1 Introduction

Welcome to CS442! This course is titled “Principles of Programming Languages”, and rightly so. This course will discuss the principles underlying the design and implementation of programming languages.

First, let’s try to eliminate some misunderstandings by explaining what this course is not:

• This course is not “a programming language every week”. Although we will be looking at several programming languages you’re probably not familiar with, our focus is on depth. We will examine programming languages as artifacts themselves.

• This course is not a compilers course. That’s CS444. Although you will be implementing languages, the implementations won’t be good; the goal is understanding, not efficiency.

• This course is not a history course, although we will look a bit at the history of programming languages for context. We aim to examine timeless concepts.

So, if this course isn’t any of those things, what is it? We’ll be looking at two major aspects of programming languages: How they are formally defined, in a framework allowing for mathematical rigor and proofs, and the scope of language paradigms that exist.

Formally, a programming language is modeled as a calculus. If your first language is English, you’ve probably never encountered the concept of a calculus, but in fact, the system we describe just as “calculus” is “the calculus of differentials and integrals”, or “infinitesimal calculus”. It came to be known simply as “calculus” because the Fundamental Theory of Calculus unified two calculi which were previously independent. A calculus is simply a mathematical language; a language for describing mathematical ideas. Set theory has a calculus, the calculus of sets; linear algebra has a calculus, the calculus of matrices. Indeed, even arithmetic is a calculus: it could be described simply as the calculus of numbers, but is more properly called, well, arithmetic. We never typically describe these things as calculi, because we don’t usually think about the calculi themselves. They’re used as tools.

We will bridge the gap between these mathematical calculi and programming languages by building a particular, simple programming language, called the $\lambda$-calculus (Lambda calculus), both as a mathematical calculus and as a (rather impractical) programming language. As a calculus, we will describe it with mathematical rigor, allowing us to prove some properties of $\lambda$-calculus expression. As a language, we will implement it in software, such that $\lambda$-calculus expressions are also $\lambda$ programs. In both, we will extend it to understand how language concepts affect language behavior both formally and practically.

As well as $\lambda$-calculus and its derivatives, we will be looking at real programming paradigms. A programming paradigm is simply a way of thinking about software, usually exemplified by a fundamental way that data and code are stored and interacted with. You’re probably familiar with functional programming languages, such as Racket, and object-oriented languages, such as Java and (arguably) C++. In this course, we will be looking at a few more programming paradigms. For each paradigm, we will be looking at an exemplar programming language—i.e., a programming language which exemplifies the paradigm without trying to be “multi-paradigm” and thus muddying
the water—and we will look at formal calculi which model the behavior of such languages. The paradigms and exemplars we will be examining are:

- Functional and Haskell
- Logic and Prolog
- Imperative and Pascal
- Object oriented and Smalltalk
- Concurrent and Erlang
- Systems and C

2 Administrata

This course is being taught online. Presumably, you already know that. Winter 2021 is the first term in which this course is being taught online\(^1\), so expect things to be a bit bumpy.

The course web site is https://student.cs.uwaterloo.ca/~cs442. There you can find the course outline, links to these course notes, assignments, information on office hours, and, importantly, the course schedule. As the course schedule is on the web site, it will not be repeated here. Please familiarize yourself with it there. Announcements will be on Piazza, linked from the course web site.

The primary learning material in this course are text-based notes such as this document. These text-based notes will sometimes have supplementary videos, referenced like so:

**Video 1.1** (https://student.cs.uwaterloo.ca/~cs442/W21/videos/1.1/): Course Introduction

Note that that isn’t just a demonstration of how we’ll reference videos, it’s also a real video which you should watch!

You are highly recommended to watch these supplementary videos, as they should help to clarify the text. As this course is being developed this term, videos may be added after the notes have been initially released. If this happens, it will be announced in Piazza.

In addition, update videos will be posted every Monday on Piazza. These do not contain learning material, only scheduling information, reminders, and hints.

Some additional content which is not part of the required course content, but might help to contextualize the course content, will be shown in separate “aside” segments, like so:

**Aside:** Gregor Richards, who wrote this module, is a total programming languages dweeb, and may sometimes find it difficult not to put in asides that he finds interesting, but are actually far removed from the important material!

The basis for grading in this course is assignments and exams. The assignments are programming assignments, in which you will mostly be implementing programming language features. There are six assignments, which will be posted on the course web page when they are available. The exams test all course material. There are five exams, coinciding with assignments one through five, which will become available at the due time of those assignments. All assignment and exam scheduling information is available on the course web site.

You will be writing assignments in two programming languages: OCaml and Smalltalk. You are not expected to already know either of these languages, though of course it’s good if you do. They will be introduced later in this module.

\(^1\)If you’re reading this in a term later than Winter 2021, hello! Tell your instructor that this message is no longer relevant!
3 History of Programming Languages

The history of programming languages predates the history of computers, but has followed closely with the power and capability of computers since their inception. The earliest experience of real programming was manually entering CPU-specific code (machine code) in binary. Ultimately, this happens even to this day: all CPU’s run their own machine code. Advancements in programming languages are possible because programming allows abstraction: writing a program in machine code is excessively annoying, but once the first assembler is written in machine code, the programmer is free to program in assembly instead of raw machine code. Once the first language compiler targeting assembly is written in assembly, the programmer is free to program in its language. Each of these steps allows a higher-level language, albeit not without consequences in terms of raw performance and predictability.

One of the first widely-used “high-level” languages—here, “high-level” really just means higher-level than assembly code—was Fortran, originally developed in the 1950s by IBM. The original version of Fortran predated a feature we consider so fundamental to programming languages since that we rarely even feel the need to name it: structured programming.

The thing that is “structured” about structured programming is control flow. Now-familiar concepts such as if blocks and procedures were yet to be invented. Before structured programming, the program was one giant list of numbered instructions, and the programmer could choose to conditionally jump to a different instruction. If they wanted to jump back again (and thus form a conditional block, like an if statement), they would have to do so manually! And this is setting aside the fact that Fortran programs of the era weren’t entered on a keyboard, but painstakingly encoded and punched into hundreds of cards.

Structured programming was probably first implemented in Algol, a language designed in the late 1950s and into the 1960s. Although Algol itself wasn’t particularly popular, almost all modern procedural languages (C, Java, JavaScript, Python, etc.) can trace at least some history to Algol’s design. Importantly, Algol was designed. Fortran was changed and enhanced as needs arose, with no particular design motivation other than to serve its purpose. Algol, as well as contemporaries such as LISP and COBOL, were, at least at their inception, carefully designed to encourage a particular style of programming. This era was the beginning of programming languages having different paradigms.

Aside: In fact, however, some aspects of programming paradigms predate programming languages. In the 1930’s, the Church-Turing thesis unified two different models of computation: Church’s and Turing’s. Programming languages which followed Church’s philosophy of computing would later be known as functional languages (i.e., languages in the functional paradigm), while languages following Turing’s philosophy are imperative. The Church-Turing thesis itself proves that these paradigms are equivalently powerful, so the families of languages that they spawned differ not in what they can do, but in how one expresses what they do. We’ll see a bit more about the Church-Turing thesis in Module 2.

Fortran, Algol, COBOL, and many of their successors are imperative programming languages: their fundamental model of computing is a list of instructions which is run in the order that it appears, with any instruction able to change the state of the computer in a way that affects how the following instructions operate. Lisp, also developed in the late 1950’s, followed a different basic design: the basic unit of computation was the function, and data was usually encapsulated so that the same sort of state change, while possible, was not central. It was possibly the first functional programming language. In this context, of course, “functional” doesn’t mean “working”, it means that functions are first-class, in that they are values in the language. Lisp also pioneered the concept of homoiconicity: code and data having the same form. Lisp’s central datatype is the list, and Lisp functions are represented as nested lists. As such, Lisp code can manipulate Lisp code. This tradition continued in languages such as Scheme (and Racket), where the ability of code to manipulate code has led to extremely powerful macro systems.

Imperative and functional languages continued to develop greater sophistication while not changing the fundamental paradigm until object-oriented programming was invented in the 1980’s, and concurrent languages appeared around the same time.
4 Programming Paradigms

In introducing the history of programming languages, we’ve discussed imperative and functional programming. These are two of the programming language paradigms we will investigate in this course.

A programming paradigm is a mode of thought, and as such, it’s impossible to formally define. The edges of a programming paradigm are often unclear, and so it’s impossible to answer questions such as “is this object-oriented programming?” Nonetheless, as programming paradigms developed, programming languages developed to support those paradigms, and understanding programming paradigms is crucial to having a broad understanding of programming languages. Teaching the breadth of programming language paradigms is one of the fundamental goals of this course.

Because it’s impossible to define precisely what is or is not in a programming paradigm, we will use exemplars: languages which exemplify the concepts of a particular programming paradigm and, ideally, little else. As such, exemplar languages are chosen not necessarily because they’re especially practical programming languages; languages which are more flexible are often more versatile. Rather, they’re chosen because to understand an exemplar programming language is to understand the mode of thought behind the paradigm it exemplifies.

Aside: An exemplar is different from an example in that while an example of property X has property X, an exemplar of property X is a template or model for property X. For instance, although C++ is an object-oriented programming language, its history and its goals make it difficult to separate out the concepts of object-oriented programming from systems, imperative, procedural, structural programming, etc. Thus, C++ is an example of object-oriented programming, but not an exemplar.

It should come as no surprise that there is no perfect list of all the programming language paradigms that exist. The paradigms selected for this course are those of enduring, practical importance.

The languages you’re asked to program in, OCaml and Smalltalk, are an example and an exemplar of functional programming and object-oriented programming, respectively. In general, the assignments will be to implement a simple interpreter for an example of some paradigm of programming language, and you will be required to use the less similar of these two languages. For instance, you will be asked to implement a functional language in Smalltalk and an object-oriented language in OCaml. The reason for this isn’t just cruelty\(^2\): it is often easier to implement an interpreter for a programming language in a similar programming language because you can use the host language’s features in place of the guest language’s features, but taking this easy route does not help you to actually understand the paradigm. By using the “opposite” language, you will gain a more complete understanding of the paradigm. Note that OCaml has object-oriented features (that’s what the “O” stands for), but we won’t be using them in this course.

\(^2\)Note that I have not claimed that cruelty isn’t a reason.
5 OCaml and Smalltalk

The remainder of your responsibility in module 1 is to learn OCaml and Smalltalk. You don’t need to be an expert in either language, just competent to write some interpreters in them—by this point, you should be able to pick up new languages by their analogy to languages that you’re familiar with, and indeed the degree of similarity to languages you should know is why OCaml and Smalltalk were chosen. You will only be given a fairly brief introduction to each language; the programs you will be asked to implement don’t require any exotic libraries, and you are expected to be able to get comfortable enough with the languages to fulfill the requirements of the assignments without much more.

In order to be a more useful reference, both language introductions are in separate files, available on the course web site. Although in separate files, they are considered part of this module.

For OCaml, we will principally use a book available online, Real World OCaml, and in particular, the Guided Tour within it. Our introduction serves as an addendum to that discussing how we’ll do things specifically in this course.

6 Fin

In the next module, we will begin formalizing programming languages, by introducing the λ-calculus.

Rights

Copyright © 2020, 2021 Gregor Richards.
This module is intended for CS442 at University of Waterloo.
Any other use requires permission from the above named copyright holder(s).