

Elements of Programming Languages

Lecture 0: Introduction and Course Outline

James Cheney

University of Edinburgh

September 22, 2026

What is programming?

- Computers are deterministic machines, controlled by low-level (usually binary) machine code instructions.
- A computer **can [only] do whatever we know how to order it to perform** (Ada Lovelace, 1842)
- Programming is **communication**:
 - between a person and a machine, to tell the machine what to do
 - between people, to communicate ideas about algorithms and computation
 - (between machines and humans, to tell the human what the coding agent has done)

From machine code to programming languages

- The first programmers wrote all of their code directly in machine instructions
 - ultimately, these are just raw sequences of bits.
- Such programs are extremely difficult to write, debug or understand.
- Simple “assembly languages” were introduced very early (1950’s) as a human-readable notation for machine code
- FORTRAN (1957) — one of the first “high-level” languages (procedures, loops, etc.)

What is a programming language?

- For the purpose of this course, a programming language is a *formal, executable* language for *defining computations*
- Non-examples:

What is a programming language?

- For the purpose of this course, a programming language is a *formal, executable* language for *defining computations*
- Non-examples:
 - English (not formal/executable)

What is a programming language?

- For the purpose of this course, a programming language is a *formal, executable* language for *defining computations*
- Non-examples:
 - English (not formal/executable)
 - First-order Logic (formal, but not executable in general)

What is a programming language?

- For the purpose of this course, a programming language is a *formal, executable* language for *defining computations*
- Non-examples:
 - English (not formal/executable)
 - First-order Logic (formal, but not executable in general)
 - HTML4 (formal, executable but not computational)

What is a programming language?

- For the purpose of this course, a programming language is a *formal, executable* language for *defining computations*
- Non-examples:
 - English (not formal/executable)
 - First-order Logic (formal, but not executable in general)
 - HTML4 (formal, executable but not computational)
 - Prompts for ChatGPT, Copilot etc. (executable, computational, not formal, but can be asked to write programs based on natural language specs!)

What is a programming language?

- For the purpose of this course, a programming language is a *formal, executable* language for *defining computations*
- Non-examples:
 - English (not formal/executable)
 - First-order Logic (formal, but not executable in general)
 - HTML4 (formal, executable but not computational)
 - Prompts for ChatGPT, Copilot etc. (executable, computational, not formal, but can be asked to write programs based on natural language specs!)
- (HTML is in a gray area — with JavaScript or HTML5 extensions it is a lot more “computational”)

Some different languages

```
# Python, Java, SQL  
print("Hello, world!")
```

```
// Java  
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World");  
    }  
}
```

```
-- SQL  
SELECT DISTINCT 'Hello world!' AS new_value  
FROM AnyTableWithOneOrMoreRows  
WHERE 1 = 1;
```

Why are there so many?

Many different goals/motivations

- Scientific computation: FORTRAN, R, Julia
- Commercial needs/industry backing: COBOL, C, C++, Java, C#, F#, Ruby, JavaScript, Rust, SQL, WebAssembly
- Scripting: Perl, Python, Ruby
- Explore research ideas: LISP, Simula, Smalltalk, Algol, Pascal, Scheme, Racket, ML, OCaml, Haskell, Prolog, Curry

These migrate over time, for example Python now widely used for scientific computation

What do they have in common?

- All (formal) languages have a written form: we call this (concrete) *syntax*
- All (executable) languages can be implemented on computers: e.g. by a *compiler* or *interpreter*
- All programming languages define computations: they have some *computational meaning*, or *semantics*
- In addition, most languages provide *abstractions* for organizing, decomposing and combining parts of programs to solve larger problems.

What are the differences?

There are many so-called “programming language paradigms”:

- imperative (variables, assignment, if/while/for, procedures)
- object-oriented (classes, inheritance, interfaces, subtyping)
- typed (statically, dynamically, strongly, un/uni-typed)
- functional (λ -calculus, pure, lazy)
- logic/declarative (computation as deduction, query languages)

each representing a (more or less coherent) philosophy of what computation is

Languages, paradigms and elements

- A great deal of effort has been expended trying to find the “best” paradigm, with no winner declared so far.
- In reality, they all have strengths and weaknesses, and almost all languages make compromises or synthesize ideas from several “paradigms”.
- This course emphasizes different programming language **features**, or **elements**
 - Analogy: periodic table of the elements in chemistry
- Goal: understand the basic components that appear in a variety of languages, and how they “combine” or “react” with one another.

Applicability

- Major new general-purpose languages come along every decade or so. (C/C++, Java, Python?, Rust?)
 - Hence, few programmers or computer scientists will design a new, widely-used general purpose language, or write a compiler
 - However, domain-specific languages are increasingly used, and the same principles of design apply to them
- Moreover, understanding the principles of language design can help you become a better programmer
 - Learn new languages / recognize new features faster
 - Understand when and when *not* to use a given feature
- Assignments will cover practical aspects of programming languages: *interpreters* and *DSLs/translators*

Course Administration

Staff

- Lecturer: James Cheney <jcheney@inf.ed.ac.uk>, IF 5.29
 - Office hours: by appointment
- TA: Toby Ueno

Format

- 20 **lectures** (Tu 11:10-12:00, Th 1410–1500)
 - 2 intro/review [non-examinable]
 - 2 guest lectures [non-examinable]
 - 16 core material [examinable]
- 1 two-hour **lab session** (September 30, 1210–1400)
- 8 one-hour **tutorial sessions**, starting in week 3 (times and groups TBA)
- 3 **assignments**, first two feedback only

All of these activities are **part of the course** and may cover examinable material, unless explicitly indicated.

Feedback and Assessment

- Coursework:
 - Assignment 1: **Lab exercise sheet**, available during week 2, due during week 3, worth 0% of final grade
 - Assignment 2: available during week 3, due week 6, worth 0% of final grade.
 - Assignment 3: available during week 6, due week 10, worth 20% of final grade.
 - The first two assignments are marked for formative feedback only, but the third **builds on the first two**.
- One (written) exam: worth 80% of final grade.

Scala

- The main language for this course will be *Scala*
 - Scala offers an interesting combination of ideas from functional and object-oriented programming styles
 - We will use Scala (and other languages) to illustrate key ideas
 - We will also use Scala for the assignments
- However, this is not a “course on Scala”
 - You will be expected to figure out certain things for yourselves (or ask for help)
 - We will not teach every feature of Scala, nor are you expected to learn every dark corner
 - In fact, part of the purpose of the course is to help you recognize such dark corners and avoid them unless you have a good reason...

Recommended reading

- There is no official textbook for the course that we will follow exactly
- However, the following are recommended readings to complement the course material:
 - Practical Foundations for Programming Languages, second edition, (PFPL2), by Robert Harper. Available online from the author's webpage and through the University Library's ebook access.
 - Concepts in Programming Languages (CPL), by John Mitchell. Available through the University Library's ebook access.
- **Slides** available on web page

Course Outline

Wadler's Law

In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.

0. Semantics
1. Syntax
2. Lexical syntax
3. Lexical syntax of comments

See also: *bikeshedding* (n). Technical disputes over minor, marginal issues conducted while more serious ones are being overlooked.

Few languages are well-designed because few people know what (good or bad) language design is. Let's change that.

Syntax

- This course is primarily about language design and semantics.
- As a foundation for this, we will necessarily spend some time on abstract syntax trees (and programming with them in Scala)
- We will cover: Name-binding, substitution, static vs. dynamic scope
- We will not cover: Concrete syntax, lexing, parsing, precedence (Compiling Techniques does some of this)

Interpreters, Compilers and Virtual Machines

- Suppose we have a *source* programming language L_S , a *target* language L_T , and an *implementation* language L_I
 - An *interpreter* for L_S is an L_I program that executes L_S programs.
 - When both L_S and L_I are low-level (e.g. $L_S = \text{JVM}$, $L_I = \text{x86}$), an interpreter for L is called a *virtual machine*.
 - A *translator* from L_S to L_T is an L_I program that translates programs in L_S to “equivalent” programs in L_T .
 - When L_T is low-level, a translator to L_T is usually called a *compiler*.
- In this course, we will use interpreters to explore different language features.

Semantics

- How can we understand the meaning of a language/feature, or compare different languages/features?
- Three basic approaches:
 - *Operational semantics* defines the meaning of a program in terms of “rules” that explain the step-by-step execution of the program
 - *Denotational semantics* defines the meaning of a program by interpreting it in a mathematical structure
 - *Axiomatic semantics* defines the meaning of a program via logical specifications and laws
- All three have strengths and weaknesses
- We will focus on operational semantics in this course: it is the most accessible and flexible approach.

Abstraction, abstraction, abstraction

- The three most important considerations for programming language design are:
 - (Data) Abstraction
 - (Control) Abstraction
 - (Modular) Abstraction
- We will investigate different language elements that address the need for these abstractions, and how different design choices interact.
- In particular, we will see how **types** offer a fundamental organizing principle for programming language features.

Data Structures and Abstractions

- **Data structures** provide ways of organizing data:
 - option types vs. null values
 - pairs/record types;
 - variant/union types;
 - lists/recursive types;
 - pointers/references
- **Data abstractions** make it possible to hide data structure choices:
 - overloading (ad hoc polymorphism)
 - generics (parametric polymorphism)
 - subtyping
 - abstract data types

Control Structures and Abstractions

- **Control structures** allow us to express flow of control:
 - goto
 - for/while loops
 - case/switch
 - exceptions
- **Control abstractions** make it possible to hide implementation details:
 - procedure call/return
 - function types/higher-order functions
 - continuations

Design dimensions and modularity

- Programming “in the large” requires considering several cross-cutting **design dimensions**:
 - eager vs. lazy evaluation
 - purity vs. side-effects
 - static vs. dynamic typing
- and **modularity** features
 - modules, namespaces
 - objects, classes, inheritance
 - interfaces, information hiding

The art and science of language design

- Language design is both an art and a science
- The most popular languages are often not the ones with the cleanest foundations (and vice versa)
- This course teaches the science: formalisms and semantics
- Aesthetics and “good design” are hard to teach (and hard to assess), but one of the assignments will give you an opportunity to experiment with domain-specific language design

Course goals

By the end of this course, you should be able to:

- 1 Investigate the design and behaviour of programming languages by studying implementations in an interpreter
- 2 Employ abstract syntax and inference rules to understand and compare programming language features
- 3 Design and implement a domain-specific language capturing a problem domain
- 4 Understand the design space of programming languages, including common elements of current languages and how they are combined to construct language designs
- 5 Critically evaluate the programming languages in current use, acquire and use language features quickly, recognise problematic programming language features, and avoid their (mis)use.

Relationship to other UG3 courses

- **Compiling Techniques (S2)**
 - covers complementary aspects of PL implementation, such as lexical analysis and parsing, compilation of imperative programs to machine code
- **Introduction to Theoretical Computer Science (S1)**
 - covers formal models of computation (Turing machines, etc.) as well as some λ -calculus and type theory
- In this course, we focus on *interpreters*, *operational semantics*, and *types* to understand programming language features.
- There is relatively little overlap with CT or ITCS.

Summary

- Today we covered:
 - Background and motivation for the course
 - Course administration
 - Outline of course topics
- Next time:
 - Concrete and abstract syntax
 - Programming with abstract syntax trees (ASTs)