ECE326

PROGRAMMING LANGUAGES

Lecture 2 : Classification of Programming Languages

Kuei (Jack) Sun
ECE
University of Toronto

Fall 2019

Programming Languages

- Thousands of them
- Goal
  - How to describe a programming language in one line?
- Buzz words!
  - Intended audience (i.e. programmers)
  - Implementation
  - Level of abstraction
  - Dominant style, syntax, or feature
  - etc…

By Intended Users

- General-purpose language
  - Used by various application domains
  - E.g. C++, Python, Java, …etc
- Domain-specific language (DSL)
  - Specialized use
  - E.g. *Galaxy*
    - Designed for StarCraft 2’s map editor by Blizzard Entertainment
- No clear boundary
  - Perl and SQL used to be considered DSL

By Implementation

- Compiled
  - Source code compiled to machine code (executable)
  - Runs faster/more efficient
  - Requires recompilation for any source code change
    - Can impact productivity, especially for large software
  - Portability concerns
    - Compiler must support all target architectures
    - Bugs may be introduced when porting to another architecture
      - E.g. `long` in C is architecture dependent – can break developer assumption
      - `long long` – at least 64 bits
      - In couple of years: `long long long` for 128-bits?

By Implementation

- Interpreted
  - Usually associated with high-level languages
  - Interpreter directly executes instructions of a program
  - Usually done in one of the following way:
    1. Parse as you go
       - Syntax errors not caught until line is (about to be) executed
       - E.g. Bash script
    2. Translate to some intermediate representation first
       - E.g. Python
  - Limited form of optimization possible

By Implementation

- Mixture of both
  - Virtual machine
    - Source code compiled to bytecode
    - Run bytecode on virtual machine
    - Virtual machine can run on any hardware platform (portability)
    - E.g. Java Virtual Machine (JVM)
  - Just-in-time compilation
    - Compile while program is running!
    - Compile when “needed”
      - E.g. if compilation can result in speed up over running on the interpreter

Programming Idioms

- A language-specific convention of accomplishing a task
  - E.g. the “Pythonic” way
    - Create a string of numbers delimited by white space
    - nums is an array of numbers

```
def slow(nums):
    text = str(nums[0])
    for n in nums[1:]:
        text += " %d"%n
    return text
```

```
def fast(nums):
    return “ ".join(
        map(str, nums))
```

    - `fast` is about 50% faster than slow (on my laptop)

∴ Performance can still be good if you know the way

By Level of Abstraction

- more abstraction

- easier to write

- more direct hardware access

- better performance

High-Level Languages	Python, Ruby Java, Kotlin, Scala, Clojure Haskell, Racket Visual Basic, C#
Systems Languages	C/C++ Ada, D, Rust Swift (by Apple, for iOS apps)
Low-Level Languages	Assembly Languages Machine Languages

By Level of Abstraction

- Systems programming languages
  - Designed for performance
  - Allows some level of hardware awareness
    - Optimization hints (e.g. `restrict`, `volatile`, …etc)
    - Inline assembly
  - Still provides some high-level concepts
- High-level programming languages
  - Designed for convenience
  - Designed for expressiveness
    - Functional programming languages, E.g. Haskell

By Programming Style

- Imperative Programming
  - Writing commands and statements, changing program state
  - Concerns with *how* a program operates
  - E.g. procedural programming, object-oriented programming
- Declarative Programming
  - Writing expressions and desired result
  - Concerns with *what* a program should achieve
  - E.g. SQL queries, functional programming

```
SELECT firstName, LastName FROM Customers WHERE city=“Toronto”;
```

Turing Complete

- A programming language that can solve any computation problems (theoretically)
- Requirements
  1. Supports conditional branching
     - Allows for conditional (e.g. *if else*) and loops
  2. Can work with unlimited amount of memory
- Some languages are *not* Turing complete
  - E.g. regular languages, vanilla SQL, Datalog

By Type System

- Type system
  - the rules governing the use of types in a program, and how types affect the program semantics

```
unsigned sum_of_squares(unsigned a, unsigned b);
// which one of these is an error in C++11?
char res = area(3.3, -2);
```

- Statically Typed
  - Types of variables checked before runtime
- Dynamically Typed
  - Types are checked at runtime, on the fly

Implicit Type Conversion

```
#include <cstdio>
#include <cstdlib>

unsigned sum_of_squares(unsigned a, unsigned b) {
      return a*a + b*b;
}

int main(int argc, const char * argv[]) {
      int neg = atoi(argv[argc-1]);
      char res = sum_of_squares(3.3, neg);
      printf("sos = %d\n", res);
      return 0;
}
```

```
> ./err -2
sos = 13
> 😑
```

By Memory Safety

- Protection from invalid memory access
- Example
  - Buffer overflow
  - Use after free (dangling pointers)
  - Double free
- Memory *unsafe* languages
  - Languages that allows arbitrary pointer arithmetic (C/C++)
- Solution
  - runtime error detection (e.g. Java)
  - Static program analysis (e.g. Rust)

By Type Safety

- Protection from incorrect use of a variable
- Example
  - untagged union (C/C++)
  - Union
    - All member variables share *the same* memory location
    - Can lead to type-unsafe usage!
  - Solution: tagged union
    - Adds a tag field to indicate which member is in use

```
union Foo {
      int i;
      float f;
};

// in main()
Foo u;
u.i = atoi(argv[argc-1]);
printf("%f\n", u.f);

> ./union 1237864534
1640970.750000
```

By Features

- Generic Programming
  - Functions and classes defined in terms of *parameterized types*
  - Parameterized types
    - Instantiation of a generic type with actual type arguments
  - E.g. Java

```
public class Box<T> {
    private T content;
    public void set(T repl) { this.content = repl; }
    public T get() { return content; }
}
…
Box<Fruit> fruit_box = new Box<Fruit>();
```

By Simplicity

- E.g. Java vs. C++
- E.g. Visual Basic
- Simple is good
  - “Focus on debugging your application rather than debugging your programming language knowledge” – Zig developers
  - Design language for average programmers, not pros
    - Reduces chance of allowing for mistakes
    - Cheaper to hire 😒

By Syntax

- E.g. Off-side rule
  - Blocks in the language are expressed by indentation
  - Python:

```
def sum(n):
    if n == 0:
        return 0
    return 2*n + sum(n-1)
```

- Free-form languages
  - Whitespace characters serve only as delimiters
  - Scheme:

```
(define (sum n) (if (= n 0)
    0
    (+ (* n 2) (sum (- n 1)))
))
```

By Seriousness

- Esoteric programming languages
  - Programming as art, or a joke
  - Sometimes a subset of another language
  - E.g. JSF👀k (sanitized)
    - A subset of JavaScript
    - Uses only 6 characters: [ ] ( ) ! +
    - Prints “Hello world” in 26,924 characters

```
[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]
+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[
+!+[]]][([][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]
+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(
!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]
]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]…
```