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 ":="

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 0.4: 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. We may string multiple substitutions together, such as:

$$[3/x][4/y](x + y) = 3 + 4$$

Where x is replaced with 3 and y is replaced with 4.

— 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.

I.e., Syntax gives us meaning, Types tell us how it is used, and Semantics tell us what it does. Here is an example of defining the operator (+) for addition in a language:

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. ■

However, we can be a bit more concise using mathematic notation:

Definition 1.3: Production Rule (::=)

A **Production Rule** defines the syntax of a language by specifying how non-terminal symbols can be expanded into sequences of terminal and non-terminal symbols. It is denoted by the symbol ::=:

$$\langle \text{non-terminal symbol} \rangle ::= \langle \text{definition} \rangle$$

Where the left-hand side non-terminal symbol can be expanded/represented by the right-hand side definition. We may also define multiple rules for a single non-terminal symbol, separated by the pipe symbol (|):

$$\langle e_1 \rangle ::= \langle e_2 \rangle | \langle e_3 \rangle | \ldots | \langle e_n \rangle$$

Where $\langle e_1 \rangle$ can be expanded into $\langle e_2 \rangle$, $\langle e_3 \rangle$, $\ldots$, or $\langle e_n \rangle$.

Example 1.2: Production Rule

Here are some possible production rules:

- $\langle \text{date} \rangle ::= \langle \text{month} \rangle / \langle \text{year} \rangle$
- $\langle \text{year} \rangle ::=$ 2020 | 2021 | 2022 | 2023 | 2024 | 2025
- $\langle \text{month} \rangle ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 10 \mid 11 \mid 12$
- $\langle \text{OS} \rangle ::= \langle \text{Linux} \rangle \mid \langle \text{Windows} \rangle \mid \langle \text{MacOS} \rangle$

Incorrect Derivations: we cannot take a terminal symbol and expand it further:

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

Here 8 means the token 8, it cannot be expanded any further. ■

Now we can clean up our previous syntax for defining addition:

Example 1.3: Production Rule for Addition

Let $\langle \text{expr} \rangle$ be a non-terminal symbol representing a well-formed expression. Then,

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

I.e., "$\langle \text{expr} \rangle + \langle \text{expr} \rangle$" (right-hand side) is a valid "$\langle \text{expr} \rangle$" (left-hand side). ■

Now that we have defined syntax, we have to define its usage using types. We first cover variables, which have a slightly different meaning in this context than what most programmers are used to.

Definition 1.4: 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 1.4: 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. ■

Now we must understand "**context**" and "**judgments**" in the context of type theory:

Definition 1.5: Judgments

In type theory, a **judgment** is a formal assertion about an expression. This does not have to be a true assertion, but just a statement about the expression. For example: "Pigs can fly" is a judgment, regardless of its validity.

Definition 1.6: Context and Typing Environment

In type theory, a **context** defines an environment which establishes data types for variables. Say we have an environment Γ (Gamma) for which defines the context. This context Γ holds an ordered list of pairs. Say, $\langle x : \tau \rangle$, typically written as $x : \tau$, where x is a variable and τ (tau) is its type. We denote our judgment as such:

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

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 type τ. We may add temporary variables to the context. For example:

$$\Gamma, x : \text{int} \vdash e : \text{bool}$$

Reads, Given the context Γ with variable declaration (x :int) added, the expression e has type **bool**. [1]

Definition 1.7: 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 1.5: 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 1.6: 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 1.7: 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:

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

Read as, "If e_1 evaluates to the integer v_1 and e_2 evaluates to the integer v_2, then the expressions, 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 imperatively, *how to do it*.

When we begin programming in OCaml, we will skip features which are not necessary for functional programming. Though this text does cover some imperative features for curiosity sake, it is excludes many others. For full OCaml documentation visit: https://ocaml.org/docs/values-and-functions.

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.

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

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.

1.2.1 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.

1.2.2 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`.

1.2.3 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.

1.2.4 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 (0.4). 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:

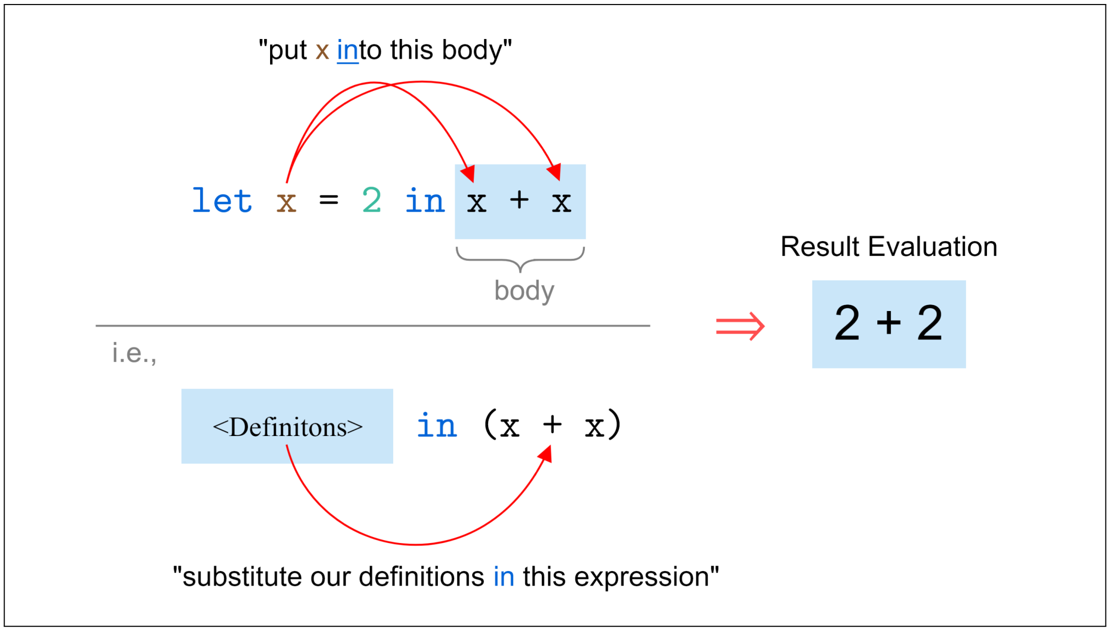

Figure 1.1: The `in` Keyword in OCaml

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

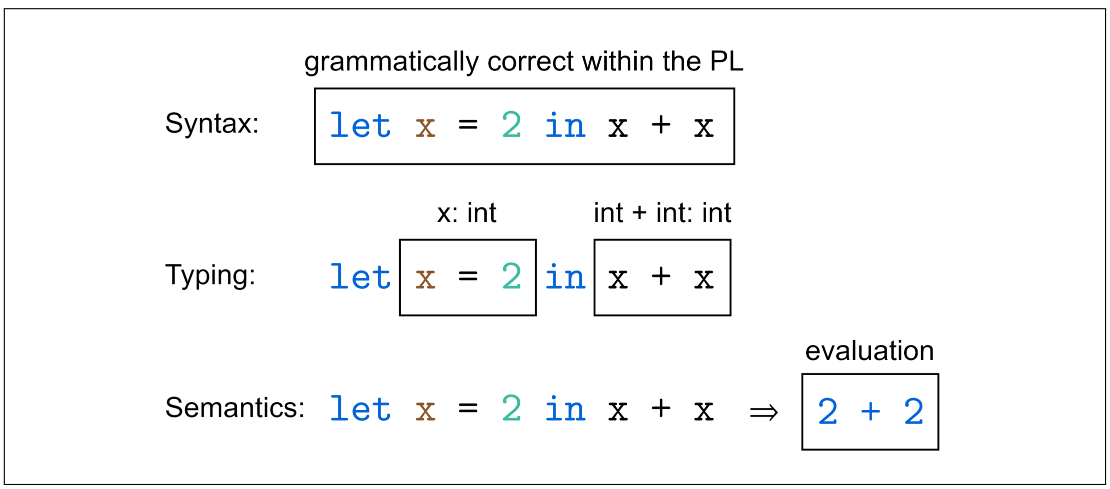

Figure 1.2: Syntax, Semantics, and Types in OCaml

- **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.

1.3.1 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

This works because of **closures**, where the inner functions have access to the variables of the outer functions. It's called **closures** because the variables are *enclosed* within another function's scope.

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

Figure 1.4: 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 **closures**.

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.

Definition 3.7: Side Effects in OCaml

A **side effect** refers to any change in the state of the program or its environment caused by a function or expression. This includes modifying variables, printing to the console, writing to a file, or interacting with external systems:

Listing 1.26: Function Without a Side Effect

```
let square x = x * x
(* This function computes the square of a number.
   It has no side effects as it doesn't change
   state or interact with the outside world. *)
```

Listing 1.27: Function With a Side Effect

```
let print_square x =
  let result = x * x in
  Printf.printf "The square of %d is %d\n" x result
(* This function prints the square of a number.
   It has a side effect: printing to the console. *)
```

Definition 3.8: The (unit) Type in OCaml

The `unit` type represents a value that carries no information. It is denoted by `()`, which is both the type and the sole value of `unit`. Functions returning `unit` are typically used for side effects.

Listing 1.28: Unit Type and Value

```
let x = ()
(* x has the type unit and the value (). *)
```

Listing 1.29: Function Returning Unit

```
let print_hello () = print_endline "Hello"
(* This function takes unit as an argument and returns unit. *)
```

The `unit` type ensures that functions used for their effects are explicit in their intent, making it clear they do not return meaningful data.

Definition 3.9: Void Functions in OCaml

Void functions are functions that perform actions but do not return meaningful values. These functions take `unit` as an argument and return `unit`, making their purpose explicit.

Listing 1.30: Void Function Example

```
let log_message () = print_endline "Logging message"
in log_message ()
(*Evaluates to "Logging message", which takes and returns a unit.*)
```

Void functions are often seen in the main entry point of an OCaml program:

Listing 1.31: Using `let ()` in the Main Function

```
let () =
print_endline "Program starting...";
(* Additional program logic here. *)
```

Definition 3.10: Skeleton Code

To write skeleton code one can use the `assert` keyword, though of course running this function will result in an error.

Listing 1.32: Skeleton Code Example

```
let skeleton () = assert false
(* This function is a placeholder for unwritten code. *)
```

Definition 3.11: Multiple Function Arguments

Applying expressions without parentheses may lead to unexpected results. For example:

Listing 1.33: Incorrect Function Application

```
(fun x y -> x + y) 3 5 * 2
(* Evaluates: 16 as ((fun x y -> x + y) 3 5) * 2 *)
```

As it takes the immediate arguments that are available, to avoid this, use parentheses to group expressions correctly:

Listing 1.34: Correct Function Application

```
(fun x y -> x + y) 3 (5 * 2)
(* Evaluates: 13 *)
```

1.3.2 If-Expressions

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

Definition 3.12: 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:

Listing 1.35: 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 1.36: 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 1.37: 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"
```

Definition 3.13: Conditional Assignment

A **conditional assignment** is a variable is assignment based off a condition:

Listing 1.38: Conditional Assignment

```
fun x -> let result = if x mod 2 = 0 then "even" else "odd"
in result
(* Evaluates result as "even" or "odd" depending on x *)
```

1.3.3 Type Hinting in OCaml

Type hinting is a way to explicitly specify the types of variables or function parameters in OCaml. This can be useful for documentation, readability, and debugging purposes.

Definition 3.14: Type Hinting in OCaml

In OCaml, **type hinting** allows programmers to explicitly specify the types of variables or function parameters. While type hints are not necessary due to OCaml's strong and static type system, they can help clarify intent and make code easier to understand, especially for larger projects.

Listing 1.39: Adding Type Annotations to Functions

```
let add (x : int) (y : int) : int = x + y
(* Explicitly states that x and y are integers, which results as an
    integer. *)
```

Listing 1.40: Adding Type Annotations to Variables

```
let name : string = "OCaml"
(* Explicitly states that name is a string. *)
```

Listing 1.41: Type Hinting in Anonymous Functions

```
(fun (x : float) (y : float) -> x *. y) 3.0 4.2;;
(* Multiplies two floats stating explicitly typing arguments as floats.
     *)
```

Though possibly adding redundancy, theoretically we may type any expression in OCaml.

Listing 1.42: Type Hinting in Conditional Expressions

```
((fun (x : int) ->
        if x mod 2 = 0 then "even"
        else "odd"
 ) : int -> string) 5
(* Evaluates to "odd" *)
```

1.3.4 OCaml Data Structures: Arrays, Lists, and Tuples

There are arrays, lists, and tuples in OCaml. While OCaml is not a purely functional language, we will treat it as such in this text.

Definition 3.15: Lists in OCaml

A **list** in OCaml is an ordered, immutable collection of elements of the same type, created via square brackets `[ ]` semicolon separated:

Listing 1.43: Defining a List

```
[1; 2; 3; 4]
(*Syntax: [e1;e2;e3;...;en] *)

[[1; 2]; [3; 4]; [5; 6]]
(* 2D list of type: int list list *)
```

Listing 1.44: Indexing a List & Finding Length

```
List.nth [1; 2; 3; 4] 2
(*Evaluates to 3; Syntax: List.nth <list> <index> *)

List.length [1; 2; 3; 4]
(*Evaluates to 4; Syntax: List.length <list> *)
```

Listing 1.45: Joining Lists

```
[1; 2] @ [3; 4]
(*Evaluates to [1; 2; 3; 4];
  Syntax: <list1> @ <list2> *)

1 :: [2; 3; 4]
(* ``::'' is the cons operator;
  Evaluates to [1; 2; 3; 4]; Syntax: <element> :: <list>
  Equiv. to: 1 :: 2 :: 3 :: 4 :: [] I.e., 1 :: (2 :: (3 :: (4 :: [])))
*)
```

Listing 1.46: Pattern Matching on Lists

```
let rec sum_list l =
    match l with
    | [] -> 0
    | x :: xs -> x + sum_list xs
in
sum_list [1; 2; 3; 4]
(* Evaluates to 10 as 1 + 2 + 3 + 4 *)
```

Definition 3.16: Arrays in OCaml

Arrays are a fixed-length random access (indexable) mutable collection of elements with the same type. They are created with brackets and vertical bars `[| |]`:

Listing 1.47: Defining and Modifying an Array

```
[|1; 2; 3; 4|]
(* Creates an array of integers: [|1; 2; 3; 4|] *)
```

Listing 1.48: 2D Array

```
[|[|1; 2|]; [|3; 4|]|]
(* Creates a 2D array of type: int array array *)
```

Listing 1.49: Arrays.make: Prefill Length Array

```
Array.make 3 0
(*Evaluates to [|0; 0; 0|];
  Syntax: Array.make <length> <initial_value> *)
```

Listing 1.50: Accessing Array Elements

```
let arr = [|1; 2; 3; 4|] in arr.(2)
(*Evaluates to 3;
  Syntax: <array>.(<index>) *)
```

Listing 1.51: Arrays.init: Creating Array with Function

```
Array.init 5 (fun i -> i * 2)
(*Evaluates to [|0; 2; 4; 6; 8|] where i is the index;
  Syntax: Array.init <length> <function> *)
```

Listing 1.52: Mutating Array Elements

```
let arr = [|1; 2; 3; 4|] in arr.(2) <- 5
(*Evaluates [|1; 2; 5; 4|];
  Syntax: <array>.(<index>) <- <new_value> *)
```

Listing 1.53: Length of Array

```
Array.length [|1; 2; 3; 4|]
(*Evaluates to 4;
  Syntax: Array.length <array> *)
```

Definition 3.17: Tuples in OCaml

A **tuple** in OCaml is an ordered collection of elements, where each element can have a different type. Tuples are immutable and their size is fixed. They are created using parentheses with elements separated by commas:

Listing 1.54: Defining a Tuple

```
(3, 4)
(*Syntax: (e1, e2, ..., en) *)
```

Listing 1.55: 2D Tuple

```
((1, 2), (3, 4))
(*
2D tuple of type: (int * int) * (int * int)
*)
```

Listing 1.56: Mixed Type Tuple

```
(3, "hello", true, 4.2)
(*
Mixed type tuple (3, "hello", true, 4.2): int * string * bool * float
*)
```

Listing 1.57: Accessing Tuple via Pattern Matching

```
match (3, 4) with (x, y) -> x + y
(*Evaluates to 7;
  Syntax: match <tuple> with (<pattern>) -> <expr> *)
```

More on `match` (Pattern Matching) in the next section.

Listing 1.58: Accessing Tuple via Decomposition

```
let (x, y) = (3, 4)
(*Evaluates to x = 3, y = 4;
  Syntax: let (<pattern>) = <tuple> *)
```

Note: There is no built-in functions to index or retrieve a length of a tuple in OCaml. Tuples are seen as a single entity, where pattern matching is typically utilized to access elements.

1.3.5 Pattern Matching & Switch-Case Absence

In OCaml, **There is no switch-case statement**, pattern matching is used instead:

Definition 3.18: OCaml Pattern Matching (match ... with ...)

Pattern matching in OCaml is a mechanism for inspecting and deconstructing data based on its structure. The `match ... with` expression evaluates a value and compares it against a series of patterns, executing the first matching case. Its syntax is as follows:

Listing 1.59: Pattern Matching Syntax

```
match <expression> with
| <pattern1> -> <result1>
| <pattern2> -> <result2>
| ...
| <patternN> -> <resultN>
```

Here, `<expression>` is the value being evaluated, and each `<pattern>` represents a condition or structure to match.

Listing 1.60: Matching an Integer

```
fun x ->
  match x with
  | 0 -> "zero"
  | 1 -> "one"
  | 2 | 3 -> "two or three"
  | _ -> "other";;

describe_number 0;; (* Evaluates to "zero" *)
describe_number 5;; (* Evaluates to "other" *)
```

Listing 1.61: Matching Multiple Arguments

```
fun x y ->
  match (x, y) with
  | (0, 0) -> "origin"
  | (0, _) -> "x-axis"
  | (_, 0) -> "y-axis"
  | _ -> "other"
(* Utilizing a tuple to match multiple arguments *)
```

- **Underscore (_):** A wildcard pattern that matches anything not explicitly listed.
- **Multiple Patterns:** Separate patterns with `|` to match multiple cases.
- **Deconstruction:** Use pattern matching to extract values from compound data structures such as tuples, lists, or variants.

1.3.6 Looping: Recursion, Tail-End Recursion, Mutually Recursive (and)

In functional programming, looping is typically achieved through **recursion**. Unlike imperative programming, where loops rely on mutable state, recursion allows us to iterate by repeatedly calling a function while unravelling our expressions. While OCaml is not a purely functional language and does provide `for` and `while` loops, in the context of this text, we will only use recursion for looping.

Definition 3.19: Ocaml Recursion

Recursion is the process of a function calling itself. Any and all recursive functions need the keyword `rec` to be used in OCaml:

Listing 1.62: Summing to n Using Recursion

```
let rec sum_to_n n =
    if n = 0 then 0
    else n + sum_to_n (n - 1)
in
sum_to_n 5
(* Evaluates to 15 as 5 + 4 + 3 + 2 + 1 + 0 *)
```

Listing 1.63: Fibonacci Sequence Using Recursion

```
let rec fibonacci n =
    if n <= 1 then n
    else fibonacci (n - 1) + fibonacci (n - 2)
in
fibonacci 6
(* Evaluates to 8 as 0, 1, 1, 2, 3, 5, 8 *)
```

Listing 1.64: Sum an Array of Integers Using Recursion

```
let rec sum_arr arr i =
    if i <= 0 then 0
    else arr.(i - 1) + sum_arr arr (i - 1)
in
let my_array = [|1; 2; 3; 4; 5|] in
sum_arr my_array (Array.length my_array)
(* Evaluates to 15 as 1 + 2 + 3 + 4 + 5 *)
```

Definition 3.20: Tail-End Recursion

Tail-end recursion refers to a type of recursion where the recursive call is the *last operation* performed in the function. This allows the OCaml compiler to reuse the same stack frame through **tail call optimization (TCO)**, helping avoid stack overflow errors.

Listing 1.65: Non-Tail-Recursive Factorial Function

```
let rec factorial n =
    if n <= 1 then 1
    else n * factorial (n - 1)
(* The multiplication (n * ...) occurs after the recursive call. *)
```

The above function **is not tail-recursive** as for `n` to be multiplied by the result of `factorial (n - 1)`. The recursive call must be evaluated first to get an answer, forcing us to track of intermediate stacks possibly leading to a stack overflow. Instead, we can pass an accumulated result as an argument to the function known as the **accumulator**:

Listing 1.66: Tail-Recursive Factorial Function

```
let rec factorial n acc =
    if n <= 1 then acc
    else factorial (n - 1) (n * acc)
(* The recursive call is the last operation performed. *)
```

To use the above we can call `... in factorial n 1` to start the recursion with an initial accumulator of 1. To abstract this, we can define a auxiliary helper function to hide the accumulator from the user:

Listing 1.67: Tail-Recursive Factorial Function with Helper

```
let factorial n =
    let rec aux n acc =
        if n <= 1 then acc
        else aux (n - 1) (n * acc)
    in aux n 1
in factorial 5
(* Evaluates to 120 as 5! = 5 * 4 * 3 * 2 * 1 *)
```

Listing 1.68: Tail-Recursive Summation of Positive Even Numbers

```
let rec sum_even n acc =
    if n <= 0 then acc
    else if n mod 2 = 0 then sum_even (n - 1) (acc + n)
    else sum_even (n - 1) acc
in sum_even 5 0
(* Evaluates to 6 as 4 + 2 + 0 = 6. *)
(* Despite two recursive calls they are independent of each other. *)
```

Definition 3.21: Mutually Recursive Functions (`and`)

Mutually recursive functions are functions that call each other in a cycle. Instead of defining them separately, we use the keyword `and` to define them simultaneously.

For example, we can define a pair of functions that determine whether a number is even or odd using mutual recursion:

Listing 1.69: Mutually Recursive Even and Odd Functions

```
let rec is_even n =
    if n = 0 then true
    else is_odd (n - 1)
and is_odd n =
    if n = 0 then false
    else is_even (n - 1)
in is_even 4
(* Evaluates to true as 4 is even. *)
```

In this example, `is_even` calls `is_odd` and vice versa. The recursion stops when $n = 0$, where we return either `true` (for even) or `false` (for odd).

1.3.7 Strings, Characters, and Printing in OCaml

Strings, characters, and printing behave much the same as in other languages, with some unique functions and syntax in OCaml.

Definition 3.22: Strings in OCaml

Strings in OCaml are immutable and allow for various operations, such as concatenation, slicing, and transformation:

Listing 1.70: Creating and Concatenating Strings

```
let greeting = "Hello" ^ " " ^ "World!"
(* Evaluates to "Hello World!";
Syntax: <string1> ^ <string2> *)
```

Listing 1.71: Getting the Length of a String

```
String.length "Hello"
(* Evaluates to 5;
Syntax: String.length <string> *)
```

Listing 1.72: Accessing a Character in a String

```
"Hello".[1]
(* Evaluates to 'e';
Syntax: <string>.[<index>] *)
```

Listing 1.73: Slicing a String

```
String.sub "Hello World" 6 3
(* Evaluates to "Wor";
Syntax: String.sub <string> <start> <length> *)
```

Listing 1.74: Converting to Uppercase

```
String.uppercase_ascii "hello"
(* Evaluates to "HELLO";
Syntax: String.uppercase_ascii <string> *)
```

Listing 1.75: String Equality

```
"hello" = "world" (* Evaluates to false*)
"hello" = "hello" (* Evaluates to true*)
```

Definition 3.23: Characters in OCaml

Characters in OCaml are individual elements of strings and are represented using single quotes (e.g., 'a'). Unlike strings, characters are immutable single units that cannot be directly concatenated or manipulated as strings:

Listing 1.76: Defining Characters

```
let char_a = 'a'
(* A character is represented with single quotes. *)
```

Listing 1.77: Comparing Characters

```
'a' < 'b'
(* Evaluates to true;
   Syntax: <char1> < <char2> *)
```

Listing 1.78: Converting Characters to Strings

```
Char.escaped 'a'
(* Evaluates to "a";
   Syntax: Char.escaped <char> *)
```

Listing 1.79: Converting Characters to Integers

```
Char.code 'a'
(* Evaluates to 97 (ASCII code of 'a');
   Syntax: Char.code <char> *)
```

Listing 1.80: Converting Integers to Characters

```
Char.chr 97
(* Evaluates to 'a' (ASCII character for 97);
   Syntax: Char.chr <int> *)
```

Listing 1.81: Checking if a Character is Uppercase

```
Char.uppercase_ascii 'a'
(* Evaluates to 'A';
   Syntax: Char.uppercase_ascii <char> *)
```

Definition 3.24: Printing in OCaml

Displays information on the terminal, typically for debugging or interacting with users:

Listing 1.82: Printing a String

```
print_endline "Hello, World!"
(* Prints "Hello, World!" followed by a newline;
   Syntax: print_endline <string> *)
```

Listing 1.83: Printing Integers, Floats, and Characters

```
print_int 42; print_newline ()
(* Prints "42" followed by a newline;
   Syntax: print_int <int>; print_newline () *)

print_float 3.14159; print_newline ()
(* Prints "3.14159" followed by a newline;
    Syntax: print_float <float>; print_newline () *)

print_char 'A'; print_newline ()
(* Prints "A" followed by a newline;
    Syntax: print_char <char>; print_newline () *)
```

Note: If strictly using `stdlib320` then `Printf` is not available.

Listing 1.84: Formatted Printing with Printf

```
Printf.printf "The number is: %d\n" 42
(* Prints "The number is: 42" with formatted output;
   Syntax: Printf.printf <format> <value> *)
```

Listing 1.85: Formatted Printing for Floats

```
Printf.printf "Pi is approximately: %.2f\n" 3.14159
(* Prints "Pi is approximately: 3.14" with 2 decimal places;
   Syntax: Printf.printf <format> <float> *)
```

Listing 1.86: Printing Multiple Values

```
Printf.printf "Name: %s, Age: %d\n" "Alice" 30
(* Prints "Name: Alice, Age: 30" with formatted output;
   Syntax: Printf.printf <format> <value1> <value2> *)
```

Listing 1.87: Using Printf.eprintf for Error Messages

```
Printf.eprintf "Error: %s\n" "File not found"
(* Prints "Error: File not found" to the standard error output. *)
```

1.3.8 Conversions in OCaml

In OCaml implicit type conversions are not allowed, so we must explicitly convert between types when needed:

Definition 3.25: Common Conversions in OCaml

Listing 1.88: Integer and Float Conversions

```
float_of_int : int -> float

int_of_float : float -> int
```

Listing 1.89: Character and Integer Conversions

```
int_of_char : char -> int

char_of_int : int -> char
```

Listing 1.90: Boolean and String Conversions

```
string_of_bool : bool -> string

bool_of_string : string -> bool

bool_of_string_opt : string -> bool option
```

Listing 1.91: String and Integer Conversions

```
string_of_int : int -> string

int_of_string : string -> int

int_of_string_opt : string -> int option
```

Listing 1.92: String and Float Conversions

```
string_of_float : float -> string

float_of_string : string -> float

float_of_string_opt : string -> float option
```

1.3.9 Defining Custom Types: Variants and Records

In OCaml, we can define new types and structures to better organize our data and improve readability. This is especially useful when working with complex data structures. In particular, **records** provide a more user-friendly alternative to tuples by allowing named fields, while **variants** let us define custom types that can hold different kinds of values.

Definition 3.26: Variants in OCaml

A **variant** is a custom type that can take multiple forms. This is similar to an **enumeration** or a **sum type** in other languages. Variants are defined using the `type` keyword, followed by multiple constructors.

Listing 1.93: Defining and Using Variants Correctly

```
(* Define a variant type for operating systems *)
    type os =
    | Windows
    | MacOS
    | Linux
    | BSD // <-- Constructor

    (* Function to describe the OS with explicit typing *)
    let describe_os (system : os) : string =
    match system with
    | Windows -> "A proprietary OS developed by Microsoft."
    | MacOS -> "A Unix-based OS developed by Apple."
    | Linux -> "An open-source OS based on the Linux kernel."
    | BSD -> "A Unix-like OS known for its security and stability."

    (* Correct Usage *)
    let my_os : os = Linux
    let description = describe_os my_os
    (* Evaluates to: "An open-source OS based on the Linux kernel." *)

    (* Incorrect Usage *)
    let unknown_os = Solaris
    (* Error: Unbound constructor Solaris *)
```

Definition 3.27: Records in OCaml

A **record** is a immutable data structure that groups multiple values together. Each value is associated with a field name and accessors.

Listing 1.94: Defining and Using Records

```
(* Define a record type for a person *)
type person = {
name : string;
age : int;
}

(* Function to greet a person with explicit typing *)
let greeting =
    let greet (p : person) : string =
        "Hello, " ^ p.name ^ "! You are " ^ string_of_int p.age ^ "
            years old."
    in
    let alice : person = { name = "Alice"; age = 30 } in
    greet alice
(* Evaluates to: "Hello, Alice! You are 30 years old." *)
```

Listing 1.95: Incorrect Usage Example

```
...
    (* Attempting to create a record with a missing field *)
    let bob = { name = "Bob" }
    (* Error: Some record fields are undefined (age is missing). *)

    (* Using an incorrect type for a field *)
    let carol = { name = "Carol"; age = "twenty-five" }
    (* Error: This expression has type string but an expression was
        expected of type int. *)
```

Listing 1.96: Updating Records

```
...
(fun name ->
    let alice = { name = "Alice"; age = 30 } in
    let _ = print_endline ("Original: " ^ alice.name) in
    let alice = { alice with name = name } in
    let _ = print_endline ("Updated: " ^ alice.name)
) "Bob"
(* Prints:
Original: Alice
Updated: Bob
*)
```

Definition 3.28: Variant (of) Keyword

We can define variant types to refer to a specific record type:

Listing 1.97: Variants Carrying Record Fields

```
(* Defining a Record *)
type sep_Rect = { base: float; height: float }

(* Defining a Variant *)
type shape =
  | Circle of float
  | Rect of sep_Rect
  | Triangle of { sides: float * float; angle: float }
(* Triangle is of record type with two fields: sides and angle *)

(* Function to calculate the area of a shape *)
let area (s : shape) =
    match s with
    | Rect r -> r.base *. r.height
    | Triangle { sides = (a, b) ; angle } -> Float.sin angle *. a *. b
    | Circle r -> r *. r *. Float.pi
in
let my_shape : shape = Triangle { sides = (3.0, 4.0); angle = 2.0 }
in area my_shape
(* short handing ``angle=angle'' to ``angle'' (field punning) *)
(* Evaluates to 10.9115691219081796 *)
```

Definition 3.29: Type Checking in OCaml

In OCaml, **type checking** is performed at **compile-time**, meaning we can't check types at runtime. Therefore we must use variants and pattern matching to conditionally check types:

Listing 1.98: Using Variants for Type Checking

```
type int_or_string =
    | Int of int
    | String of string

let typeof t =
    match t with
    | Int x -> print_endline (string_of_int x ^ ": is Integer")
    | String x -> print_endline (x ^ ": is String")

let _ = typeof (Int 42)     (* Output: 42: is Integer *)
let _ = typeof (String "hi") (* Output: hi: is String *)
```

1.3.10 Handling Absence of Values with Options

In many programming languages, you might be familiar with the concept of a **null** value to represent the absence of a value. However, using `null` can lead to unexpected runtime errors if not handled carefully. OCaml avoids this pitfall by using the built-in **option** type, which forces you to explicitly handle the case when a value is absent.

Definition 3.30: Option Type in OCaml

The **option** type is a variant that can either hold a value of a given type or no value at all. Its definition is built into OCaml and is equivalent to:

Listing 1.99: Definition of the Option Type

```
type 'a option =
| None
| Some of 'a
```

A value of type `'a option` is either:

- `Some x`, indicating that a value `x` of type `'a` is present, or
- `None`, indicating the absence of a value.

Example 3.1: A Simple Example: Safe Division

Consider the problem of performing division, where division by zero is undefined. Instead of returning a special error code or risking a runtime error, we use the option type to signal that a valid result might not always be available.

Listing 1.100: Safe Division Using Options

```
let safe_divide (x : int) (y : int) : int option =
if y = 0 then None
else Some (x / y)

(* Usage Examples *)
let result1 = safe_divide 10 2   (* Evaluates to Some 5 *)
let result2 = safe_divide 10 0   (* Evaluates to None *)
```

In this example, `safe_divide` returns `Some result` when division is possible, and `None` when division by zero is attempted. This forces the caller to handle both cases explicitly. ■

Example 3.2: Avoiding "Null" in OCaml

In some languages, you might try to define a type that mixes an actual value with a `null`-like value. For example, one might be tempted to write:

Listing 1.101: Incorrect Approach with a "Null" Value

```
(* Incorrect: attempting to mix an int with a null literal *)
type example = int | "null"
```

OCaml's type system does not allow this kind of mixing. Instead, we define such types using the `option` type:

Listing 1.102: Correct Approach Using Option

```
type example = int option

(* Here, Some 42 represents an int value,
and None represents the absence of a value. *)
let value1 : example = Some 42
let value2 : example = None
```

Using `option` ensures that every time you work with a value that might be absent, you must handle the `None` case, making your code safer and more robust. ■

Example 3.3: Handling Optional Values When Summing Integers

Suppose we have a function that adds the head of a typed list to some sum.

Listing 1.103: Incorrect Approach for Null case

```
(* Here, suppose next is of some arbitrary strong type *)
let add_next sum (next: MyType) =
  match next with
  | [] -> sum (* Error: [] is not a valid constructor for MyType *)
  | x :: _  -> sum + x
```

Instead we should use `None` to represent the absence of a value, adding `option` to the type:

Listing 1.104: Correct: Using None for an Option

```
let add_next sum (next: MyType option) =
  match next with
  | None -> sum  (* Correctly handles the null case *)
  | Some x  -> sum + x
```

■

1.4 Defining Operators: infix and prefix operators

In OCaml, operators are defined as functions, and we can define our own operators using the `let` keyword.

Definition 4.1: Infix and Prefix Operators in OCaml

Infix Operators: An infix operator is a function that appears between its arguments. In OCaml, certain symbols can be used as infix operators. For example, addition (+) and multiplication (*) are built-in infix operators. The list of allowed infix is limited to:

```
! $ % & * + - . / : < = > ? @ ^ | ~
```

Operators cannot start with alphanumeric characters (a-z, A-Z, 0-9) and the following:

```
! ? ~ :
```

Though this might require trial and error depending on the pre-existing operators present in the Ocaml project.

Listing 1.105: Defining an Infix Operator

```
(* Define a custom infix operator *)
let ( +++ ) x y = x + y

let result = 4 +++ 5;;  (* result = 9 *)
```

To use an infix operator as a function, enclose it in parentheses:

```
let add = ( + );;
let sum = add 3 4;;  (* sum = 7 *)
```

Prefix Operators: A prefix operator is a function that appears before its argument. Most functions in OCaml are naturally prefix functions. However, some operators like - (negation) and ! (dereferencing) are built-in prefix operators.

Listing 1.106: Defining a Prefix Operator

```
(* Define a custom prefix operator *)
let (!@!) x = x * x;;

let !@! x;;  (* 16 *)
```

1.4.1 Print Debugging

To debug our functions we can use the print statements. For example:

Listing 1.107: Print Debugging

```
let rec factorial n acc =
    if n <= 1 then acc
    else
        let _ = Printf.printf "n: %d, acc: %d\n" n acc in
        factorial (n - 1) (n * acc)
in factorial 5 1
```

Without the `stdlib` we can do the following:

Listing 1.108: Print Debugging Without Printf

```
let rec factorial n acc =
    if n <= 1 then acc
    else
        let _ = print_endline ("n: " ^ string_of_int n ^ ", acc: " ^
            string_of_int acc) in
        factorial (n - 1) (n * acc)
in factorial 5 1
```

Perhaps a quick print without care of formatting:

Listing 1.109: Dirty Print Debugging

```
let rec factorial n acc =
    if n <= 1 then acc
    else
        let _ = print_int n in
        let _ = print_string ", " in
        let _ = print_int acc in
        factorial (n - 1) (n * acc)
in factorial 5 1
```

Section Exercises:

Exercise 4.1: Write an OCaml function that takes an integer x and evaluates `"positive"` if x is positive, `"negative"` if x is negative, and `"zero"` if x is zero.

Exercise 4.2: Write an OCaml function that takes an integer x and evaluates to the first digit of x using only integer arithmetic operations.

Exercise 4.3: Write an OCaml function that fixes the previous **Else If** function to evaluate to `"divisible by 3 and 5"` if x is divisible by both 3 and 5.

Exercise 4.4: Write an OCaml function implementation of the Fibonacci sequence using tail-end recursion.

1.5 Formalizing Ocaml Expressions

1.5.1 Basic Ocaml Expressions

Now we can begin to formalize expressions in OCaml. We again re-iterate what steps are needed to build expressions in our language, given that we have some *context* now.

Definition 5.1: Building Expressions

When creating new expressions we must follow these steps:

1. **Context:** Define variable-to-type mappings.
2. **Syntax:** Establish how the expression/operation should be written.
3. **Typing Rules:** Define the type of the whole expression and its sub-expressions.
4. **Semantics:** Clarify the resulting value/evaluation of the defined expression.

I.e., what are our types, how are they used, what type of data do they represent, and how does it evaluate?

Now we begin to formalize, though we will abstract the context to Γ, assuming all the types we've defined before (1.1).

Definition 5.2: Formalizing Let-Expressions

Let Γ be the OCaml context, and $=$ be mathematical equality, and `=` be an OCaml token:

- **Syntax:** $\langle\text{expr}\rangle ::= \texttt{let}\ \langle\text{var}\rangle\ \texttt{=}\ \langle\text{expr}\rangle\ \texttt{in}\ \langle\text{expr}\rangle$

  If x is a valid variable name, and e_1 is a well-formed expression and e_2 is a well-formed expression. Then `let` x `=` e_1 `in` e_2 is also a well-formed expression.

- **Typing-Rule:** $\dfrac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau}$

  Given context Γ, if there's some well-formed expression e_1 of type τ_1 and some well-formed expression e_2 of type τ, given a variable declaration of x of type τ_1, then within this context, the expression `let` x `=` e_1 `in` e_2 is of type τ.

- **Semantics:** $\dfrac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow v}$

  Following our context Γ, if a well-formed expression e_1 evaluates to v_1 and the substitution of v_1 for variable x in another well-formed expression e_2 evaluates to v, then the expression `let` x `=` e_1 `in` e_2 evaluates to v.

Before we continue, we introduce the concept of $\top$ and $\bot$.

Definition 5.3: Top and Bottom ($\top$, $\bot$)

In logic and computer science:

- $\top$ is used to represent *true*, *valid*.
- $\bot$ is used to represent *false* or *invalid*.

Specifically, they are the greatest and least element of a lattice/boolean algebra (hence top and bottom), which when it comes to logic means truthhood and falsehood.

We continue with the formalization of the `if` expression in OCaml.

Definition 5.4: Formalizing If-Expressions

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle ::= \texttt{if}\ \langle\text{expr}\rangle\ \texttt{then}\ \langle\text{expr}\rangle\ \texttt{else}\ \langle\text{expr}\rangle$

  If e_1 is a well-formed expression, e_2 is a well-formed expression, and e_3 is a well-formed expression, then `if` e_1 `then` e_2 `else` e_3 is also a well-formed expression.

- **Typing-Rule:** $$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

  Given context Γ, let there be well-formed expressions, e_1 of type `bool`, e_2 of type τ, and e_3 of type τ. Then the expression `if` e_1 `then` e_2 `else` e_3 is of type τ.

- **Semantics:** $$\frac{e_1 \Downarrow \top \quad e_2 \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}\ (\text{trueCond.})$$

  Following our context Γ, if a well-formed expression e_1 evaluates $\top$ and another well-formed expression e_2 evaluates to v, then the expression `if` e_1 `then` e_2 `else` e_3 evaluates to v (e_3 a well-formed expression).

- **Semantics:** $$\frac{e_1 \Downarrow \bot \quad e_3 \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}\ (\text{falseCond.})$$

  Following our context Γ, if a well-formed expression e_1 evaluates $\bot$ and another well-formed expression e_3 evaluates to v, then the expression `if` e_1 `then` e_2 `else` e_3 evaluates to v (e_2 a well-formed expression).

Take note that we must write two semantics rules for the `if` expression, one for when the condition evaluates to $\top$ and one for when it evaluates to $\bot$.

We continue with the formalization of the `function` expression in OCaml.

Definition 5.5: Formalizing Functions

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle$ ::= `fun` $\langle\text{var}\rangle$ `->` $\langle\text{expr}\rangle$

  If x is a valid variable name and e is a well-formed expression, then `fun` x `->` e is also a well-formed expression.

- **Typing-Rule:** $$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2}$$

  Given context Γ with a variable declaration of $(x : \tau_1)$ added, if there's a well-formed expression e of type τ_2 and, then the expression `fun` x `->` e is of type $\tau_1 \to \tau_2$.

- **Semantics:** $$\frac{}{\texttt{fun } x \texttt{ -> } e \Downarrow \lambda x.e}$$

  Under no premises, the expression `fun` x `->` e evaluates to the lambda function $\lambda x.e$.

Definition 5.6: Formalizing Application

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle$ ::= $\langle\text{expr}\rangle$ $\langle\text{expr}\rangle$

  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.

- **Typing-Rule:** $$\frac{\Gamma \vdash e_1 : \tau_1 \texttt{ -> } \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

  Given context Γ, if there's a well-formed expression e_1 of type τ_1 `->` τ_2 (Functions.5.5) and a well-formed expression e_2 of type τ_1, then the expression e_1 e_2 is of type τ_2.

- **Semantics:** $$\frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v \quad [v/x]e \Downarrow v'}{e_1\ e_2 \Downarrow v'}$$

  Following our context Γ, if a well-formed expression e_1 evaluates to a lambda function $\lambda x.e$, another well-formed expression e_2 evaluates to v, and the substitution of v for x in e evaluates to v', then the expression e_1 e_2 evaluates to v'.

Onto tuples and matching:

Definition 5.7: Formalizing Tuples

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle ::= \texttt{(}\langle\text{expr}\rangle\texttt{,}\ldots\texttt{,}\langle\text{expr}\rangle\texttt{)}$

  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.

- **Typing-Rule:** $$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_1, e_2, \ldots, e_n) : \tau_1 * \tau_2 * \cdots * \tau_n}$$

  Given context Γ, if there are well-formed expressions e_1 of type τ_1, e_2 of type τ_2, and e_n of type τ_n, then the expression $(e_1, e_2, \ldots, e_n)$ is of type $\tau_1 * \tau_2 * \cdots * \tau_n$.

- **Semantics:** $$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad \ldots \quad e_n \Downarrow v_n}{(e_1, e_2, \ldots, e_n) \Downarrow (v_1, v_2, \ldots, v_n)}$$

  Following our context Γ, if well-formed expressions e_1 evaluates to v_1, e_2 evaluates to v_2, and e_n evaluates to v_n, then the expression $(e_1, e_2, \ldots, e_n)$ evaluates to $(v_1, v_2, \ldots, v_n)$.

Definition 5.8: Formalizing Lists

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle ::= \texttt{[]} \mid \langle\text{expr}\rangle \texttt{ :: } \langle\text{expr}\rangle$

  The empty list `[]` is a well-formed expression. If e_1 is a well-formed expression and e_2 is a well-formed list, then e_1 `::` e_2 is also a well-formed expression.

- **Typing-Rule:** $$\frac{}{\Gamma \vdash [] : \tau\ \texttt{list}}\ (\text{nil}) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau\ \texttt{list}}{\Gamma \vdash e_1\ \texttt{::}\ e_2 : \tau\ \texttt{list}}\ (\text{cons})$$

  Given context Γ, the empty list `[]` has type τ `list` for any type τ (nil). If e_1 is of type τ and e_2 is of type τ `list`, then the expression e_1 `::` e_2 has type τ `list` (cons).

- **Semantics:** $$\frac{}{\texttt{[]} \Downarrow \varnothing}\ (\text{nilEval}) \qquad \frac{e_2 \Downarrow [v_2, \ldots, v_k] \quad e_1 \Downarrow v_1}{e_1\ \texttt{::}\ e_2 \Downarrow [v_1, v_2, \ldots, v_k]}\ (\text{consEval})$$

  The empty list `[]` evaluates to the empty list as a value (nilEval). If e_2 evaluates to the list $[v_2, \ldots, v_k]$ and e_1 evaluates to v_1, then e_1 `::` e_2 evaluates to the list $[v_1, v_2, \ldots, v_k]$ (consEval).

Matching in a general sense is complex (deep matching), we can simplify with weak matching:

Definition 5.9: Weak Matching

Weak matching is a form of pattern matching that is for specific cases.

Additionally, we introduce the concept of *side conditions* before we jump into weak matching on lists.

Definition 5.10: Side Conditions in Formal Semantics

A **side condition** is an additional constraint that must be satisfied before applying a rule. Side conditions are used to prevent undefined behavior and ensure correctness in evaluation.

Example 1: Integer Division Rule

- Consider the evaluation rule for integer division:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_2 \neq 0}{e_1 \div e_2 \Downarrow v_1 \div v_2}$$

- The side condition $v_2 \neq 0$ ensures that:
  - The denominator v_2 is not zero before performing division.
  - If $v_2 = 0$, the rule cannot be applied to avoid division by zero.
- Without this side condition, the expression could cause an error or undefined behavior.

Example 2: Exponentiation with Non-Negative Exponents

- Consider the evaluation rule for exponentiation:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_2 \geq 0}{e_1^{e_2} \Downarrow v_1^{v_2}}$$

- The side condition $v_2 \geq 0$ ensures that:
  - The exponent v_2 is non-negative before applying the exponentiation operation.
  - If $v_2 < 0$, the rule cannot be applied to avoid undefined results in integer arithmetic.
- Without this side condition, expressions like 2^{-3} would be invalid in integer arithmetic.

Side conditions help enforce correctness by restricting operations to only valid inputs.

Definition 5.11: Weak Matching on Lists

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle$ ::= `match` $\langle\text{expr}\rangle$ `with`
  `| [] ->` $\langle\text{expr}\rangle$
  `|` $\langle\text{var}\rangle$ `::` $\langle\text{var}\rangle$ `->` $\langle\text{expr}\rangle$

  If e, e_1, e_2 are well-formed expressions and x, y are valid variable names, then `match` e `with | [] ->` e_1`|` x `::` y `->` e_2 is a well-formed expression.

- **Typing Rule:**

$$\frac{\Gamma \vdash e : \tau' \texttt{ list} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau', y : \tau' \texttt{ list} \vdash e_2 : \tau}{\Gamma \vdash \texttt{match } e \texttt{ with | [] -> } e_1 \texttt{| } x \texttt{ :: } y \texttt{ -> } e_2 : \tau}$$

  If e is of type τ' `list` in the context Γ, and e_1 is of type τ in the context Γ, and e_2 is of type τ in the context Γ with $(x : \tau')$ and $(y : \tau'$ `list`$)$ added, then the entire match expression is of type τ.

- **Semantics:**

$$\frac{e \Downarrow \varnothing \quad e_1 \Downarrow v}{\texttt{match } e \texttt{ with [] -> } e_1 \texttt{| } x \texttt{:: } y \texttt{ -> } e_2 \Downarrow v}\ (\text{nil})$$

  If e evaluates to the empty list $\varnothing$ and e_1 evaluates to v, then the entire match expression evaluates to v.

$$\frac{e \Downarrow h \texttt{ :: } t \quad e_2' = [t/y][h/x]e_2 \quad e_2' \Downarrow v}{\texttt{match } e \texttt{ with | [] -> } e_1 \texttt{| } x \texttt{ :: } y \texttt{ -> } e_2 \Downarrow v}\ (\text{cons})$$

  If e evaluates to a nonempty list $h :: t$ with first element h and remainder t, and the expression e_2 with h substituted for x and t substituted for y evaluates to v, then the entire match expression evaluates to v.

1.5.2 All Ocaml Formalizations

Below is the full list from which we will reference throughout the text.

Full Specifications: For a full list of all the formalized expressions we'll be using. This list also includes the next topic we'll discuss **Derivations**:
https://nmmull.github.io/PL-at-BU/320Caml/notes.html

1.5.3 Derivations

Derivations allow us to unpack the formal expressions to prove their validity.

Definition 5.12: Tree Derivations

Tree derivations are a structured way of representing step-by-step reasoning in formal systems. They are often used in type systems, operational semantics, and logic proofs to show how conclusions follow from premises.

Each derivation is represented as a **tree**, where:

- **Leaves** represent axioms or base cases.
- **Internal nodes** apply inference rules to derive new conclusions.
- **The root** represents the final conclusion of the derivation, i.e., the starting point.

Example 5.1: Typing Derivation

Say we wanted to prove the typing derivation: `let y = 2 in y + y` : `int`

$$\dfrac{\dfrac{}{\{\} \vdash 2 : \texttt{int}}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{\{y : \texttt{int}\} \vdash y : \texttt{int}}\,(\text{var}) \qquad \dfrac{}{\{y : \texttt{int}\} \vdash y : \texttt{int}}\,(\text{var})}{\{y : \texttt{int}\} \vdash y + y : \texttt{int}}\,(\text{intAdd})}{\{\} \vdash \texttt{let y = 2 in y + y} : \texttt{int}}\,(\text{let})$$

Here the bottom of the tree is the final conclusion. We now unpack the highest level wrapper expression. The first expression we encounter is the `let` expression. The syntax of which are (e_3 ::= `let` x `=` e_1 `in` e_2). Hence we split into two branches to examine e_1 and e_2. The left branch examines the integer literal. The right branch looks at `(y + y)`. Since `(y + y)` is an addition operation, we must unravel once more into two branches examining both sides of the expression. The left and right branch examines the variable as an integer. Tunnelling from the leaf axioms to the root conclusion justifies the typing derivation as valid.

■

Example 5.2: Semantic Derivations

Continuing the same example, now with the semantics derivation: `let y = 2 in y + y` $\Downarrow 4$

$$\dfrac{\dfrac{}{2 \Downarrow 2}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{y \Downarrow 2}\,(\text{var}) \qquad \dfrac{}{y \Downarrow 2}\,(\text{var})}{y + y \Downarrow 4}\,(\text{intAdd})}{\texttt{let y = 2 in y + y} \Downarrow 4}\,(\text{let})$$

■

— 2 —

Algebraic Data Types

2.1 Recursive Algebraic Data Types

We've been working with primitive data types like integers, floats, and strings. But what if we want to create our own data types?

Definition 1.1: Algebraic Data Types (ADT)

Algebraic Data Types (`ADT`) are a way to define new data types by combining existing types using **sum** (variant) and **product** (tuple/record) constructions.

Example 1.1: Primitive Types vs. ADTs

In `OCaml`, some data types are primitive, while others are constructed using **ADTs**.

Listing 2.1: Primitive Types vs. ADTs

```
(* Primitive types (Not ADTs) *)
let x : int = 42       (* int is a built-in type *)
let y : bool = true    (* bool is a built-in type *)

(* Algebraic Data Types (ADTs) *)

(* Sum type (Variant) *)
type shape =
  | Circle of float
  | Rectangle of float * float

(* Product type (Record) *)
type point = { x: float; y: float }
```

Here, `int` and `bool` are **primitive types**, meaning they exist independently. In contrast:

- The `shape` type is a **sum type (variant)**, an ADT where a value can be either a `Circle` or a `Rectangle`.
- The `point` type is a **product type (record)**, an ADT that groups multiple values together.

■

2.1.1 Recursive Types

ADTs can be recursive, meaning they can refer to themselves in their definition. This allows the creation of data structures with variable lengths.

Theorem 1.1: Recursive ADTs and Variable-Length Data

Consider the following recursive definition of a list in `OCaml`:

Listing 2.2: Recursive List Definition

```
type intlist =
    | Nil
    | Cons of int * intlist

let example = Cons (1, Cons (2, Cons (3, Nil)))
```

Consider the following illustration:

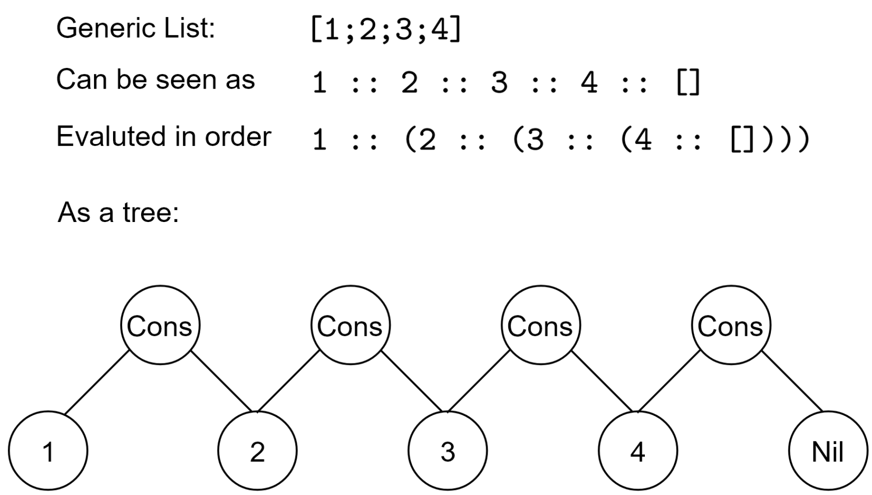

Figure 2.1: Ocaml list representations.

We can represent this of an integer list as either being,

- **Base case**: Empty (`Nil`)
- **Recursive case**: An Integer entry with reference to another entry (`int * intlist`), for which we label `Cons`.

In essence, a list is actually a type of tree structure, where each node has a value and a reference to the next node. And that in reality, we may represent lists, trees, and graphs as an Algebraic Data Type.

For instance, binary tree traversal can be represented as follows:

Example 1.2

We can define a binary tree as a recursive `ADT` in `OCaml`, where each node contains a value and references to left and right subtrees.

Listing 2.3: Binary Tree Definition and Traversals

```
(* Define a binary tree type *)
type 'a btree =
  | Empty
  | Node of 'a * 'a btree * 'a btree

(* Example tree *)
let example_tree =
  Node (1,
    Node (2, Node (4, Empty, Empty), Node (5, Empty, Empty)),
    Node (3, Node (6, Empty, Empty), Node (7, Empty, Empty))
  )

(* Preorder traversal: Root -> Left -> Right *)
let rec preorder t =
  match t with
  | Empty -> []
  | Node (v, left, right) -> [v] @ preorder left @ preorder right

(* Inorder traversal: Left -> Root -> Right *)
let rec inorder t =
  match t with
  | Empty -> []
  | Node (v, left, right) -> inorder left @ [v] @ inorder right

(* Postorder traversal: Left -> Right -> Root *)
let rec postorder t =
  match t with
  | Empty -> []
  | Node (v, left, right) -> postorder left @ postorder right @ [v]

(* Example usage *)
let _ =
  let pre = preorder example_tree in
  let inord = inorder example_tree in
  let post = postorder example_tree in
  (pre, inord, post)
(* Outputs:
([1; 2; 4; 5; 3; 6; 7], [4; 2; 5; 1; 6; 3; 7], [4; 5; 2; 6; 7; 3; 1])
*)
```

■

We may use recursive ADTs to model expressions. Take for instance a basic arithmetic expression:

$$3 + ((2 \cdot 4) - 14) \quad \Rightarrow$$

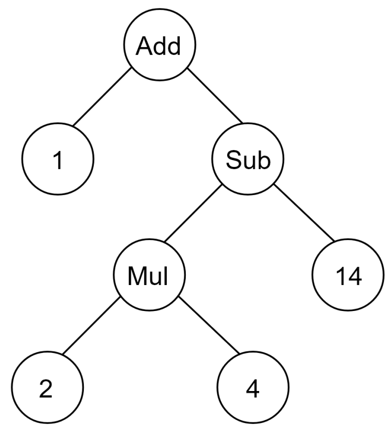

Figure 2.2: Arithmetic expression tree.

Example 1.3: Arithmetic Data Type

We define an arithmetic data type for Addition, Multiplication, and Subtraction of integers:

Listing 2.4: Arithmetic Expression Definition

```
type expr =
  | Num of int
  | Add of expr * expr
  | Mul of expr * expr
  | Sub of expr * expr

let _ = Add (Val 3, Sub (Mul (Val 2, Val 4), Val 14))
(* Renders:      3 + ((2 * 4) - 14)                *)
```

■

2.1.2 Parametric & Polymorphic Types

Now what if we want to create a generic list that can store any type of value? In this case, we must parametrize the type definition.

Definition 1.2: Parametric Types

Parametric types are types that can take one or more type parameters. They are useful for creating generic data structures that can store values of any type.

Example 1.4: Parametric List

We can define a parametric list in `OCaml` that can store values of any type:

Listing 2.5: Parametric List Definition

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list

let example_ints = Cons (1, Cons (2, Cons (3, Nil)))
let example_strings = Cons ("a", Cons ("b", Cons ("c", Nil)))
```

Where `'a` is a type parameter and `list` is the type constructor. ■

Making the type generic also makes it polymorphic, meaning it can store values of any type. This is useful for creating reusable data structures.

Definition 1.3: Polymorphic Types

Polymorphism is the ability of a function or data type to operate on values of different types. **Parametric polymorphism** refers to functions or data types that are generic and can operate on values of any type.

There is no overloading on types in `OCaml`:

Definition 1.4: Ad-Hoc Polymorphism

Ad-hoc polymorphism refers to a type of polymorphism where a function can be defined multiple times with different type signatures, allowing it to operate on different types using *distinct implementations*.

Some languages, such as C++ (via function overloading) and Haskell (via type classes), support ad-hoc polymorphism. However, **OCaml does not support ad-hoc polymorphism** because function overloading is not allowed—redefining a function replaces the previous definition.

Listing 2.6: No Overloading in OCaml

```
(* The following isn't possible *)
let add (a : int) (b : int) : int = a + b
let add (a : string) (b : string) : string = a ^ b (* Overwrite *)
let add (a : 'a list) (b : 'a list) : 'a list = a @ b (* Overwrites *)
```

Tip: Tony Hoare calls his invention of the null pointer a “billion-dollar mistake” OCaml doesn’t have null pointers

---

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
– Tony Hoare, inventor of null pointers

— 3 —

Higher-Order Programming

3.1 Function Order

In Higher-Order Programming, classes of values are established.

Definition 1.1: First-Class Values

A **first-class value** in a programming language is an entity that can be:

- **Assigned to variables**
- **Passed as an argument** to a function
- **Returned from a function**
- **Stored in data structures**

In `OCaml`, **functions are first-class values**, meaning they can be used like any other value. This allows for:

- Defining functions as values using `let`
- Passing functions as arguments to other functions
- Returning functions from other functions (closures)

However, **types are not first-class in OCaml**, meaning they cannot be manipulated as runtime values (e.g., dynamically created or passed as arguments).

Passing functions to other functions is where the idea of **higher-order functions** comes into play.

Definition 1.2: Higher-Order Functions

A **higher-order function** is a function that:

- **Takes one or more functions as arguments**
- **Returns a function as a result**

We've discussed such functions before in Subsection (1.3.1). E.g., (`fun f x -> f x`) is (`fun f -> fun x -> f x`) where the first function returns and takes a function as an argument. Recall (`f x`) is a function application, where (`f`) is a function value.

Definition 1.3: Order of Functions

The concept of "higher-order" extends beyond first-order functions, which take and return only values. A function's **order** is determined by how many levels of function application it involves:

- **1st order**: `int`
- **2nd order**: `int -> int`
- **3rd order**: `(int -> int) -> int`
- **4th order**: `((int -> int) -> int) -> int`

In theory, this hierarchy can extend infinitely, but in practice, functions rarely exceed **third or fourth order**.

Tip: [What Does "Higher-Order" Mean?] *"Like things and functions are different, so are functions whose arguments are functions* ***radically different*** *from functions whose arguments* ***must be things****. I call the latter functions of first order, the former functions of second order."*

– Gottlob Frege

3.1.1 The Abstraction Principle: Maps, Filters, Folds

The **Abstraction Principle** is a fundamental concept in computer science that states:

Definition 1.4: Abstraction Principle

The **Abstraction Principle** states that programs should be structured by separating **core functionality** from specific details.

This principle is applied by:

- **Abstracting core functionality** to improve reusability and modularity.
- Using **higher-order functions** to **parametrize** behavior based on specific problem requirements.
- Understanding the **algebra of programming**, which helps reason about program structure and transformations.

Following this principle results in more **flexible, maintainable, and reusable** programs.

We'll discuss three common patterns we see a lot in programmings **Maps**, **Filters**, and **Folds**.

Definition 1.5: Map Function

Given a function f and a list $[x_1, x_2, \ldots, x_n]$, the **map** function produces:

$$\texttt{map}\ f[x_1, x_2, \ldots, x_n] = [f(x_1), f(x_2), \ldots, f(x_n)]$$

I.e., it applies f to each element of the list, returning a new list with the results.

Listing 3.1: Ocaml Implementation of Map

```
let rec map f lst =
  match lst with
  | [] -> []
  | x :: xs -> f x :: map f xs

(* Example usage *)
let doubled = map (fun x -> x * 2) [1; 2; 3]  (* Returns [2; 4; 6] *)
```

The **map** function abstracts over how we apply a function to each element of a list, allowing us to reuse the same logic for different functions.

Definition 1.6: Filter Function

Given a predicate function p and a list $[x_1, x_2, \ldots, x_n]$, the **filter** function produces:

$$\texttt{filter}\ p\ [x_1, x_2, \ldots, x_n] = [x_i \mid p(x_i)\ \texttt{is true}]$$

I.e., it returns a new list containing only elements for which p evaluates to `true`.

Listing 3.2: Ocaml Implementation of Filter

```
let rec filter p lst =
  match lst with
  | [] -> []
  | x :: xs ->  (if p x then [x] else []) @ filter p xs

(* Example usage *)
let evens = filter (fun x -> x mod 2 = 0) [1; 2; 3; 4; 5]
(* Returns [2; 4] *)
```

The **filter** function abstracts over how we select elements from a list, allowing us to express selection logic concisely.

Definition 1.7: Fold Functions

Given a binary function f, an initial accumulator value a_0, and a list $[x_1, x_2, \ldots, x_n]$, the **fold** functions `fold_left` and `fold_right` reduce the list to a single value by combining elements recursively.

The **fold left** function applies f from left to right (recursive statement on the right):

$$\texttt{fold\_left}\ (-)\ a_0\ [x_1, x_2, \ldots, x_n] = (((a_0 - x_1) - x_2) \cdots - x_n)$$

Listing 3.3: Ocaml Implementation of Fold_Left

```
let rec fold_left f acc lst =
  match lst with
  | [] -> acc
  | x :: xs -> fold_left f (f acc x) xs

(* Example usage *)
let subtract = fold_left (-) 10 [1; 2; 3]
(* Returns 4, since ((10 - 1) - 2) - 3 = 4 *)
```

The **fold right** function applies f from right to left (recursive statement on the left):

$$\texttt{fold\_right}\ (-)\ a_0\ [x_1, x_2, \ldots, x_n] = (x_1 - (x_2 - (\cdots - (x_n - a_0))))$$

Listing 3.4: Ocaml Implementation of Fold_Right

```
let rec fold_right f lst acc =
  match lst with
  | [] -> acc
  | x :: xs -> f x (fold_right f xs acc)

(* Example usage *)
let subtract = fold_right (-) [1; 2; 3] 10
(* Returns -8, since 1 - (2 - (3 - 10)) = -8 *)
```

Tip: Think in opposition: fold left (recursive statement on the right), fold right (recursive statement on the left).

To illustrate the difference between `fold_left` and `fold_right`, consider the following illustration:

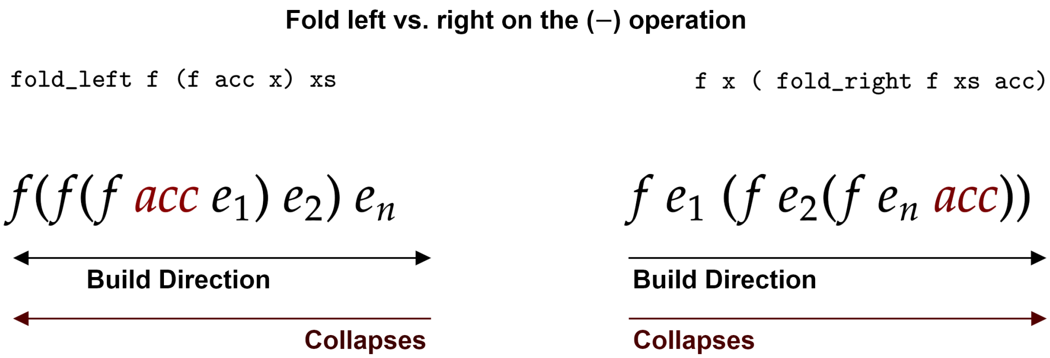

Figure 3.1: Fold Left vs. Fold Right

In both, we still build from the base case/last element to the front of the list. So above, we are always building from right to left. The difference:

- **Fold Left**: We are essentially wrapping the accumulator form the start to the end of the list. Meaning, The list folds from right to left starting with the last element.
- **Fold Right**: Here we make recursive calls starting with the first element and appending the accumulator to the end of the list. Meaning, we fold from left to right starting with the first element.

3.2 Handling Errors & Testing: Results, Bind, & Monads

3.2.1 Monads & Binds

Definition 2.1: Monads

A **monad** extends the functionality of a type by wrapping it in a monadic context. For example, we could extend the type `int` to include a `Null` type by wrapping it in a a custom type, `type myNumber = Null | Int of int`. Monads consists of:

- A **type constructor** M that wraps values.
- A **unit** (or `return`) function that lifts a value into the monadic context:

$$\eta : A \to M(A).$$

Where η is our unit function taking A and wrapping it in the monadic context $M(A)$.

- A **bind** function that ensures the monadic structure is maintained between operations:

$$\mu : M(A) \times (A \to M(B)) \to M(B),$$

Where μ is our bind function. Recall, the $(\times)$ is the cartesian product of the two types, i.e., a tuple of the two objects.

A monad satisfies the **monad laws**:

1. **Left Identity:** $(\eta(x)\ \texttt{bind}\ f) = f(x)$. Binding a monadic value to a function is equivalent to applying the function to the value.
2. **Right Identity:** $(m\ \texttt{bind}\ \eta) = m$. Binding a monadic value to the unit function is equivalent to the original value.
3. **Associativity:** $(m\ \texttt{bind}\ f)\ \texttt{bind}\ g = m\ \texttt{bind}\ (\lambda x.f(x)\ \texttt{bind}\ g)$. The order of binding functions does not matter.

In an OCaml context, `options` are an example of a monad.

Listing 3.5: Option Monad in OCaml

```
type 'a option =
| Some of 'a
| None
```

For the Option monad, `Some` serves as the unit function, as it takes a value of type `'a` and lifts it into the monadic context `'a option`. Both `Some` and `None` are constructors for the `option` type.

Definition 2.2: Bind with `let*` in OCaml

The `let*` operator corresponds to the monad's bind function. The bind operation could be thought of as "try to unwrap x and then do f". For example, in the `option` monad:

```
(* Define the bind function for option *)
let bind opt f =
  match opt with
  | Some x -> f x
  | None -> None
(* Define the let* operator *)
let ( let* ) = bind
```

Though OCaml has saved us the trouble of defining the bind function, via the `.bind` function in ocaml.

```
let ( let* ) = Option.bind
```

Using `let*` in Monadic Expressions: Once `let*` is defined, it allows chaining monadic computations naturally. Consider an example using the `option` monad:

```
(* Using let* to chain option operations *)
let foo =
    let ( let* ) = Option.bind in
    let* x = Some 4 in
    let* y = Some 3 in
    let* z = Some 2 in
    Some (x + y + z);

(* foo evaluates to Some 9 *)
```

This could be seen as the below nested match expressions:

```
match Some 4 with
| None -> None
| Some x -> (
    match Some 3 with
    | None -> None
    | Some y -> (
        match Some 2 with
        | None -> None
        | Some z -> Some (x + y + z)
        )
    )
```

This is for curiosity sake, and conceptually would be simpler to think of bind as a means of unwrapping the monad to preform some operation before rewrapping it.

Definition 2.3: Result Type in OCaml

The `result` type is another example of a monad in OCaml that represents computations which may **succeed or fail**. Unlike the `option` type which only indicates presence or absence, `result` provides information about why a computation failed.

Listing 3.6: Result Type in OCaml

```
type ('a, 'e) result =
| Ok of 'a      (* Success case with value of type 'a *)
| Error of 'e   (* Error case with error of type 'e *)
```

For the Result monad:

- `Ok` serves as the unit function, lifting a value into the success context
- The bind operation sequences computations while handling errors

Using Result with Bind:

```
(* Define the bind operator for result *)
let ( let* ) = Result.bind

(* Example chaining result operations *)
let divide x y =
  if y = 0 then Error "Division by zero"
  else Ok (x / y)

let computation x y z =
  let* result1 = divide x y in
  let* result2 = divide result1 z in
  Ok (result2 * 2)

(* Success case: computation 10 2 1 = Ok 10 *)
(* Error case:   computation 10 0 1 = Error "Division by zero" *)
```

In the example above, if any Error occurs it short-circuits the computation and returns the error immediately. Recall the nested match in the `let*` Definition (2.2), this is conceptually similar to that.

3.2.2 Testing & Ounit2

This section is about testing as a means of ensuring the correctness of our code down the development pipeline.

Definition 2.4: Types of Testing

In software development, testing can be categorized into several hierarchical levels, with the three primary types being:

1. **Unit Testing:** Tests individual components (functions, modules) in isolation.
    - Focuses on verifying that each unit of code works as expected
    - Typically automated and run frequently during development
2. **Integration Testing:** Tests interactions between components.
    - Verifies that different units work together correctly
    - Components may be nested or distributed across the simulated workflow
3. **End-to-End (E2E) Testing:** Tests the application from start to finish.
    - Simulates real user scenarios and workflows
    - Verifies the system works in real-world conditions with actual data
    - Tests the entire application stack including UI, API, database connections, etc.

In software development **testing frameworks** are used to help speed up the process of writing and running tests. These are libraries that provide tools for writing, organizing, and running tests.

Tip: While academic settings often focus on the theoretical aspects of testing, industry practices are typically more nuanced and pragmatic. In many professional software development environments:

- **Test-Driven Development (TDD)** has gained significant traction, where developers write tests before implementing functionality. This approach often leads to more testable and modular code, but requires discipline to maintain.
- **Continuous Integration (CI)** systems run tests automatically when code changes are committed, catching regressions early in the development cycle. Companies may run thousands of tests multiple times daily.
- The **testing pyramid** concept is widely followed, with many unit tests forming the base, fewer integration tests in the middle, and even fewer E2E tests at the top—balancing thoroughness with execution speed.

We choose **OUnit2** for testing, which should have been installed in Sub-section (1.2.3). Though not the most featured testing framework, it is simple and easy to use.

Definition 2.5: OUnit2 in OCaml

OUnit2 is a unit testing framework for OCaml that allows developers to write and run tests to verify code correctness. To use OUnit2, add it as a dependency in your dune project file:

```
(test
 (name test_program)
 (libraries ounit2))
```

Key OUnit2 Functions:

- `(>::)` - Creates a labelled test
- `(>:::)` - Creates a labelled test suite
- `assert_equal` - Compares two values in a unit test
- `assert_raises` - Checks that an expression raises the expected exception
- `run_test_tt_main` - Runs a test suite

Example OUnit2 Test:

```
open OUnit2

(* Function to test *)
let add x y = x + y

(* Test cases *)
let tests =
    "test suite for addition" >:::
    [
    "adding two positive numbers" >:: (fun _ ->
        assert_equal 5 (add 2 3));

    "adding zero" >:: (fun _ ->
        assert_equal 7 (add 7 0));

    "testing exception" >:: (fun _ ->
        assert_raises (Failure "division by zero")
        (fun () -> 1 / 0))
    ]

(* Run the tests *)
let () = run_test_tt_main tests
```

3.3 Modules In Ocaml

This section details how OCaml deals with modular programming, including abstractions and interfaces.

Definition 3.1: Modular Programming

Modular programming is a software design approach that emphasizes separating a program's functionality into independent, interchangeable modules, which are composed of three key elements:

1. **Namespaces:** A way of separating code into logical units of functions, types, and values together while avoiding name conflicts.
2. **Abstraction/Encapsulation:** A way of abstracting away implementation details and organizing core functionality. This creates a clear boundary between the module's intent and its implementation for clarity.
3. **Code Reuse:** Well-designed modules can serve as reusable components across multiple projects, reducing duplication.

Definition 3.2: The (`module`) & (`struct`) Keyword in OCaml

The **module** keyword in OCaml is used to define a collection of related code elements (types, values, functions) that are grouped together into a single namespace.

Listing 3.7: **Basic Module Syntax:**

```
module ModuleName = struct
(* Types, values, and functions *)
type t = int * int
let create x y = (x, y)
let add (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
end
```

Where the `struct` keyword defines the collection of definitions under the module. Once defined, module elements are accessed using the dot notation:

```
let point = ModuleName.create 10 20
let sum = ModuleName.add point point
```

Multiple modules may be defined in a single file, and be used in and between other files.

Definition 3.3: Module Access: (open) and Local Opens

Caml provides multiple ways to access module contents:

Qualified access: uses dot notation: `ModuleName.function_name`

Global open: brings all module contents into the current scope:

```
(* All List functions now available without qualification *)
open List
let x = map (fun x -> x * 2) [1; 2; 3]  (* No need for List.map *)
```

Local open: provides temporary access within a limited scope:

```
(* Using Module.(expr) syntax *)
let result = List.(
map (fun x -> x * 2) [1; 2; 3]
(* List is open only in this scope *)
)

(* Outside the parentheses, module is not opened *)
let standard = List.length [1; 2; 3]  (* Need qualification again *)
```

Definition 3.4: Module Signatures: (sig) & (module type)

Signatures are interfaces to modules. They are created in `.mli` files:

```
module type POINT = sig
  (* Abstract type - implementation hidden *)
  type t
  val private create : int -> int -> t
  val add : t -> t -> t
end
```

The `val` keyword explains rather than defines like `let`. Signatures are then applied to modules to ensure they conform to the interface:

```
module Point : POINT = struct
  type t = int * int
  let create x y = (x, y)
  let add (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

  (* This function is private as its not in the signature *)
  let sub x y = x - y
end
```

Definition 3.5: Module Helper Pattern for Data Structures

A common pattern in OCaml is to create helper functions inside modules to simplify working with complex data structures. This approach makes code more concise and readable by providing shorthand notations for constructors.

For example, with a binary tree:

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree

module TreeExample = struct
  let l = Leaf
  let n l r = Node ((), l, r)
end

(* Using local open for concise tree construction *)
let example = TreeExample.(n (n (n l l) l) (n l l))
```

Instead of writing the full constructor names repeatedly, the module provides shorthand aliases (`l` for `Leaf` and `n` for creating `Node`s). Combined with local opens, this makes tree construction much more readable than the equivalent:

```
let example = Node ((), Node ((), Node ((), Leaf, Leaf), Leaf), Node
    ((), Leaf, Leaf))
```

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].