Lecture 1: Introduction to Programming Languages: λ-Calculus

Ankush Das

January 22, 2025

1 Introduction

Let's begin with an age-old question: what is a programming language? Wikipedia states that "A programming language is a system of notation for writing computer programs.". In this course, we will explore how a programming language is so much more than a mere tool for writing programs.

Before delving into this exploration, let's return to another age-old question: what is the essence of programming? I claim that the most basic abstraction that a programming language provides capturing its core is the notion of *function*. In today's lecture, we will study this most basic abstraction and how powerful it is. We will study functions in the context of a simple and minimal programming language: the λ-calculus. This will allow us to study this concept in its full depth.

2 The λ-Calculus

The λ-calculus is one of the most foundational languages with a rich history. One of the (many) reasons it is special is due to the Church-Turing Thesis which establishes an equivalence between the λ-calculus and Turing machines, stating that any function that can be implemented using a Turing machine can be effectively computed in the λ-calculus.

Now, let's explore the λ-calculus in more detail. As we noted before, the main abstraction this language provides is *function*. How are these functions defined? Let's see a mathematical function first.

$$f(x) = x + 20 \qquad g(y) = y \times y$$

This describes how these functions operate. In the λ-calculus, these functions are defined as follows:

$$f = \lambda x.x + 20 \qquad g = \lambda y.y \times y$$

The general λ expression is written as $\lambda x.e$ where e is the function body.

The other general expression in the language is function application, written formally as $f\,e$ which means calling the function f on the expression e. We will see more examples soon.

3 Definition of λ-Calculus

Now, we will try to answer the fundamental question we asked at the start of the lecture: what is a programming language and how do we define it? I like to think of a programming language as a mathematical object (similar to other objects like circle, triangle, polynomial, etc.) which can be defined using the following two components:

- ***Syntax***: How are programs in this language written? This is similar to defining how to construct a polynomial.
- ***Semantics***: How do programs behave? This is similar to describing how to evaluate a polynomial.

In the remaining lecture, we will study the syntax and semantics of the λ-calculus. But before we tackle λ-calculus, we will look at a simpler language (which we called LL1: Language for Lecture 1) and define the syntax and semantics for that language.

4 LL1: Syntax and Semantics

This language has only two expressions: $\overline{n}$ which stands for a number (e.g., 5, 10, etc.) and $e_1 + e_2$ which stands for addition of two expressions e_1 and e_2. Therefore, the formal syntax is written as

$$\text{Expressions} \quad e ::= \overline{n} \mid e \oplus e$$

Expressions in this language will be written as $\overline{n_1} \oplus \overline{n_2} \oplus \ldots$ for a finite number of numbers.

To define the semantics, we need two judgments: $e \mapsto e'$ to define the expression e steps to e' in one step. But since some expressions will be a final expression (i.e., a value) and cannot step further, we introduce a terminal judgment called $e\ \mathsf{value}$ to indicate that e is a value.

The semantics are defined as

$$\frac{}{\overline{n}\ \mathsf{value}}\ \textsc{Val}$$

This means that all numbers are values. They cannot be evaluated any further.

Next comes addition. We need three rules for addition, one to step the lhs, one to step the rhs, and one that will finally add the two numbers.

$$\frac{e_1 \mapsto e_1'}{e_1 \oplus e_2 \mapsto e_1' \oplus e_2}\ \textsc{Add-L} \qquad \frac{e_1\ \mathsf{value} \qquad e_2 \mapsto e_2'}{e_1 \oplus e_2 \mapsto e_1 \oplus e_2'}\ \textsc{Add-R} \qquad \frac{}{\overline{n_1} \oplus \overline{n_2} \mapsto \overline{n_1 + n_2}}\ \textsc{Add-E}$$

The Add-L rule dictates the lhs. Similarly, the Add-R rule dictates stepping the rhs when the lhs has become a value. Once both the summands have become values, we just add the two summands. The $\oplus$ symbol denotes the usual addition in ordinary mathematics.

Big-Step Semantics

In class, we also covered big-step semantics:

$$\frac{}{\overline{n} \Downarrow \overline{n}}\ \textsc{ConstEval} \qquad \frac{e_1 \Downarrow \overline{n_1} \qquad e_2 \Downarrow \overline{n_2} \qquad \overline{n} = \overline{n_1 + n_2}}{e_1 \oplus e_2 \Downarrow \overline{n}}\ \textsc{AddEval}$$

Type System

Since there are no variables, the type rules are pretty simple.

$$\frac{}{\overline{n} : \mathsf{int}}\ \textsc{Const} \qquad \frac{e_1 : \mathsf{int} \qquad e_2 : \mathsf{int}}{e_1 \oplus e_2 : \mathsf{int}}\ \textsc{AddEval}$$