a value from one type to another
• Can happen explicitly or implicitly

```
double da = 3.3
double db = 3.3;
double dc = 3.4;
int result = (int)da + (int)db + (int)dc;
int result = da + db + dc;
```

• Can be done safely (checked) or unsafely (unchecked)
• Objects can be upcast (to supertype) or downcast (to subtype)
What Does it All Mean?

- Most of these distinctions are not mutually exclusive
  - Languages that do static type-checking often have to do some dynamic type-checking as well
  - Some languages use a combination of nominal and duck typing
- Terminology useful shorthand for describing language characteristics
- The terms “strong” or “weak” typing are often applied
  - These lack any formal definition
  - Use more precise, informative descriptors instead
- Languages aren’t necessarily limited to “official” tools
Memory Safety

- Memory errors
  - Buffer overflow
  - Dynamic
  - Uninitialized variables
  - Out of memory
- Often closely tied to type safety
- Can be checked at compile-time or run-time (or not at all)
- Memory can be managed manually or automatically
  - Garbage collection is a type of automatic management
  - Some languages make use of both
Programming Paradigms

• A programming paradigm describes some fundamental way of constructing and organizing computer programs
  – A programming language supports one or more paradigms
• Imperative
  – A program is a series of statements which explicitly change the program state.
• Declarative
  – A program describes what should happen without describing how it happens
• Functional (can be considered a type of declarative)
  – Computation done by evaluation of functions, avoiding state and mutable data
• Object-oriented (as opposed to procedural)
  – Computation done via objects (containing data and methods)
Language Development

- Many attempts to develop a “universal language”
  - have failed due to diverse needs
  - program size, programmer expertise, program requirements, program evolution, and personal taste
- Languages often change over time
  - Generics were added to Java 9 years after initial release
  - Take extreme care not to break existing code
- One “standard,” many implementations
  - Standard defines syntax and semantics
- Whether a language will become popular is unpredictable
  - Some research suggests things like library availability and social factors may be more important than language features
Java

• Age: 19 years
• Developer: Oracle Corporation
• Paradigms: imperative, object-oriented
• Type system: static, nominative, manifest
• One of the most popular languages in use today
  – Lots of great tools and other resources
• Write Once, Run Anywhere approach (via JVM)
  – Used to be considered slow, improved by JIT optimization
  – Other languages using JVM (Scala, Jython, Clojure, Groovy)
• Can be quite verbose, lacks a number of nice features
• Sees lots of use in large-scale enterprise software
• I would only choose to use Java if given no other options
C/C++

- Age: 42/31 years
- Developer: International Organization for Standardization
- Paradigms: imperative, procedural, object-oriented (C++ only)
- Type system: static, nominative, manifest (C++11 has inference)
- Two of the most popular languages in use today
- “Closer to the hardware” than Java
  - Used where predictable resource use is necessary
  - OS, graphics, games, compilers
- Manual memory management, less protection from memory errors, sometimes inscrutable compiler errors
  - Generally easier to “do dumb things”
- I’ve only used C/C++ when doing systems programming or when a library I needed was in C++
C#

• Age: 14 years
• Developer: Microsoft
• Paradigms: imperative, object-oriented, functional
• Type system: static, nominative, partially inferred
  – optionally dynamic
• Runs on the .NET Framework
  – Provides things like garbage collection (similar to the JVM)
• Allows access to system functions with `unsafe` keyword
• Less verbose than Java, safer than C++
• Primary use is writing Windows applications
• I have really enjoyed programming in C#, but Windows-only can be a big drawback
Haskell

• Age: 24 years
• Developer: many (research language)
• Paradigm: pure functional, lazy evaluation
• Type system: static, inferred
• Pure functional programming is a different way of thinking
  – maybe liberating, maybe frustrating
• Functional programming has seen only limited industrial use
• Safer and more transparent than an imperative language
  – Same function with same args always returns same value
  – Allows for compiler optimizations
• Performance suffers as hardware better suited to mutable data
• I think functional programming is fascinating, and enough languages include functional elements to make it worth learning
Haskell examples

factorial 0 = 1
factorial n | n > 0 = n * factorial (n - 1)

factorial n = product [1..n]

product xs = prod xs 1
    where
        prod [] a = a
        prod (x:xs) a = prod xs (a*x)

quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser = filter (< p) xs
        greater = filter (>= p) xs
SQL (Structured Query Language)

- Age: 40 years
- Developer: ISO
- Paradigms: declarative
- Type system: static
- Used as a database query language
  - Declarative paradigm perfect for this application

```
UPDATE country
SET population = population + 1
WHERE name = 'USA';
```

- Using SQL is both easy and very powerful
- If you have a lot of data, definitely consider using free database software like MySQL
Python

- Age: 23 years
- Developer: Python Software Foundation
- Paradigm: imperative, object-oriented, functional, procedural
- Type system: dynamic, duck
- Has a Read-Eval-Print-Loop (REPL)
  - Useful for experimenting or one-off tasks
- Scripting language
  - Supports “scripts,” small programs run without compilation
- Often used in web development or scientific/numeric computing
- Variables don’t have types, only values have types
- Whitespace has semantic meaning
- Lack of variable types and compile-time checks mean more may be required of documentation and testing
- Python is my language of choice for accomplishing small tasks
JavaScript

• Age: 19 years
• Developer: Mozilla Foundation
• Paradigm: imperative, object-oriented, functional, procedural
• Type system: dynamic, duck
• Also a scripting language (online/browser REPLs exist)
• Primary client-side language of the web
• Does inheritance through prototypes rather than classes
  – Objects inherit by cloning the behavior of existing objects
• Takes a continue at any cost approach
  – Shared by many web-focused languages (PHP, HTML)
  – Things that would be errors in other languages don’t stop execution, and are allowed to fail silently
• JavaScript is nice for simple things, immediately running on the web is great, but doing larger/more complex software is terrible
PHP

- Age: 19 years
- Developer: The PHP Group
- Paradigm: imperative, object-oriented, functional, procedural
- Type system: dynamic
- Works with Apache (>50% all websites), so very common server-side language
- Minimal type system, lots of strange behavior, just awful
  - If two strings are compared with ==, PHP will silently cast them to numbers (0e45h7 == 0w2318 evaluates to true)
- I’ve never used it and I never will (hopefully)
LOLCODE

- Age: 7 years
- An example of an esoteric programming language

HAI
CAN HAS STDIO?
PLZ OPEN FILE "LOLCATS.TXT"?
   AWSUM THX
      VISIBLE FILE
O NOES
   INVISIBLE "ERROR!"
KTHXBYE

HAI
CAN HAS STDIO?
IM IN YR LOOP UPPIN YR VAR TIL BOTH SAEM VAR AN 10
   VISIBLE SUM OF VAR AN 1
IM OUTTA YR LOOP
KTHXBYE