Lecture Outline

- Introduction
- Art of programming language design
- Programming language spectrum
- Why study programming languages?
- Overview of compilation

Read: Scott Chapter 1
Introduction

- **Quizzes**
  - 10 in-class quizzes
  - Will cover material of previous week
  - Work in groups is encouraged
  - We will distribute quizzes at the beginning of lecture
  - Keep and submit quiz at the end of class

- **In-class exercises**
  - At the end of class, I may give a simple problem on the material we have just covered
  - Work in groups and ask questions
  - You can discuss or submit answers to receive feedback
  - Participation in these exercises carries towards 2% extra credit

- **Graduate students taking 6430 as part of the PhD qualifying exam**
  - To earn an A, you must earn 94% or more
  - Complete (satisfactorily!) a paper critique

- **Check syllabus**
  - [www.cs.rpi.edu/~milanova/csci4430/syllabus.htm](http://www.cs.rpi.edu/~milanova/csci4430/syllabus.htm)
  - for details

- **Academic Integrity**
  - All written homework assignments must be completed **individually**
  - Programming assignments, except for last one, must be completed **individually**
  - Excessive similarities in homework and exams! will be considered cheating
  - Incidents of cheating will be taken extremely seriously and will result in penalties

- **How to Study**
  - Read textbook chapter in advance of lecture
    - I will assign reading at the end of each class
  - Read lecture notes and textbook chapter (again) immediately after class
    - I will post lecture notes short after class
  - Solve exercises in lecture notes
  - Form study groups
  - **ASK QUESTIONS** – in class, after class, etc.
Course Topics
- Programming language syntax: Scanning and parsing
- Programming language semantics: Attribute grammars
- Naming, binding and scoping
- Data abstraction and types
- Control abstraction and parameter passing
- Concurrency
- A logic-oriented language: Prolog
- A functional language: Scheme
- Imperative languages
  - An object-oriented language: Java
  - A dynamic language: Python

Lecture Outline
- Introduction to the course
- The art of programming language design
- The programming language spectrum
- Why study programming languages
- Overview of compilation

The Art of Language Design
- Why are there so many languages?
  - Different programming domains
    - Business applications
    - Process data, reports, transactions, increasingly distributed
    - Manufacturing/control systems
    - Reliability and non-stop operation
    - Scientific applications
    - Computationally intensive
  - Personal preference…

The Programming Language Spectrum
- Imperative languages
  - Von Neumann languages: Fortran, C,…
  - Object-oriented languages: Java, C++, Smalltalk, …
  - Dynamic languages: Perl, Python, PHP,…
- Declarative languages
  - Functional languages: Scheme/Lisp, ML, Haskell
  - Logic languages: Prolog
  - There are other declarative languages: e.g., dataflow languages

Imperative languages
- Evolved from the von Neumann Architecture
- Variables
- Assignment Statements
Imperative languages

- Most widely popular programming style
  - FORTRAN, C, C++, C#, Java, Python, Visual BASIC, Perl, JavaScript, Ruby, etc.
- Variable and assignment statement are central concepts
- Program is a sequence of statements:
  - Execution is a sequence of transitions on memory state

Functional Programming

- Main alternative to imperative programming
  - Lisp/Scheme, ML, Haskell
- Program consists of function definitions + evaluation
  - \(\text{fun3} \circ (\text{fun2} \circ (\text{fun1 data})))\)
  - \(\text{data4}\)
- Execution is a sequence of function applications (i.e., reductions)

Logic Programming

- Perform queries against knowledge base
  - Prolog, Datalog, SQL

An Example: Inner Product

- Inner product in FORTRAN:
  1. \(C := 0;\)
  2. \(\text{for } I := 1 \text{ step } 1 \text{ until } N \text{ do}\)
  3. \(C := C + a[I]^\ast b[I];\)
- Illustrates state-transition semantics

- Inner product in FP:
  - \(\text{Def } \text{IP} = (\text{Insert } +) \circ (\text{ApplyToAll } \ast) \circ \text{Transpose}\)
  - \(\text{IP }\langle 1,2,3\rangle,\langle 6,5,4\rangle\) is
  - \((\text{Insert } +) \circ (\text{ApplyToAll } \ast) \circ \text{Transpose} \langle 1,2,3\rangle,\langle 6,5,4\rangle\)
  - \(\langle 1,6,10,12\rangle\)
- Illustrates reduction (applicative) semantics
Why Study Programming Languages

Goal of the course: learn to analyze programming languages
- What are the questions we ask when facing a new programming language
- Helps learn new languages, choose the right language for a problem, understand language features, design better languages!

Lecture Outline

- Introduction to the course
- The art of language design
- The programming language spectrum
- Why study programming languages
- Overview of compilation

Compilation and Interpretation

- Compilation
  - A "high-level" program is translated into executable machine code
  - Compiler
- Pure interpretation
  - A program is translated and executed one statement at a time
  - Interpreter
- Hybrid interpretation
  - A program is "compiled" into intermediate code; intermediate code is "interpreted"
  - Both a compiler and an interpreter

Compilation

```
position = initial + rate * 60;
```

Scanner
```
<id,1> <=> <id,2> <+> <id,3> <*> <60>
```

Parser
```
1. position, ...
2. initial, ...
3. rate, ...
```

Symbol table

Semantic analysis and intermediate code generation

Compilation

```
tmp1 = inttoreal (60)
tmp2 = id3 * tmp1
tmp3 = id2 + tmp2
id1 = tmp3
```

Abstract Syntax Tree

Intermediate form
(three-address code)
Compilation

Machine-independent code improvement

\[
\text{tmp1} = \text{id3} \times 60.0 \\
\text{id1} = \text{id2} + \text{tmp1}
\]

Code generation

\[
\text{movf id3, R2} \\
\text{mulf #60.0, R2} \\
\text{movf id2, R1} \\
\text{addf R2, R1} \\
\text{movf R1, id1}
\]

Improved intermediate form

Target language

Pure Interpretation

Source program

\[
\text{REM} \\
\text{LET X = 5} \\
\text{LET Y = 8} \\
\text{PRINT X} \\
\text{PRINT Y} \\
\text{LET Z = X} \\
\text{PRINT Z}
\]

Also JavaScript...

Code generation

Target language

Hybrid Interpretation

Hybrid code generation

e.g. Java byte code

e.g. Java Virtual Machine (JVM)

Compilation vs. Interpretation

Advantages of compilation?
- Faster execution

Advantages of interpretation?
- Greater flexibility

Advantages of compilation?
- Faster execution

Advantages of interpretation?
- Greater flexibility

What are the features that impact “compilability”?
- “Static” languages, i.e., languages that perform most semantic checks before program execution, are suitable for compilation
- “Dynamic” languages, i.e., languages that perform most semantic checks during execution, are suitable for interpretation

Compilation vs. Interpretation

A language can be implemented using a compiler or using an interpreter
- One can build a compiler for Lisp and one can easily build an interpreter for C or Fortran
- However, language features (determined during language design) have tremendous impact on “compilability” and the decision “compiler vs. interpreter”
Compilation vs. Interpretation

- New languages are increasingly dynamic
  - Hybrid implementations are common: compiler from source to intermediate form, then a smart interpreter (Virtual Machine) which does interpretation and JIT (Just-in-Time) compilation
- Tons of code written in static languages (C, C++, Fortran)
- Dynamic instrumentation tools (Valgrind, Pin, DynamoRio) serve as "interpreters" for compiled binaries

Next Class

- We will review regular expressions and context free grammars
- Read Chapter 2.1 and 2.2 from Scott's book