Introduction and overview

Principles of Programming Languages

Mark Armstrong

Fall 2020

1 Preamble

The preamble section of each notes will include

- notable references,
  - i.e., specific chapters of our recommended/additional texts from which the notes are derived, or which expand on the notes,
- a table of contents, and
- an update history, chronicling any major changes.
  - Note the git commit history will provide a more fine-grained record of upates.

1.1 TODO Notable references

:TODO:

1.2 TODO Table of contents

2 Introduction

This section of notes introduces the course and the staff, and lays out a few central concepts.

3 Welcome

Welcome to the course!

3.1 Instructor: Mark Armstrong

- Email: mailto:armstmp@mcmaster.ca
- Website: https://armkeh.github.io

3.2 Teaching assistants

:TODO:

4 Purpose and goals of this course

4.1 Calendar description

Design space of programming languages; abstraction and modularization concepts and mechanisms; programming in non-procedural (functional and logic) paradigms; introduction to programming language semantics.

4.2 Informal objectives

- Investigate a number of programming languages which exemplify different paradigms.
  - A relatively shallow but comprehensive survey.
  - Focusing on general-purpose languages.
- *Formally* describe programming language syntax and semantics.
  - An application of theory learned previously.
- Apply various abstraction and modularisation techniques,
  - Learning how to apply them and to which situations they are best applied.

4.3 Course preconditions

Before beginning this course:

1. Students should know and understand: a. Basic concepts about integers, sets, functions, & relations. b. Induction and recursion. c. First order logic, axiomatic theories & simple proof techniques. d. Regular expressions & context-free grammars. e. Programming in imperative languages. f. Basic concepts of functional programming languages.

2. Students should be able to: a. Produce proofs involving quantifiers and/or induction. b. Understand the meaning of a given axiomatic theory. c. Construct regular sets & context-free languages. d. Produce small to medium scale programs in imperative languages. e. Produce small scale programs in functional languages.

4.4 Course postconditions

After completion of this course:

1. Students should know and understand: a. Programming in functional languages. b. Programming in logical languages. c. Formal definitions of syntax & semantics for various simple programming languages. d. Various abstraction & modularisation techniques employed in programming languages.

2. Students should be able to: a. Reason about the design space of programming languages, in particular tradeoffs & design issues. b. Produce formal descriptions of syntax & semantics from informal descriptions, identifying ambiguities. c. Select appropriate abstraction & modularisation techniques for a given problem. d. Produce tools for domain-specific languages in imperative, functional and logical languages.

4.5 Formal rubric for the course

Topic	Below	Marginal	Meets	Exceeds
Familiarity with various programming languages	Shows some competence in procedural languages, but not languages from other paradigms	Shows competence in procedural languages and limited competence in languages from other paradigms	Achieves competence with the basic usage of various languages	Achieves competence with intermediate usage of various languages
Ability to identify and make use of abstraction, modularisation constructs	Cannot consistently identify such constructs	Identifies such constructs, but does not consistently make use of them when programming	Identifies such constructs and shows some ability to make use of them when programming	Identifies sucj constructs and shows mastery of them when programming
Ability to comprehend and produce formal descriptions of PL syntax	Unable or rarely able to comprehend given grammars; does not identify ambiguity or precedence rules	Comprehends given grammars, but produces grammars which are ambiguous or which do not correctly specify precedence	Makes only minor errors regarding precedence or ambiguity when reading or producing grammars	Consistently fully understands given grammars and produces correct grammars.
Ability to comprehend and produce operational semantics for simple PLs	Rarely or never comprehends such semantic descriptions	Usually comprehends such semantic descriptions, but cannot consistently produce them	Comprehends such semantic descriptions and produces them with only minor errors	Comprehends such semantic descriptions and produces them without errors

5 "Principles of programming languages"

We begin the course with these fundamental questions.

- What is a *programming language*?
- What are the *components* of a programming language?
- How do we *classify* a programming language?

5.1 What is a programming language?

- A *formal*, *finitely described* language used for describing (in most cases, potentially infinite) *processes*.
  - *Formal* meaning described by a mathematical tool.
    - Formality is necessary for a machine to understand the language.
    - Natural (human-spoken) languages are not formal.
  - A *process* being some sequence of actions or steps.

5.1.1 Example of a process

Consider the mathematical function $f(x) = x + 10$.

- On its own, this function is not a process;
  - it is only a *rule* that $f(x)$ is related to $x + 10$.

However, you likely learned as a child a "program" describing the process for calculating $f(x)$.

```
start with all your fingers down
say "x"
repeat until you run out of fingers:
  say the result of adding one to the number you just said
  put up one finger
the answer is the last number you said
```

In computing, we sometimes conflate programs and (mathematical) functions.

- Sometimes, we must remember they are not the same.

- Mathematical functions are rules. They do no computing.
- Programs describe a sequences of steps. They may tell us how to compute the results of mathematical functions.

5.2 What are the components of a programming language?

Just like a natural language, a programming language consists of

- *syntactic* rules
  - which describe the legal forms of programs, and
- *semantics* rules
  - which describe the meaning of legal programs,
    - if they in fact have a meaning!

5.2.1 Syntax and semantics example

For example, English syntax tells us a sentence structured

```
adjective adjective (plural noun) (plural verb) adverb
```

is grammatically correct.

In the same way, a Python compiler tells us a program of the form

```
expression + expression
```

is syntactically correct.

Note that in both cases, though, such sentences/programs may be meaningless! Noam Chomsky gave the example

> Colourless green ideas sleep furiously.

And we could construct the Python program

```python
1 + "hello"
```

which crashes when run.

5.2.2 Exercise: a meaningless C or Java program

Our example Python program above

```
1 + "hello"
```

is syntactically correct because Python is *dynamically typed*, meaning that type errors such as this are not caught until runtime.

As an exercise, can you construct a similar example of a program which is syntactically correct but semantically meaningless in the *statically typed* languages C and Java?

Hint: consider using a value which does not have just one type.

5.3 How do we classify a programming language?

First and foremost, we classify languages into *paradigms*,

- characterised by the set of *abstractions* the language makes available.

But also in many other ways, such as:

- Typing properties, including
  - static or dynamic (runtime) typechecking,
  - "weak" or "strong" typing discipline,
  - polymorphism support, builtin types, methods of defining new types, etc.
- (Primary) implementation strategy: compiled or interpreted?
- Ancestery or culture.
  - "Scripting languages"
  - "JVM languages"
  - "The C-family"
    - https://en.wikipedia.org/wiki/List_of_C-family_programming_languages

6 Abstraction

:TODO:

7 TODO Exercises