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.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

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 $x^2 + 1$.

Definition 0.5: Substitution: $[v/x]e$

Formally, $[v/x]e$ denotes the substitution of v for x in the expression e. For example:

$$[3/x](x + x) = 3 + 3$$

This means that every occurrence of x in e is replaced with v.

— 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 Development Environment 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.

Definition 2.1: OCaml

OCaml is a general-purpose programming language from the ML family, known for its strong static type system, type inference, and support for functional, imperative, and object-oriented programming. It is widely used in areas like compilers, financial systems, and formal verification due to its safety, performance, and expressive syntax. The **Ocaml Extension** is `.ml`

In addition to using Ocaml we will use Dune and Opam.

Definition 2.2: Dune

Dune is a build system for **OCaml** projects, designed to simplify the compilation and management of code. It automates tasks such as building executables, libraries, and tests, while handling dependencies efficiently. Dune is widely used in the OCaml ecosystem due to its ease of use and minimal configuration.

Definition 2.3: OPAM

OPAM (OCaml Package Manager) is the standard package manager for the OCaml programming language. It simplifies the installation, management, and sharing of OCaml libraries and tools, providing developers with a convenient way to manage dependencies and project environments.

If you are familiar with **npm** or **yarn**, **OPAM** serves a similar purpose but is specifically designed for the **OCaml** ecosystem. Like npm and yarn, OPAM is a package manager that simplifies the installation and management of libraries and dependencies. Additionally, it offers features tailored to OCaml development, such as managing multiple compiler versions and isolating project environments.

Window Users: It may be easier to use WSL or a Linux VM to run OCaml and Dune rather than a native install. This text will use **Ubuntu** distro. If using WSL, make sure the terminal is running the distro, it will give you a fresh file system to work with. If you are a Mac user, you may use **Homebrew** to install OCaml and Dune.
WSL Installation: https://learn.microsoft.com/en-us/windows/wsl/setup/environment

Tip: If you plan to use github as your repository manager, you may have to create a personal access token to connect your account to your local machine.
Creating a Personal Access Token: https://docs.github.com/en/authentication/kee...

We use the terminal in this text, but an IDE could be used with additional setup.

Definition 2.4: Basic Terminal Commands

- **Navigation:**
  - cd <directory> : Change to a specified directory.
  - cd : Navigate to the home directory.
  - cd ../ : Move up one level in the directory hierarchy.
  - pwd : Print the current working directory.
- **Viewing and Listing:**
  - ls : List the contents of the current directory.
  - ls -l : Display detailed information about files and directories.
  - cat <file> : Display the contents of a file.
  - tree <directory> : Prettier ls -l , install: sudo apt install tree .
- **Creating:**
  - mkdir <directory> : Create a new directory.
  - touch <file> : Create an empty file.
- **Deleting:**
  - rm <file> : Delete a file.
  - rm -r <directory> : Delete a directory and its contents recursively.
- **Renaming and Moving:**
  - mv file.txt /path/to/new/directory/
  - mv <oldname> <newname> : Rename or move a file.
- **File Properties:**
  - chmod <permissions> <file> : Change the permissions of a file.
  - chmod u+rwx file.txt : Gives u (owner) **r**ead, **w**rite, and e**x**ecute permissions.
  - chmod g-w file.txt : Removes g (group) **w**rite permission.
  - file <file> : Determine the type of a file.

Vim will be our text-editor of choice. We will write code, and edit files using Vim.

Definition 2.5: Vim Common Commands

- **Starting and Exiting:**
  - `vim <file>`: Open or a create a file in Vim.
  - `:w`: Save (write) changes to the file.
  - `:q`: Quit Vim.
  - `:wq`: Save changes and quit Vim.
  - `:q!`: Quit without saving changes.
- **Modes:**
  - `i`: Switch to ***Insert Mode*** to start editing text.
  - `Esc`: Return to ***Normal Mode***, read-only mode for **navigation** and **commands**.
- **Navigation:**
  - `h` (left), `j` (down), `k` (up), `l` (right): Moves the cursor.
  - `:<line number>`: Jump to a specific line in the file.
  - `G`: Jump to the end of the file.
  - `gg`: Jump to the beginning of the file.
- **Editing:**
  - `x`: Delete the character under the cursor.
  - `dd`: Delete the current line.
  - `yy`: Copy (yank) the current line.
  - `p`: Paste copied or deleted text.
  - `u`: Undo the last change.
  - `Ctrl+r`: Redo the undone change.
- **Searching:**
  - `/text`: Search for `text` in the file.
  - `n`: Jump to the next occurrence of the search term.
  - `N`: Jump to the previous occurrence of the search term.

`:help`: for more Vim commands and options.

Preparing the Environment

Next we enable our machine to compile and run OCaml code. Choose a line below that corresponds to your operating system, and run it in the terminal.

Listing 1.1: Installing OPAM on Various Systems

```
# Homebrew (macOS)
brew install opam

# MacPort (macOS)
port install opam

# Ubuntu
apt install opam

# Debian
apt-get install opam

# Arch Linux
pacman -S opam
```

Before we can use OPAM to manage OCaml libraries and tools, we need to prepare the system by running the `opam init` command. This sets up OPAM by:

Listing 1.2: Initializing OPAM

```
# Initialize OPAM
opam init

# Configure your shell environment
eval $(opam env)

# Verify OPAM is ready to use
opam --version
```

After these steps, OPAM will be ready to manage OCaml dependencies, compilers, and project environments.

Important: With every new terminal, `eval $(opam env)` must be ran for OCaml use. Without it, the terminal might not recognize OPAM-installed tools or compilers.

Creating and Using an OPAM Switch

To manage different versions of OCaml and keep project dependencies isolated, OPAM provides a feature called a **switch**. A switch is an environment tied to a specific OCaml compiler version and a unique set of installed packages. This is especially useful for working on multiple projects with different requirements.

For this setup, we will create a new switch to ensure a clean environment with the required version of OCaml. Follow these steps:

Listing 1.3: Creating and Activating an OPAM Switch

```
# Step 1: Create a new switch named "my_switch" with OCaml version 5.2.1
opam switch create my_switch 5.2.1

# Step 2: Activate the newly created switch
opam switch my_switch

# Step 3: Update your terminal environment to reflect the switch
eval $(opam env)

# Step 4: Verify the switch is active
opam switch

# (Or / Optionally) Check the OCaml version
ocaml -version
```

Once these commands are executed, your terminal will be configured to use the OCaml version and environment defined by the switch `my_switch`.

Updating OPAM and Installing Essential Packages

After initializing OPAM and creating a switch, the next step is to update OPAM's package repository and install the tools we'll need for development. These packages provide essential utilities for OCaml programming and project management. Run the following commands:

Listing 1.4: Updating OPAM and Installing Packages

```
# Step 1: Update OPAM to fetch the latest package information
opam update

# Step 2: Install essential development tools
opam install dune utop ounit2 menhir ocaml-lsp-server

# Step 3: Install the custom library for this course
opam install stdlib320/.
```

Here's what each package does:

- `dune`: A modern build system for OCaml projects. It automates the compilation and management of OCaml code.
- `utop`: A user-friendly OCaml REPL (Read-Eval-Print Loop) for testing and experimenting with OCaml code interactively.
- `ounit2`: A testing framework for OCaml, similar to JUnit for Java, used for writing and running unit tests.
- `menhir`: A parser generator for OCaml, often used for developing compilers and interpreters.

- `ocaml-lsp-server`: A Language Server Protocol (LSP) implementation for OCaml, enabling features like autocompletion, type inference, and error checking in editors.
- `stdlib320/`: A custom library created for the CS320 course at Boston University by Nathan Mull. It provides It's a very small subset of the OCaml Standard Library with a bit more documentation. Documentation: https://nmmull.github.io/CS320/....

These will be the main tools used throughout this text.

Creating a Dune Project: Ocaml Introduction

To understand how `dune` structures projects and facilitates OCaml development, we'll create a simple project called `hello_dune`. This hands-on example will demonstrate the purpose of each folder and guide you through building, running, and testing an OCaml project.

Step 1: Prepare Your Environment

Before starting, ensure OPAM and your environment are set up. Run the following command to prepare the shell:

Listing 1.5: Preparing Your OPAM Environment

```
eval $(opam env)
```

This ensures that your terminal is configured correctly to work with OCaml and `dune`.

Step 2: Create the Project

Run the following commands to create a new `dune` project called `hello_dune`:

Listing 1.6: Creating the Project

```
mkdir demo # Create a new folder named hello_dune for our project
cd demo    # Move into the project directory
dune init project hello_dune # Initialize a new dune project
dune clean                   # Clean project from previous build files
```

This will generate the following project structure inside the `demo` folder:

Listing 1.7: Generated Project Structure

```
hello_dune/
|-- bin/         # Contains the main executable code
|-- lib/         # Contains reusable library code
|-- test/        # Contains test code
|-- dune-project         # Defines the project
|-- hello_dune.opam      # OPAM package definition
```

For now, we will focus on the `bin/` and `lib/` folders.

Step 3: Build and Verify the Project

To ensure everything is set up correctly, use the following command to build the project:

Listing 1.8: Building the Project

```
dune build
```

This command:

- Compiles the OCaml source files in your project.
- Resolves dependencies and ensures libraries and executables are built in the correct order.
- Creates a build cache to speed up future builds.
- Verifies that your project is configured correctly.

Important Notes:

- You must run `dune build` every time you make changes to your code to ensure the build reflects your edits.
- Running `dune build` from any subdirectory within redirect to the project root and build.
- If there are any issues (e.g., syntax errors, missing files, or incorrect configurations), `dune` will report them.

Step 4: Modify and Run the Program

To modify the program, first open the file `bin/main.ml` using Vim:

Listing 1.9: Opening the File in Vim

```
vim bin/main.ml
```

This opens the **main executable file** in the Vim editor. Once the file is open, press `i` to switch to *Insert Mode* and replace its contents with the following code:

Listing 1.10: Hello, Dune Program

```
let () = print_endline "Hello, Dune!"
```

After editing, press `Esc` to return to *Normal Mode*, then type `:wq` to save the changes and exit Vim. Now, run the program using the following command:

Listing 1.11: Running the Program

```
dune exec ./bin/main.exe
```

You should see the output:

```
Hello, Dune!
```

Step 5: Add a Library and Explore Its Use

The `lib/` folder is reserved for reusable code that can be shared across different parts of a project. In object-oriented programming languages like Java, this is analogous to creating static utility classes (e.g., a `Math` class for reusable mathematical functions).

1. Create a new file in the `lib/` folder. **Important:** The name of the file must match the project name. If your project is named `hello_dune`, the file should be named:

```
vim lib/hello_dune.ml
```

2. Add a reusable function to `lib/hello_dune.ml` (`^` concats strings, `+` is strictly for integers):

```
let greet name = "Hello, " ^ name ^ "!"
```

3. Verify or update the `lib/dune` file to expose the library. The `name` in the `dune` file should also match the project name:

```
(library
 (name hello_dune))
```

If this file is already configured with the above content, no changes are needed.

4. Interactively use the library in `utop`. To end a line in OCaml, use `;;`:

```
dune utop
```

Once inside `utop`, you can interact with the library:

Listing 1.12: Using the Library in Utop

```
Hello_dune.greet "Testing123";;
```

You should see the output:

```
 - : string = "Hello, Testing123!".
```

Important: Despite `lib/hello_dune.ml` being lowercase, it's referenced as `Hello_dune` in utop (capitalized). More on `utop` will be discussed later. But you may think of it as a calculator where we can access our functions and libraries.

5. To quit `utop`, type `#quit;;` or press `Ctrl+d`.

Listing 1.13: Quitting utop

```
#quit;;
```

6. We may also modify `bin/main.ml` to use the library:

Listing 1.14: Using the Library in Main

```
let () = print_endline (Hello_dune.greet "Library")
```

7. Build and run the program:

```
dune build
dune exec ./bin/main.exe
```

The output should now be:

```
Hello, Library!
```

What Are Dune Files?

As you explore the project, you'll notice `dune` files in various folders such as `bin/` and `lib/`. These files are configuration files used by the *Dune build system* to manage how your project is compiled and linked.

---

1. **Dune File in `lib/`:**

Listing 1.15: Library Dune File

```
(library
 (name hello_dune))
```

This file defines the `hello_dune` library. Dune compiles the code in `lib/hello_dune.ml` into a reusable module named `Hello_dune`, which can be used in other parts of the project.

2. **Dune File in `bin/`:**

Listing 1.16: Executable Dune File

```
(executable
 (public_name hello_dune)
 (name main)
 (libraries hello_dune))
```

This file specifies the executable program:

- `public_name hello_dune`: Defines the name of the program, which you can run with `dune exec hello_dune`.
- `name main`: Points to `bin/main.ml`, which serves as the entry point.
- `libraries hello_dune`: Links the `hello_dune` library to the executable.

Step 6: Adding Multiple Functions to a Library

The `lib/` folder can contain multiple functions to make the library more versatile and reusable. Instead of limiting the library to one function, we can define several functions in the same file and access them individually or collectively.
Steps to Add and Use Multiple Functions:

1. Modify the `lib/hello_dune.ml` file:

Listing 1.17: Adding Multiple Functions

```
(* Greets a person with their name. *)
let greet name = "Hello, " ^ name ^ "!"

(* Adds two integers. *)
let add x y = x + y

(* Multiplies two integers. *)
let multiply x y = x * y

(* Checks if x is divisible by y. *)
let is_divisible x y = x mod y = 0
```

Important: equality is `=` and not `==` in OCaml.

2. Use the functions interactively in `utop`:

```
dune utop
```

Now we can access the functions:

Listing 1.18: Accessing Functions in Utop

```
# Hello_dune.greet "OCaml";;
- : string = "Hello, OCaml!"

# Hello_dune.add 3 5;;
- : int = 8

# Hello_dune.multiply 4 6;;
- : int = 24

# Hello_dune.is_divisible 10 2;;
- : bool = true
```

Note: Every time the library is updated, `dune build` must run to reflect changes, and `utop` must be restarted to access the updated functions.

Alternatively, you can use the `open` command to avoid prefixing with `Hello_dune`:

Listing 1.19: Using the `open` Command

```
# open Hello_dune;;
# greet "Functional Programming";;
- : string = "Hello, Functional Programming!"

# add 7 2;;
- : int = 9
```

3. Update `bin/main.ml` to use the new functions:

Listing 1.20: Using the Library in Main

```
let () =
    let greeting = Hello_dune.greet "OCaml" in
    let sum = Hello_dune.add 3 5 in
    let product = Hello_dune.multiply 4 6 in
    let divisible = Hello_dune.is_divisible 10 2 in
    Printf.printf "%s\nSum: %d\nProduct: %d\nDivisible: %b\n"
    greeting sum product divisible
```

In a moment we will discuss `in`, for brevity you may think, “let `variable`, which is this `expression`, be substituted `in` this other `expression` respectively.” Let us continue.

4. Build and run the program:

```
dune build
dune exec ./bin/main.exe
```

You should see the output:

```
Hello, OCaml!
Sum: 8
Product: 24
Divisible: true
```

Take Note of The Above:

There are no semi-colons at the end of the lines. Though possible, that would make the code imperative, not functional. Again in functional programming, our code is one large expression. We add lets only to shorthand expressions, then carry them down the chain of expressions with `in`. We could have written the our entire expression in the print statement, but it would be harder to read and write. Hence, there is no idea of state, really emphasizing that,

“Code is one large equation, not a series of steps.”

5. Test the new functions by updating `test/test_hello_dune.ml`:

Listing 1.21: Adding Tests for Multiple Functions

```
let () =
    (* Test the greet function *)
    assert (Hello_dune.greet "OCaml" = "Hello, OCaml!");

    (* Test the add function *)
    assert (Hello_dune.add 3 5 = 8);

    (* Test the multiply function *)
    assert (Hello_dune.multiply 4 6 = 24);

    (* Test the is_divisible function *)
    assert (Hello_dune.is_divisible 10 2 = true);
    assert (Hello_dune.is_divisible 5 3 = false)
```

6. Verify or update the `test/dune` file to include the library:

Listing 1.22: Test Dune File

```
(test
 (name test_hello_dune)
 (libraries hello_dune))
```

7. Run the tests:

```
dune runtest
```

If all tests pass, there will be no errors. Otherwise, you will see detailed messages pointing to any failures. **Try** to make an error to see if your tests are working.

Onboarding Conclusion:

This concludes the onboarding process for OCaml and Dune. We have:

- Installed OCaml, Dune, and other essential tools.
- Created a new Dune project and explored its structure.
- Built, modified, and executed an OCaml program.
- Added a library and multiple functions with tests.

In the next section, we dive deeper into OCaml's syntax, features, and functional programming concepts.

1.3 Ocaml Basics: Syntax, Types, and Semantics

Strong Typing

OCaml is a strongly typed language, meaning that operations between incompatible types are not allowed. Additionally, the underscore (_) is used as a throwaway variable for values that are not intended to be used.

Listing 1.23: Example of Strong Typing

```
let x : int = 2
let y : string = "two"
let _ = x + y (* THIS IS NOT POSSIBLE *)
```

This will result in the following error:

Listing 1.24: Error Message

```
3 | let _ = x + y (* THIS IS NOT POSSIBLE *)
        ^
Error: This expression has type string but an expression was expected of
    type int
```

Demonstrating that, in OCaml, unlike other languages, operator overloading and implicit type conversions are not allowed. This means, no adding strings and integers, floats and integers, etc. There are separate operators for each type.

Basic Ocaml Operators:
Operators in OCaml behave just like other languages, with a few exceptions. Here are the basic operators at a quick glance:

Type	Literals Examples	Operators
int	0, -2, 13, -023	+, -, *, /, mod
float	3., -1.01	+., -., *., /.
bool	true, false	&&, ||, not
char	'b', 'c'	
string	"word", "@*&#"	^

Table 1.1: Basic OCaml Types, Literals, and Operators

For emphasis:

Definition 3.1: OCaml Operators

Operator Distinctions:
Operators for `int` and `float` are *different*. For example:

- `+` (integer addition)
- `+.` (float addition)
- `^` (string concatenation)

Moreover, the `mod` operator is used for integer division. This is to say that there is no implicit type conversion in OCaml.

No Operator Overloading:
OCaml has **no operator overloading**, meaning operators are strictly tied to specific types.

Comparison Operators:
Comparison operators are standard and can be used to compare expressions of the same type:

- `<`, `<=`, `>`, `>=`

Equality and Inequality:

- Equality check: `=`
- Inequality check: is `<>` **and not**, `!=`

Definition 3.2: OCaml (in) Keyword

Consider the expression below:

```
let x = 2 in x + x
```

The `in` keyword is used to bind the value of `x` to the expression `x + x`. This is a common pattern in OCaml. In a sense we are saying, " let x stand for 2 in the expression $x + x$."

This is similar to the prerequisite definition of the substitution operation (0.5). Mathematically, we can think of this as:

$$[2/x](x + x) = 2 + 2$$

Where the value of 2 is substituted for x in the expression $x + x$.

To illustrate this, Observe the diagram below:

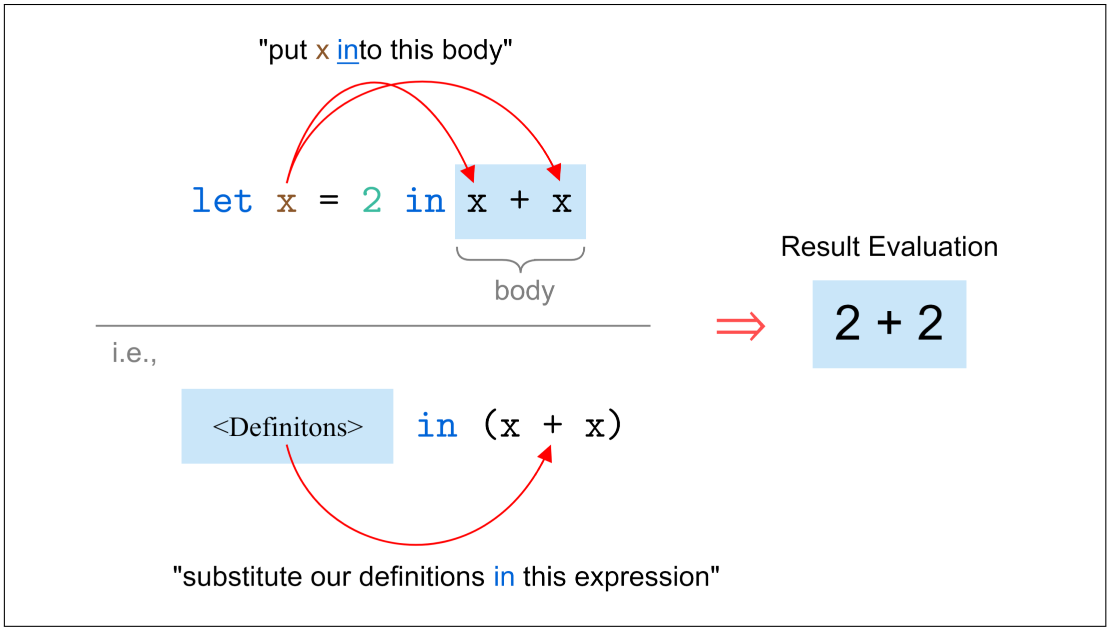

Figure 1.1: The `in` Keyword in OCaml

To disect the roles of **syntax**, **semantics**, and **types** in the expression `let x = 2 in x + x`:

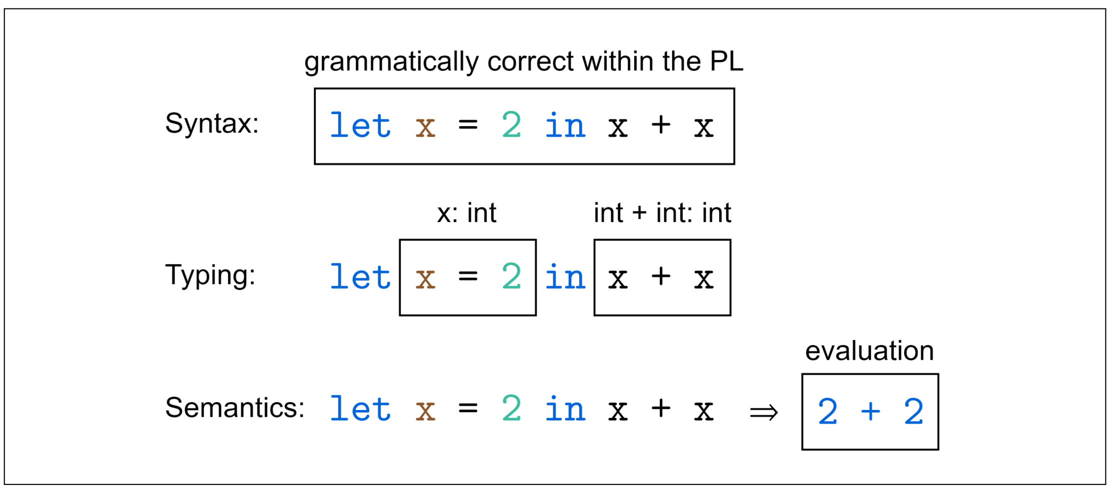

Figure 1.2: Syntax, Semantics, and Types in OCaml

- **Syntax**: The expression `let x = 2 in x + x` is a valid OCaml expression.
- **Typing**: Well-typed, as x is an `int` and $x + x$ is an `int`.
- **Semantics**: After substitution, the expression evaluates to $2 + 2$.

Definition 3.3: Whitespace Agnostic

Caml is **whitespace agnostic**, meaning that the interpreter does not rely on the presence or absence of whitespace to determine the structure of the code. Whitespace can be used freely for readability without affecting the semantics of the program. For example, the following expressions are equivalent:

Listing 1.25: Whitespace Agnostic Example

```
let x = 1 + 2
```

and

```
let x
= 1
  +
  2
```

Both produce the same result, as whitespace does not alter the meaning of the expression.

Understanding Functions in OCaml

In OCaml, functions do not require parentheses, arguments directly follow the function name. For example:

```
let add x y z= x + y + z in
let result = add 3 5 5
(* semantically evaluates to 3 + 5 + 5 *)
```

Here, the `add` function takes two arguments, `x` and `y`, which is substituted into `result` with arguments 3,5, and 5.

Definition 3.4: Anonymous Functions

An **anonymous function** is a one-time-use function that is not bound to a name. In OCaml, anonymous functions are created using the `fun` keyword. They are useful for passing functions as arguments to other functions or for defining functions locally. For example:

```
let add x y z = x + y z
```

is equivalent to:

```
let add = fun x -> fun y -> fun z -> x + y + z
```

These are formally known as **lambda expressions**, where in **lambda calculus** `fun x -> e` is written as "$\lambda x.e$", s.t., λ denotes the anonymous function, x the argument, and e the expression. The add function is equivalent to, $\lambda x.\lambda y.\lambda z.x + y + z$, in lambda calculus.

Functions with multiple arguments can be thought of as nested anonymous functions, where variables are passed down the chain of functions. To illustrate:

```
let add = fun x -> fun y -> fun z -> x + y + z
```

i.e.,

```
let add = fun x ->
               fun y ->
                    fun z ->
                              x + y + z
```

function use: `(add 3 4 5)` can be thought of as `(((add 3) 4) 5)`

Figure 1.3: Anonymous Functions in OCaml

Alternatively, we may illustrate an analogous example with scoped functions in a pseudo syntax:

```
fun(x){
    fun(y){
        fun(z){
            x + y + z
}}}
```

Figure 1.4: Anonymous Functions in OCaml

Where x is a local variable of the outer-most function within scope of the inner functions, and so on with y and z. This is the concept known as **currying**.

Tip: Lambda calculus was developed by **Alonzo Church** in the 1930s at Princeton University. Church was the doctoral advisor of **Alan Turing**, the creator of the Turing Machine (1936), a theoretical model that laid the groundwork for modern computation.

Curry functions were introduced by **Haskell Curry** around the 1940-1950s as he worked in the U.S. He expanded upon combinatory logic, emphasizing breaking down functions into a sequence of single-argument functions.

Definition 3.5: Curried Functions

A **curried function** is a function that, when applied to some arguments, returns another function that takes the remaining arguments. For example:

```
add x y = x + y
```

is internally equivalent to:

```
add = fun x -> (fun y -> x + y)
```

In OCaml, functions are **curried** by default. This means that a function of multiple arguments is treated as a sequence of single-argument functions.

We let `add` stand for `fun x -> fun y -> x + y`. Therefore in reality we are doing:

```
(fun x -> fun y -> x + y) 3 5
```

This is know as **Application**, as we are *applying* arguments to a function.

Definition 3.6: Application

Application is the process of applying arguments to a function. **Full application** is when all arguments are applied to a function. For example:

```
(fun x -> fun y -> x + y) 3 5
```

Here, the function `fun x -> fun y -> x + y` is fully applied to the arguments 3 and 5.

Partial application is when only some arguments are applied to a function, which evaluates to another function accepting the remaining arguments. For example:

```
(fun x -> fun y -> x + y) 3
```

Here, the function `fun x -> fun y -> x + y` is partially applied to the argument 3, resulting in a new function `fun y -> 3 + y`.

In Lambda Calculus we may represent this as:

$$(\lambda x.\lambda y.(x+y))\ 3\ 5 \to$$
$$(\lambda y.(3+y))\ 5 \to$$
$$(3+5)$$

In this process, arguments are sequentially applied to the corresponding variables.

If-Expressions

In OCaml, **if-expressions** are used to conditionally evaluate expressions. This behaves similarly to other PLs with a few distinctions.

Definition 3.7: Ocaml if-then-else

In OCaml, if statements follow the form: `if <condition> then <expr1> else <expr2>`, i.e., if the condition is true, then `expr1` is evaluated, else `expr2` is evaluated. For example:

Listing 1.26: If-Expression: Divisible by 2

```
fun x -> if x mod 2 = 0 then "even" else "odd"
```

Here, the anonymous function finds if x is divisible by 2, evaluating to `"even"` if true, or otherwise `"odd"`.

Typing: The `then` and `else` expressions must evaluate to the same type. So the following expression is **invalid**:

Listing 1.27: Invalid If-Expression

```
fun x -> if x mod 2 = 0 then "even" else 0 (* INVALID *)
```

Else If: In OCaml, there is no `else if` keyword. Instead, nested if-expressions are used to achieve the same effect.

Listing 1.28: Else If Example

```
fun x ->
    if x mod 3 = 0 then
        "divisible by 3"
    else if x mod 5 = 0 then
        "divisible by 5"
    else "not divisible by 2 or 3"
```

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].