Functional Programming Language Design

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Nathan Mull**
for teaching CS320: Concepts of Programming Languages
at Boston University [2].
Content in this document is based on content provided by Mull.

Prerequisite Definitions

This text assumes that the reader has a basic understanding of programming languages and grade-school mathematics along with a fundamentals grasp of discrete mathematics. The following definitions are provided to ensure that the reader is familiar with the terminology used in this document.

Definition 0.1: Token

A **token** is a basic, indivisible unit of a programming language or formal grammar, representing a meaningful sequence of characters. Tokens are the smallest building blocks of syntax and are typically generated during the lexical analysis phase of a compiler or interpreter.

Examples of tokens include:

- `keywords`, such as `if`, `else`, and `while`.
- `identifiers`, such as `x`, `y`, and `myFunction`.
- `literals`, such as `42` or `"hello"`.
- `operators`, such as `+`, `-`, and `=`.
- `punctuation`, such as `(`, `)`, `{`, and `}`.

Tokens are distinct from characters, as they group characters into meaningful units based on the language's syntax.

Definition 0.2: Non-terminal and Terminal Symbols

Non-terminal symbols are placeholders used to represent abstract categories or structures in a language. They are expanded or replaced by other symbols (either terminal or non-terminal) as part of generating valid sentences in the language.

- **E.g.**, "Today is ⟨name⟩'s birthday!!!", where ⟨name⟩ is a non-terminal symbol, expected to be replaced by a terminal symbol (e.g., "Alice").

Terminal symbols are the basic, indivisible symbols in a formal grammar. They represent the actual characters or tokens that appear in the language and cannot be expanded further. For example:

- `+`, `1`, and `x` are terminal symbols in an arithmetic grammar.

Definition 0.3: Symbol "::="

he symbol `::=` is used in formal grammar notation, such as Backus-Naur Form (BNF), to mean "is defined as" or "can be expanded as". It is used to define the syntactic structure of a language by specifying how non-terminal symbols can be replaced or expanded into other symbols.

For example:

$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{number} \rangle$$

This states that the non-terminal symbol $\langle \text{expr} \rangle$ can be defined as either:

- An expression followed by a '+' and another expression, or
- A single number.

The pipe symbol ($|$) indicates alternatives, while the symbol $\Rightarrow$ is used to denote derivations, showing the step-by-step application of the grammar rules to expand non-terminals into terminals.

Correct Derivations:

- $\langle \text{expr} \rangle \Rightarrow \langle \text{expr} \rangle + \langle \text{number} \rangle$
- $\langle \text{expr} \rangle \Rightarrow 5 + \langle \text{number} \rangle$
- $\langle \text{expr} \rangle \Rightarrow 5 + 3$
- $\langle \text{number} \rangle \Rightarrow 8$
- $\langle \text{expr} \rangle \Rightarrow \langle \text{number} \rangle$
- $\langle \text{expr} \rangle \Rightarrow 8$

Incorrect Derivations:

- $8 \Rightarrow \langle \text{number} \rangle$
- $8 \Rightarrow 5 + \langle \text{number} \rangle$
- $8 \Rightarrow 5 + 3$

Incorrect derivations arise when the direction of derivation is reversed or when terminal symbols are treated as if they can be expanded further. Terminals, such as 8, cannot act as non-terminals and do not expand into other symbols.

Definition 0.4: Symbol ":="

The symbol := is used in programming and mathematics to denote "assignment" or "is assigned the value of". It represents the operation of giving a value to a variable or symbol.

For example:

$$x := 5$$

This means the variable x is assigned the value 5.

In some contexts, := is also used to indicate that a symbol is being defined, such as:

$$f(x) := x^2 + 1$$

This means the function $f(x)$ is defined as $x^2 + 1$.

— 1 —

Functional Programming

1.1 Introduction

Programming Languages (PL) from the perspective of a programmer can be thought of as:

- A tool for programming
- A text-based way of interacting with hardware/a computer
- A way of organizing and working with data

However **This text concerns the design of PLs**, not the sole use of them. It's the difference between knowing how to fly an aircraft vs. designing one. We instead **think in terms of mathematics**, describing and defining the specifications of our language. Our program some mathematical object, a function with strict inputs and outputs.

Definition 1.1: Well-formed Expression

An expression (sequence of symbols) that is constructed according to established rules (syntax), ensuring clear and unambiguous meaning.

Definition 1.2: Programming Language

A **Programming Language (PL)** consists of three main components:

- **Syntax:** Specifies the rules for constructing well-formed expressions or programs.
- **Type System:** Defines the properties and constraints of possible data and expressions.
- **Semantics:** Provides the meaning and behavior of programs or expressions during evaluation.

Example 1.1: Syntax for Addition

If e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1 + e_2$ is also a well-formed expression. We can formalize this using the following rule for expressions, where $\langle \text{expr} \rangle$ acts as a placeholder for arbitrary expressions:

$$\langle \text{expr} \rangle \ ::= \ \langle \text{expr} \rangle + \langle \text{expr} \rangle$$

■

Programmers may have some intuition about what a **variable** is, often thinking of it as a container for data. However, within this context, variables can represent entire expressions and are, in a sense, immutable.

Definition 1.3: Meta-variables

Meta-variables are placeholders that represent arbitrary expressions in a formal syntax. They are used to generalize the structure of expressions or programs within a language.

Example 1.2: Meta-variables:

An expression e could be represented as 3 (a literal) or $3 + 4$ (a compound expression). In this context, variables serve as shorthand for expressions rather than as containers for mutable data. ■

Before talking about types we must understand "**context**" when working with PLs.

Definition 1.4: Context and Typing Environment

In type theory, a context defines an environment which establishes data types for variables. In particular, an environment Γ is a set of ordered list of pairs $\langle x : \tau \rangle$, usually written as $x : \tau$, where x is a variable and τ is its type. We now write a **judgment**, a formal assertion about an expression or program within a given context. We denote:

$$\Gamma \vdash e : \tau$$

which reads "in the context Γ, the expression e has type τ". We may also write judgments for functions, denoting the type of the function and its arguments.

$$f : \tau_1, \tau_2, \ldots, \tau_n \to \tau$$

where f is a function taking n arguments $(\tau_1, \tau_2, \ldots, \tau_n)$, outputting the type τ.

[1]

Tip: Symbol names and command in LaTeX used above are as follows:

- Γ reads as "Gamma" (\Gamma).
- $\vdash$ reads as "turnstile" (\vdash).
- τ reads as "tau" (\tau).

Definition 1.5: Rule of Inference

In formal logic and type theory, an **inference rule** provides a formal structure for deriving conclusions from premises. Rules of inference are usually presented in a **standard form**:

$$\frac{\text{Premise}_1, \quad \text{Premise}_2, \quad \ldots, \quad \text{Premise}_n}{\text{Conclusion}} \ (\text{Name})$$

- **Premises (Numerator):** The conditions that must be met for the rule to apply.
- **Conclusion (Denominator):** The judgment derived when the premises are satisfied.
- **Name (Parentheses):** A label for referencing the rule. [3]

Now we may begin to create a type system for our language, starting with some basic rules.

Example 1.3: Typing Rule for Integer Addition

Consider the typing rule for integer addition for which the inference rule is written as:

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \ (\text{addInt})$$

This reads as, "If e_1 is an **int** (in the context Γ) and e_2 is an **int** (in the context Γ), then $e_1 + e_2$ is an **int** (in the same context Γ)".

Therefore: let $\Gamma = \{x : \text{int}, y : \text{int}\}$. Then the expression $x + y$ is well-typed as an **int**, since both x and y are integers in the context Γ. ■

Example 1.4: Typing Rule for Function Application

If f is a function of type $\tau_1 \to \tau_2$ and e is of type τ_1, then $f(e)$ is of type τ_2.

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash f(e) : \tau_2} \ (\text{appFunc})$$

This reads as, "If f is a function of type $\tau_1 \to \tau_2$ (in the context Γ) and e is of type τ_1 (in the context Γ), then $f(e)$ is of type τ_2 (in the same context Γ)".

Therefore: let $\Gamma = \{f : \text{int} \to \text{bool}, x : \text{int}\}$. Then the expression, $f(x)$, is well-typed as a **bool**, since f is a function that takes an integer and returns a boolean, and x is an integer in the context Γ. ■

Finally, we can define the semantics of our language, which describes the behavior of programs during evaluation:

Example 1.5: Evaluation Rule for Integer Addition (Semantics)

Consider the evaluation rule for integer addition. This rule specifies how the sum of two expressions is computed. If e_1 evaluates to the integer v_1 and e_2 evaluates to the integer v_2, then the expression $e_1 + e_2$ evaluates to the integer $v_1 + v_2$. The rule is written as:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 + v_2} \text{ (evalInt)}$$

Read as, "If e_1 evaluates to the integer v_1 and e_2 evaluates to the integer v_2, then $e_1 + e_2$ evaluates to $v_1 + v_2$."

Example Evaluation:

- $2 \Downarrow 2$
- $3 \Downarrow 3$
- $2 + 3 \Downarrow 5$
- $4 + 5 \Downarrow 9$
- $(2 + 4) + (4 + 5) \Downarrow 15$

Here, the integers 2 and 3 evaluate to themselves, and their sum evaluates to 5 based on the evaluation rule. Additionally e_1 could be a compound expression, such as $(2 + 4)$, which evaluates to 6. ■

1.2 Functional Programming with OCaml

In this section, we introduce **OCaml** as our programming language of choice for exploring the principles of **functional programming**. Functional programming emphasizes a declarative style, where programs describe *what to do* rather than *how to do it*, in contrast to the imperative programming paradigm, which many programmers are familiar with.

To better understand these differences, we will compare functional programming in OCaml with imperative programming in Python. This will give us a practical perspective on how the two paradigms approach problem-solving and help us appreciate the unique features of functional programming.

Bibliography

[1] Wikipedia contributors. Typing environment — wikipedia, the free encyclopedia, 2023. Accessed: 2023-10-01.

[2] Nathan Mull. Cs320: Concepts of programming languages. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.

[3] Wikipedia contributors. Rule of inference — Wikipedia, The Free Encyclopedia, 2025. [Online; accessed 22-January-2025].