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.

<center>Prerequisite Definitions</center>

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.

<center>5</center>

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.

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 $\langle e_2 \rangle$, $\langle e_3 \rangle$, ..., 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

- $\langle\text{month}\rangle ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12$

- $\langle\text{OS}\rangle ::= \langle\text{Linux}\rangle | \langle\text{Windows}\rangle | \langle\text{MacOS}\rangle$

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.   [4]

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:

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

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 (FP)**. 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

```
1    # Homebrew (macOS)
2    brew install opam
3
4    # MacPort (macOS)
5    port install opam
6
7    # Ubuntu
8    apt install opam
9
10   # Debian
11   apt-get install opam
12
13   # Arch Linux
14   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

```
1    # Initialize OPAM
2    opam init
3
4    # Configure your shell environment
5    eval $(opam env)
6
7    # Verify OPAM is ready to use
8    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

```
1    # Step 1: Create a new switch named "my_switch" with OCaml version 5.2.1
2    opam switch create my_switch 5.2.1
3
4    # Step 2: Activate the newly created switch
5    opam switch my_switch
6
7    # Step 3: Update your terminal environment to reflect the switch
8    eval $(opam env)
9
10   # Step 4: Verify the switch is active
11   opam switch
12
13   # (Or / Optionally) Check the OCaml version
14   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

```
1  # Step 1: Update OPAM to fetch the latest package information
2  opam update
3
4  # Step 2: Install essential development tools
5  opam install dune utop ounit2 menhir ocaml-lsp-server
6
7  # Step 3: Install the custom library for this course
8  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

```
1    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

```
1    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

```
1    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

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

You should see the output:

```
1    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

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

7. Build and run the program:

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

The output should now be:

```
1    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

```
1    (library
2     (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

```
1    (executable
2     (public_name hello_dune)
3     (name main)
4     (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

```
1    (* Greets a person with their name. *)
2    let greet name = "Hello, " ^ name ^ "!"
3
4    (* Adds two integers. *)
5    let add x y = x + y
6
7    (* Multiplies two integers. *)
8    let multiply x y = x * y
9
10   (* Checks if x is divisible by y. *)
11   let is_divisible x y = x mod y = 0
```

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

2. Use the functions interactively in `utop`:

```
1    dune utop
```

Now we can access the functions:

Listing 1.18: Accessing Functions in Utop

```
1    # Hello_dune.greet "OCaml";;
2    - : string = "Hello, OCaml!"
3
4    # Hello_dune.add 3 5;;
5    - : int = 8
6
7    # Hello_dune.multiply 4 6;;
8    - : int = 24
9
10   # Hello_dune.is_divisible 10 2;;
11   - : 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

```
1    # open Hello_dune;;
2    # greet "Functional Programming";;
3    - : string = "Hello, Functional Programming!"
4
5    # add 7 2;;
6    - : int = 9
```

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

Listing 1.20: Using the Library in Main

```
1    let () =
2        let greeting = Hello_dune.greet "OCaml" in
3        let sum = Hello_dune.add 3 5 in
4        let product = Hello_dune.multiply 4 6 in
5        let divisible = Hello_dune.is_divisible 10 2 in
6        Printf.printf "%s\nSum: %d\nProduct: %d\nDivisible: %b\n"
7        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:

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

You should see the output:

```
1    Hello, OCaml!
2    Sum: 8
3    Product: 24
4    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

```ocaml
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

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

This will result in the following error:

Listing 1.24: Error Message

```
1    3 | let _ = x + y (* THIS IS NOT POSSIBLE *)
2            ^
3    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 (3.11). 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
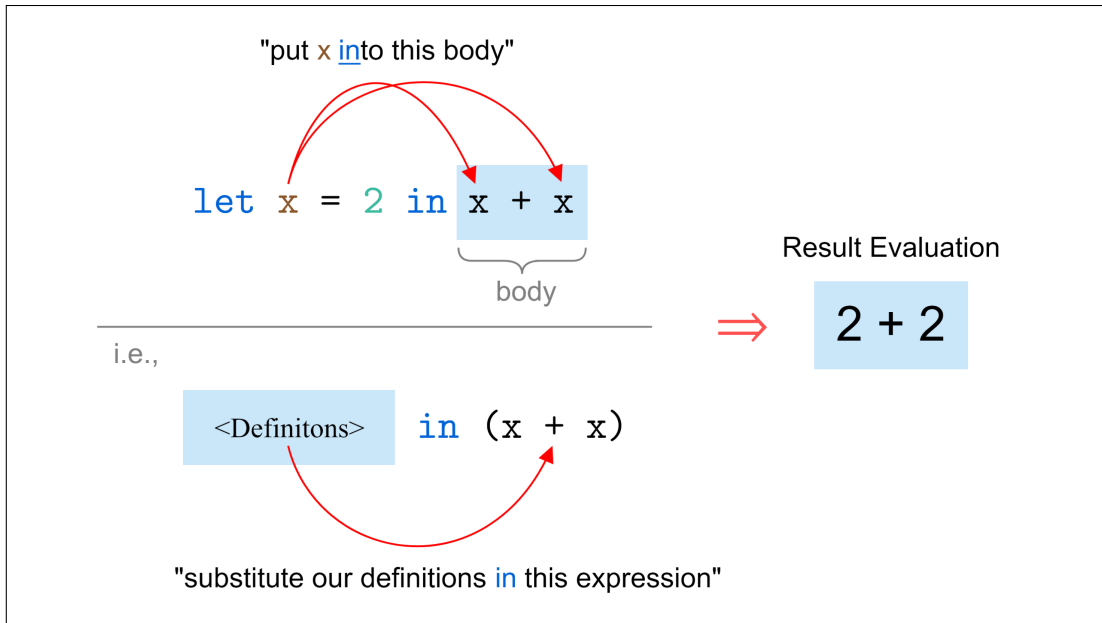
To disect the roles of **syntax**, **semantics**, and **types** in the expression `let x = 2 in x + x`:
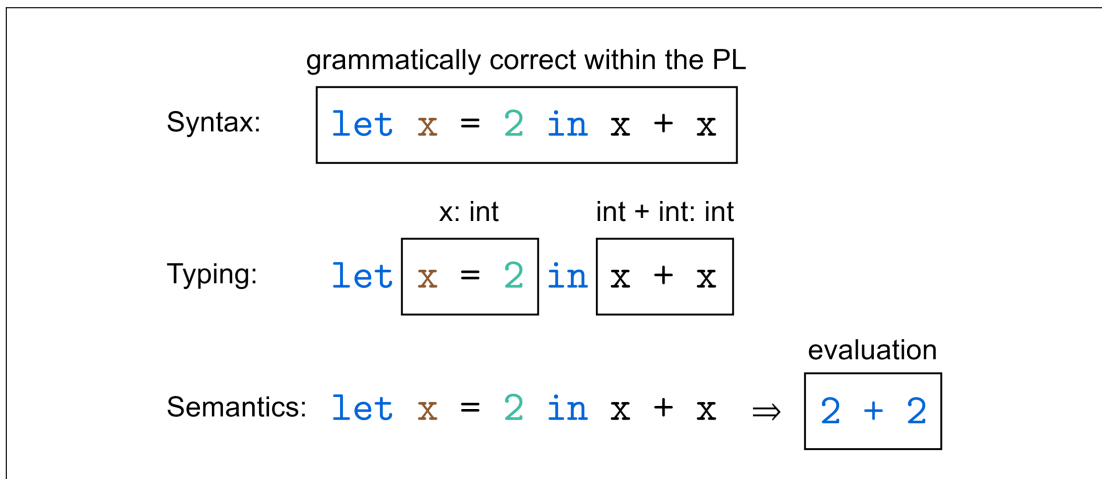

Figure 1.2: Syntax, Semantics, and Types in OCaml

- **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 three arguments, `x`, `y`, and `z` 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:
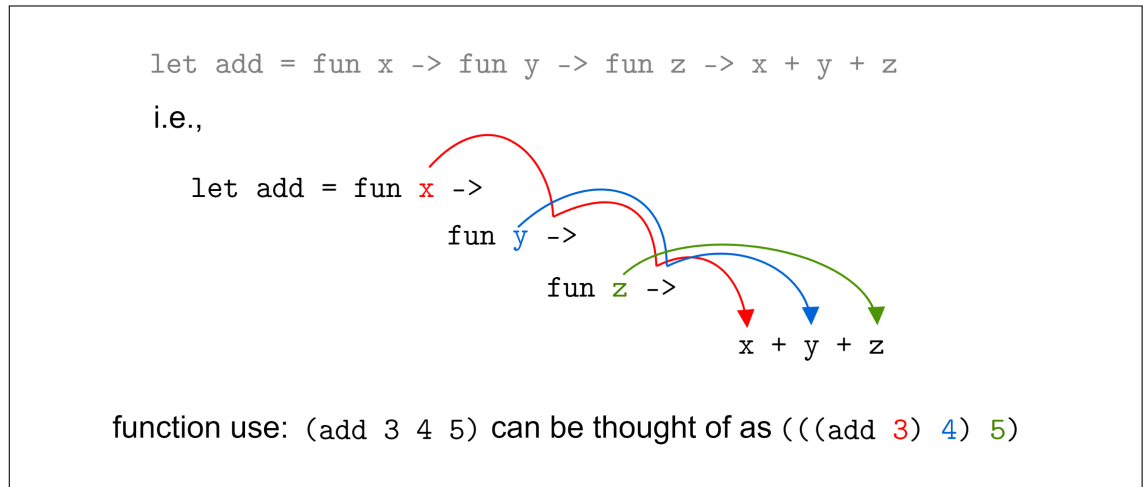
Figure 1.3: Anonymous Functions in OCaml

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.
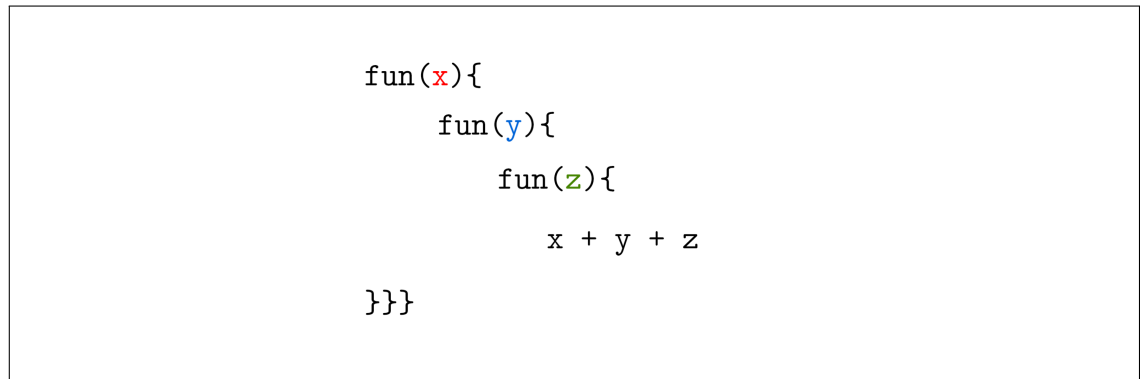
Figure 1.4: Where x is a local variable of the outer-most function within scope of the inner functions, and so on with y and z. This is the concept known as **closures**.

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:

```
1    (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:

$$(\lambda x.\lambda y.(x + y)) \; 3 \; 5 \rightarrow$$
$$(\lambda y.(3 + y)) \; 5 \rightarrow$$
$$(3 + 5)$$

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

```ocaml
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

```ocaml
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

```ocaml
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

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

Listing 1.71: Getting the Length of a String

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

Listing 1.72: Accessing a Character in a String

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

Listing 1.73: Slicing a String

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

Listing 1.74: Converting to Uppercase

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

Listing 1.75: String Equality

```ocaml
"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

```ocaml
(* 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

```ocaml
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

```ocaml
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

```ocaml
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
```

∎

1.3.11   Defining Operators: infix & prefix operators

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

Definition 3.31: 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 *)
```

1.3.12 Print Debugging

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

Listing 1.107: Print Debugging

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

Without the `stdlib` we can do the following:

Listing 1.108: Print Debugging Without Printf

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

Perhaps a quick print without care of formatting:

Listing 1.109: Dirty Print Debugging

```
1   let rec factorial n acc =
2       if n <= 1 then acc
3       else
4           let _ = print_int n in
5           let _ = print_string ", " in
6           let _ = print_int acc in
7           factorial (n - 1) (n * acc)
8   in factorial 5 1
```

Section Exercises:

Exercise 3.1: 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 3.2: Write an OCaml function that takes an integer x and evaluates to the first digit of x using only integer arithmetic operations.

Exercise 3.3: 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 3.4: Write an OCaml function implementation of the Fibonacci sequence using tail-end recursion.

1.4 Formalizing Ocaml Expressions

1.4.1 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 4.1: 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 4.2: 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 4.3: 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 4.4: Formalizing If-Expressions

Let Γ be the OCaml context, then:

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

  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 4.5: Formalizing Functions

Let Γ be the OCaml context, then:

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

  If x is a valid variable name and e is a well-formed expression, then $\texttt{fun} \ x \ \texttt{->} \ e$ 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 $\texttt{fun} \ x \ \texttt{->} \ e$ is of type $\tau_1 \to \tau_2$.

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

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

Definition 4.6: 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 $\tau_1 \ \texttt{->} \ \tau_2$ (Functions.4.5) 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 4.7: Formalizing Tuples

Let Γ be the OCaml context, then:

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

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

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

  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 4.8: 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:** $\dfrac{}{\Gamma \vdash [] : \tau \; \texttt{list}} \; (\text{nil}) \qquad \dfrac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \; \texttt{list}}{\Gamma \vdash e_1 \; \texttt{::} \; e_2 : \tau \; \texttt{list}} \; (\text{cons})$

  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:** $\dfrac{}{\texttt{[]} \Downarrow \varnothing} \; (\text{nilEval}) \qquad \dfrac{e_2 \Downarrow [v_2, ..., v_k] \quad e_1 \Downarrow v_1}{e_1 \; \texttt{::} \; e_2 \Downarrow [v_1, v_2, ..., v_k]} \; (\text{consEval})$

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

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

Definition 4.9: 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 4.10: 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 4.11: 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.

1.4.2 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

1.4.3 Derivations

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

Definition 4.12: 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 4.1: 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 ($e_3 ::= \texttt{let } x = e_1 \texttt{ in } e_2$). 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 4.2: Semantic Derivations

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

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

■

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

```ocaml
(* 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:

Generic List: `[1;2;3;4]`

Can be seen as   `1 :: 2 :: 3 :: 4 :: []`

Evaluted in order   `1 :: (2 :: (3 :: (4 :: [])))`

As a tree:
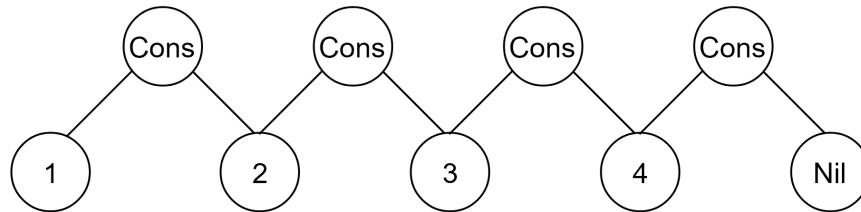
Figure 2.1: Ocaml list representations.

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
>
> ```ocaml
> (* 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:

$$3 + ((2 \cdot 4) - 14) \qquad \Longrightarrow$$
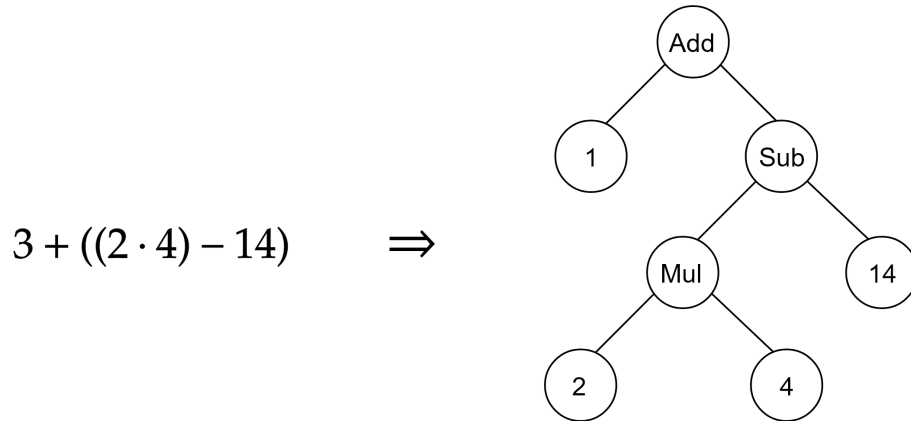
Figure 2.2: Arithmetic expression tree.

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

```ocaml
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

```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

```ocaml
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:

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

I.e., it returns a new list containing only elements for which p evaluates to $\texttt{true}$.

Listing 3.2: Ocaml Implementation of Filter

```ocaml
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

```ocaml
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

```ocaml
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:

Fold left vs. right on the (−) operation

`fold_left f (f acc x) xs`   `f x ( fold_right f xs acc)`

$$f(f(f\ acc\ e_1)\ e_2)\ e_n \qquad\qquad f\ e_1\ (f\ e_2(f\ e_n\ acc))$$

Build Direction   **Build Direction**

Collapses   **Collapses**
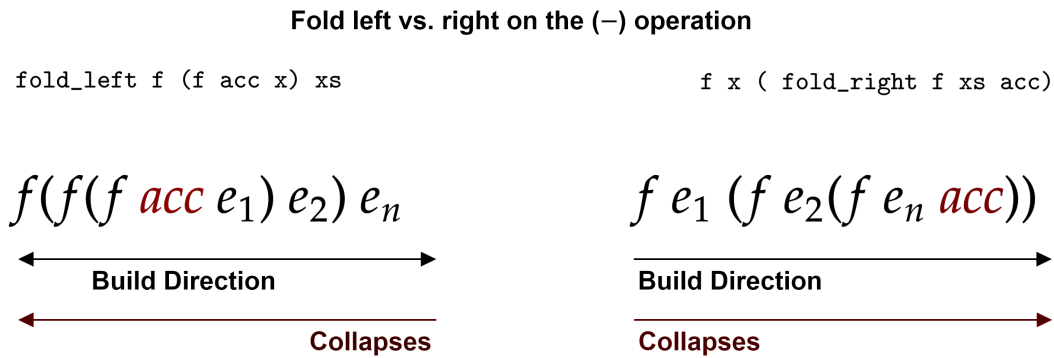
Figure 3.1: Fold Left vs. Fold Right

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:** $(\eta(x) \text{ bind } f) = f(x)$. Binding a monadic value to a function is equivalent to applying the function to the value.

2. **Right Identity:** $(m \text{ bind } \eta) = m$. Binding a monadic value to the unit function is equivalent to the original value.

3. **Associativity:** $(m \text{ bind } f) \text{ bind } g = m \text{ bind } (\lambda x.f(x) \text{ bind } g)$. 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:

```ocaml
(* 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.

```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:

```ocaml
(* 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:

```ocaml
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:

```ocaml
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:

```
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. For example, with a binary tree:

```ocaml
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))
```

This makes tree construction much more readable than the equivalent:

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

Definition 3.6: Modules and OCaml Projects

In OCaml, all `.ml` is implicitly wrapped in a module. Signatures are implicitly generated by the compiler without a corresponding `.mli` file. In such file, the signature body is defined.

Example 3.1: Module Example Project (Part 1)

Let's create a new project to demonstrate modules in OCaml. We'll create a simple module for basic shape math operations.

```
dune init project demo
cd demo
vim lib/dune
```

We'll start by letting the project know of our modules:

```
(library
  (name shapemath)          ; internal name
  (modules shape math)      ; modules that make up this library
)
```

∎

Example 3.2: Module Example Project (Part 2)

Next, we'll create the module files:

```
vim lib/shape.mli
```

Now we'll define the module interface:

```
(* shape.mli *)
type t
val square : float -> t
val area : t -> float
```

Note, the `module type ... sig` absence. The interface is defined directly in the `.mli` file.

Next, we'll define the implementation:

```
vim lib/shape.ml
```

```
(* shape.ml *)
type t = Square of float

let square side = Square side

let area = function
| Square s -> s *. s
```

Now the math ml and mli files:

```
(* math.mli *)
val add : int -> int -> int
val multiply : int -> int -> int
```

```
(* math.ml *)
let add x y = x + y

(*private function, as not defined in mli *)
let add_zero x = (+) 0 x

let multiply x y = add_zero (x * y)
```

∎

Definition 3.7: Private Functions in Ocaml

If the `.ml` file defines a function not specified in the `.mli` file, it is considered private.

Example 3.3: Module Example Project (Part 3)

Setup access for `main` and `test` :

Listing 3.8: **bin/dune** file

```
;; bin/dune
(executable
(name main)              ; internal name for the executable
(public_name demo)       ; how it appears installed or run via dune exec
(modules main)           ; the main file is main.ml
(libraries shapemath)    ; depends on our newly defined library
)
```

Listing 3.9: **bin/main.ml** file

```
open Shapemath

let () =
  (* Create a square with side length 5.0 *)
  let square = Shape.square 5.0 in

  (* Calculate and print the area of the square *)
  let area = Shape.area square in
  Printf.printf "Area of square with side 5.0: %f\n" area;

  (* Demonstrate the math module *)
  let sum = Math.add 10 20 in
  let product = Math.multiply 5 6 in
  Printf.printf "10 + 20 = %d\n" sum;
  Printf.printf "5 * 6 = %d\n" product;

  print_endline "Demo completed successfully!"
```

Now, we'll test the project:

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

Or even in utop:

```
utop # Shapemath.Math.multiply 5 6;;
- : int = 30

utop # Shapemath.Math.add_zero 4 6;;
Error: Unbound value Shapemath.Math.add_zero
```

We also see that our private function `add_zero` is not accessible outside the module. ∎

Example 3.4: Module Example Project (Part 4)

Finally, we'll add a test file to the project using OUnit2. If for some reason OUnit2 is not installed, run in the terminal:

```
opam install oUnit2
```

Now lets create the test file:

```
vim test/test_demo.ml
```

```ocaml
open OUnit2
open Shapemath

let test_square _ =
  let square = Shape.square 5.0 in
  assert_equal 25.0 (Shape.area square)

let test_add _ =
  assert_equal 30 (Math.add 10 20)

let test_multiply _ =
  assert_equal 30 (Math.multiply 5 6)

let suite =
  "suite">:::
  ["test_square">:: test_square;
   "test_add">:: test_add;
   "test_multiply">:: test_multiply]

let () =
  run_test_tt_main suite
```

Now ensure the test is added to the `dune` file:

```
(test
 (name test_demo)
 (libraries ounit2 shapemath))
```

Now run the test in the terminal:

```
dune runtest
```

```
...
Ran: 3 tests in: 0.10 seconds.
OK
```

As an exercise, try creating a test that will fail to see the error message.   ∎

Example 3.5: Module Example Project Summary

In this example, we created a simple OCaml project with two modules: `shape` and `math`. We defined the modules in `lib/shape.ml` and `lib/math.ml`, and their interfaces in `lib/shape.mli` and `lib/math.mli`. We then created a `main` executable in `bin/main.ml` that used the modules. Finally, we created a test file in `test/test_demo.ml` and ran the tests using `dune runtest`.

This leaves us with the following project structure after `dune clean`:

```
demo
├── bin
│   ├── dune
│   └── main.ml
├── demo.opam
├── dune-project
├── lib
│   ├── dune
│   ├── math.ml
│   ├── math.mli
│   ├── shape.ml
│   └── shape.mli
└── test
    ├── dune
    └── test_demo.ml
```

∎

The Interpretation Pipeline

4.1 Formal Grammars

4.1.1 Defining a Language

Now we introduce formal grammars as a way of building up our language. This is similar to English, where we have a grammar system that tells us how to build sentences. For example, we know the basic structure of a sentence is *subject-verb-object*.

We can write **linear** statements such as "John hit the ball", which has an underlying **hierarchical** structure that permits it:
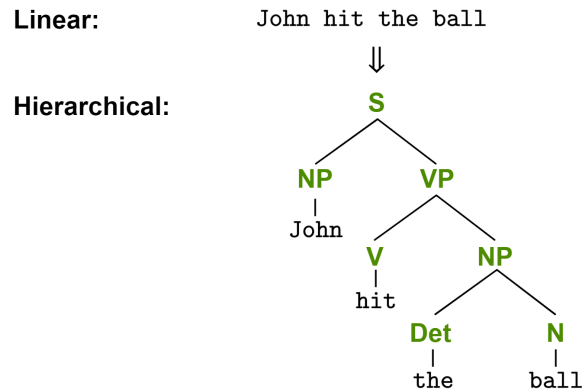
Figure 4.1: The sentence "John hit the ball" has an underlying hierarchical structure of a tree. Here, **S**: Sentence (the root of the tree), **NP**: Noun Phrase (a phrase centered around a noun), **VP**: Verb Phrase (a phrase centered around a verb), **V**: Verb (the action in the sentence), **Det**: Determiner (words like "the", "a", "an", which specify nouns), and **N**: Noun (person, place, thing, or idea).

Grammar vs. Semantics: Notice the english sentence

"Your air tied a toothbrush at school!"

is grammatically correct, but carries little to no meaning. In contrast, the sentence:

"Colorless the of allegator run am sleepily"

is perhaps an unsettling read, as it is not grammatically correct.

The same way we can represent english sentences in a tree structure, is the same way we can represent programs. First we define the difference between an **interpreter** and a **compiler**.

Definition 1.1: Interpreter

An **interpreter** is a program that directly executes instructions written in a programming language without requiring a machine code translation. The typical stages are:

1. **Lexical Analysis**: Reads a string of characters (program), converting it into tokens.

2. **Syntax Analysis**: Parses these tokens to build an abstract syntax tree (AST).

3. **Semantic Analysis**: Checks for semantic errors and annotates the AST.

4. **Intermediate Representation (IR) Generation**: Converts the AST into an intermediate representation (IR) to facilitate execution.

5. **Direct Execution**: Executes the IR or AST directly using an interpreter.

Interpreted languages are **evaluated at runtime** (e.g., Python, Ruby, JavaScript). Some interpreters use an **AST-based execution**, while others generate an **IR** (e.g., Python's bytecode for the CPython interpreter).

Definition 1.2: Compiler

A **compiler** is a program that translates code written in a high-level programming language into a lower-level language, typically machine code, to create an executable program. This involves several stages:

1. **Lexical Analysis**: Reads the source code and converts it into tokens.

2. **Syntax Analysis**: Parses these tokens to construct an abstract syntax tree (AST).

3. **Semantic Analysis**: Validates the AST against language rules and performs type checking.

4. **Intermediate Representation (IR) Generation**: Transforms the AST into a lower-level representation that is easier to optimize and translate.

5. **Optimization**: Enhances the IR to improve performance and efficiency.

6. **Code Generation**: Translates the optimized IR into machine code or another target language.

Compiled languages are **translated before runtime** (e.g., C, C++, Rust, OCaml). The **IR** plays a crucial role in optimizing the compilation process, as seen in LLVM or Java's bytecode execution in the JVM.

The following diagram illustrate the translation process of a compiler and an interpreter:

Interpreter:

char stream ──────▶
input ──────▶ │ INTERPRETER │ ──────▶ output

Compiler:

char stream ──────▶ │ COMPILER │

low-level
conversion

input ──────▶ │ TARGET │ ──────▶ output
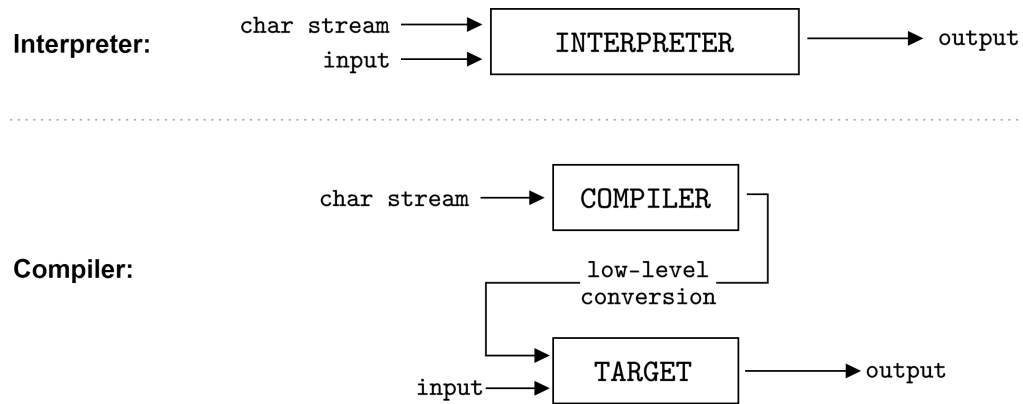
Figure 4.2: The high-level processes of a compiler and an interpreter.

The flow of a program through a compiler or interpreter is as follows:

char stream ──────▶ │ LEXICAL ANALYSIS │

──token stream──

│ SYNTACTIC ANALYSIS │

──parse tree──

│ SEMANTIC ANALYSIS │

──AST/IR──

input ──────▶ │ EVALUATION │ ──────▶ output

Figure 4.3: Stages of program processing: A character stream undergoes lexical, syntactic, and semantic analysis, transforming into an AST or IR before evaluation. Interpreters execute the AST/IR directly, while compilers translate it into machine code.

To formally layout our language, we use the **Backus-Naur Form (BNF)** notation. But before we do so, we gain some intuition by breaking down an english sentence from its **terminal symbols**, to its **non-terminal symbols**. Recall these terms from the pre-requisite section (0.2).

```
the cow jumped over the moon
<article> cow jumped over the moon
<article> <noun> jumped over the moon
<noun-phrase> jumped over the moon
<noun-phrase> jumped over <article> moon
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> <verb-phrase>
<sentence>
```
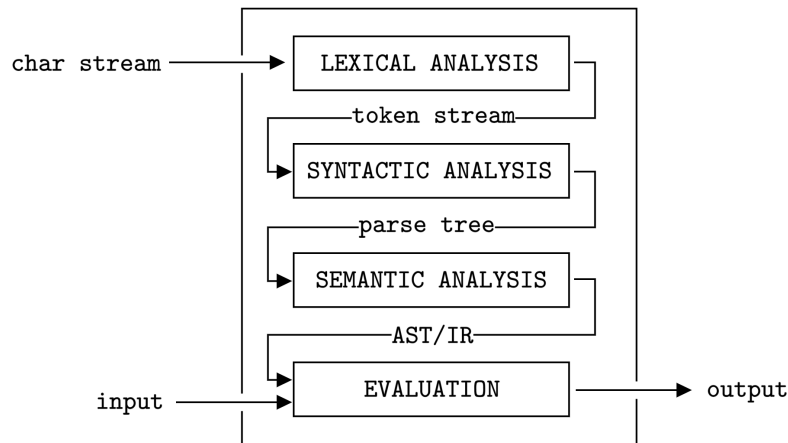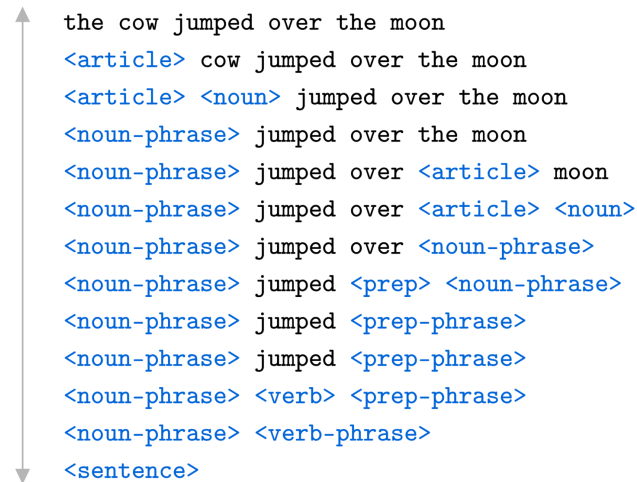
Figure 4.4: The sentence "The cow jumped over the moon." broken down into terminal and non-terminal symbols. This is a derivation showing how the sentence is built up from the start symbol of a `<sentence>`.

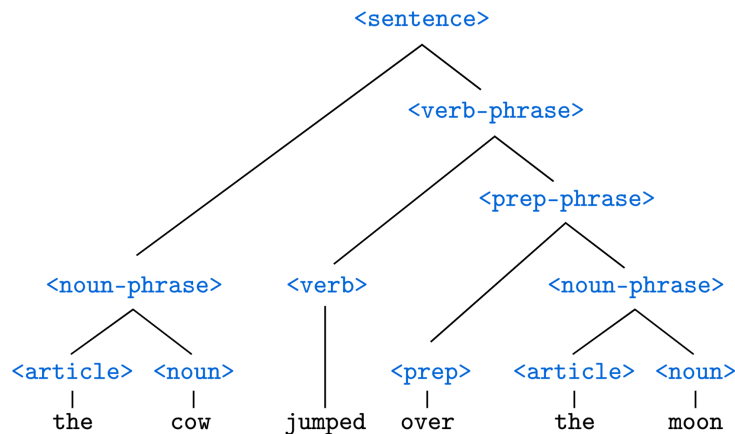From which the above can be represented in a tree structure:

Figure 4.5: The sentence "The cow jumped over the moon," represented as a **parse-tree**.

Now if we wanted to state these rules before-hand that a `<sentence>` is made up of a `<noun-phrase>` and a `<verb-phrase>` we can do so as such:

```
<sentence>      ::= <noun-phrase> <verb-phrase>
<verb-phrase>   ::= <verb> <prep-phrase> | <verb>
<prep-phrase>   ::= <prep> <noun-phrase>
<noun-phrase>   ::= <article> <noun>
<article> ::= the
<noun>     ::= cow | moon
<verb>     ::= jumped
<prep>     ::= over
```

Figure 4.6: The rules for the sentence "The cow jumped over the moon," via a thread of production rules (1.3). This example illustrates **Backus-Naur Form (BNF)** notation.

Definition 1.3: Backus-Naur Form (BNF)

Backus-Naur Form (BNF) is a formal notation used to define the context. It consists of production rules that specify how symbols in a language can be recursively composed. Each rule follows the form:

```
<non-terminal> ::= expression
```

where `<non-terminal>` represents a syntactic category, and `expression` consists of terminals, non-terminals, or alternative sequences.

Consider a more programmer-like example, assuming we read from left-to-right:

Grammar:
```
<expr> ::= <op1> <expr>
         | <expr> <op2> <expr>
         | <var>
<op1> ::= not
<op2> ::= and | or
<var> ::= x | y | z
```

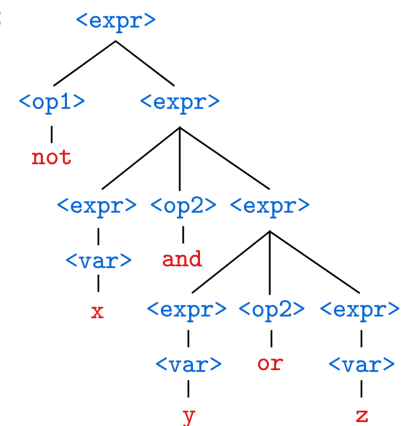**Sentence:**
```
not x and y or z
```

Parse-tree:

Figure 4.7: A simple language, consisting of `and` and `or` operations.

The process of taking non-terminal symbols and replacing them with terminal symbols is called a **leftmost derivation**.

Definition 1.4: Leftmost Derivation

A **leftmost derivation** is a sequence of sentential forms in which, at each step, the **leftmost** non-terminal symbol is replaced according to a production rule. This process continues until the entire string consists only of terminal symbols.

4.1.2 Ambiguity in Grammars

In natural language we have cases where what we say can be ambiguous. For example,

"The duck is ready for the dinner table."

Natural questions may arise:

- Was the duck ready to be eaten?

- Was the duck a servant preparing the table for dinner?

- Was the duck a wrestler, ready to body-slam a constituent?

For instance, let's clean up the previous sentence, avoiding any, and all ambiguity:

*"The duck is ready **to body slam** the dinner table."*

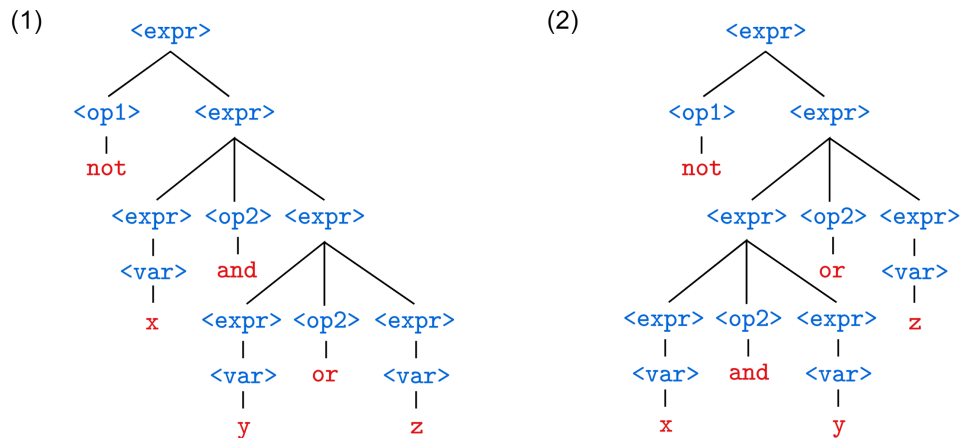Now Consider the previous example of `and or not` operations from Figure (4.7):

Figure 4.8: Two possible parse-tree derivations based off of Figure (4.7). (1) Shows our previous intention, while (2) shows a different interpretation, if parsing order was not specified.

In Figure (4.8), (1) shows, `not x and (y or z)` while (2) shows, `not (x and y) or z`. This ambiguity can cause issues; for example, $x = y = False$, $z = True$, yields different results for (1) and (2).

Definition 1.5: Ambiguity in BNF Grammar

A **BNF grammar** is **ambiguous** if there exists a sentence with multiple valid parse-trees.

These, small pieces of ambiguity can crop up anywhere. Consider the following example:

```
<expr> ::= x
         | if <expr> then <expr>
         | if <expr> then <expr> else <expr>
    ...
```

Figure 4.9: An ambiguous grammar `if` expressions.

In Figure (4.9) contains sub-cases which can lead to ambiguity. As, "is it the sub-case or the super-case?" For example, consider:

```
if x then (if y then z else w)   and   if x then (if y then z) else w
```

These problems often arise when interpreting expressions at the linear level. On an important note:

Theorem 1.1: Ambiguity is Undecidable

It is impossible to write a generalized program to determine if a given grammar set is ambiguous. This requires to finding all possible sentences; **However**—our grammar is recursive—this would mean generating an infinite number of sentences.

Though we can avoid ambiguity by specifying our operation order:

Definition 1.6: Fixity

The **fixity** of an operator refers to its placement relative to its operands:

- **Prefix**: The operator appears *before* its operand (e.g., $f\ x$, $-x$).

- **Postfix**: The operator appears *after* its operand (e.g., $a!$ for factorial or dereferencing).

- **Infix**: The operator appears *between* two operands (e.g., $a * b$, $a + b$, $a \mod b$).

- **Mixfix**: The operator is interleaved with its operands, appearing in multiple positions (e.g., `if` b `then` x `else` y).

So by specifying only Prefix or only Postfix operations ambiguity can be avoided:

Definition 1.7: Polish Notation

Polish Notation is a notation for arithmetic expressions in which the operator precedes its operands. For example, the polish notation $-/+2*1-23$ is written as $-(2+(1*(-2)/3))$.

In contrast, **Reverse Polish Notation** (RPN) places the operator after its operands. For example, Infix: $(3+4)*5$ is $3\ 4+5*$ in RPN.

However, this is not always practical. Consider `if` expressions dealt this way:

```
<expr> ::= <bool>
         | <var>
         | ifthen <expr> <expr>
         | ifthenelse <expr> <expr> <expr>

<bool> ::= tru | fls
<var> ::= x | y | z
```

Figure 4.10: An unambiguous grammar for `if` expressions.

Likewise, if we decided to enforce parenthesis everywhere, it might be a bit cumbersome:

```
<expr> ::= ( <op1> <expr> )
         | ( <expr> <op2> <expr> )        (((not x) and (not (not y))) or z)
         | <var>
                                          ((((x or y) or z) or x) or y)
                                    ⟹
<op1> ::= not                             (not ((not x) and (not y)) or (x and z))
<op2> ::= and | or
<var> ::= x | y | z                       (x and y)
```

Figure 4.11: An unambiguous grammar for `and or not` expressions, with parenthesis everywhere.

Theorem 1.2: The Ambiguity of Associativity & Precedence Rules

- **Associativity**: $a+b+c$ is ambiguous, as it could be $(a+b)+c$ or $a+(b+c)$.

- **Precedence**: $a+b*c+d$ is ambiguous to a computer, as it could be $(a+b)*c+d$.

We can resolve this by breaking the symmetry of our grammar:

Theorem 1.3: Resolving Associativity: Breaking Symmetry

The reason for ambiguity often comes symmetry in grammar rules of form:

```
<expr> ::= <expr> <op> <expr>
  <op> ::= + | - | ...
```

This means that the left or right side can be expanded indefinitely. By breaking this symmetry, we can resolve ambiguity:

```
<expr> ::= <var> <op> <expr>
  <op> ::= + | - | ...
 <var> ::= x | y | z | ...
```

In the above we fixed the left side to be a `<var>` and the right side to be an `<expr>`. This makes our expression **right-associative**. Vice-versa, if we fixed the right side, it would be **left-associative**.

Theorem 1.4: Resolving Precedence: Factor Out Higher Precedences

Precedence issues arise when operations reside on the same level. For instance,

```
<expr>::= <expr> + <expr> | <expr> * <expr>
```

Here, it's unclear whether + or - should be evaluated first. By factoring out higher precedence operations, we can resolve this, as, **terms deeper in the parse tree are evaluated first**:

```
<expr>::= <expr> + <term> | <term>
<term>::= <term> * <var> | <var>
 <var>::= x | y | z | ...
```

Here, * has higher precedence than +, as it's deeper in the parse tree. To handle **parentheses**, we can introduce another factor:

```
<expr>::= <expr> + <term> | <term>
<term>::= <term> * <pars> | <pars>
<pars>::= var | ( <expr> )
 <var>::= x | y | z | ...
```

Intuitively, in expressions like `(a + b) * c`, we want to treat `(a + b)` as a single unit/*variable*. **Note**, the parentheses are tokens surrounding the non-terminal `<expr>`.

Now consider the `and or not` expressions from Figure (4.7) resolving the associative ambiguity:

Grammar:

```
<expr> ::= <op1> <expr>
         | <var> <op2> <expr>
         | <var>
<op1> ::= not
<op2> ::= and | or
<var> ::= x | y | z
```

Sentence:

```
not x and y or z
```

Parse-tree:

```
              <expr>
             /      \
         <op1>      <expr>
           |         /  |  \
          not    <var> <op2> <expr>
                   |     |     |
                   x    and   ...
                       <var> <op2> <expr>
                         |     |     |
                         y    or   <var>
                                     |
                                     z
```

Figure 4.12: An unambiguous grammar for `and or not` expressions, by breaking symmetry.

Now we put all the rules together:

Grammar:

```
<expr> ::= <term> + <expr>
         |  <term>
<term> ::= <term> * <pars>
         |  <pars>
<pars> ::= <var> | ( <expr> )
<var> ::= x | y | z
```

Sentence:

```
x + z * ( x + y )
```

Parse-tree:

```
                        <expr>
                   /       |       \
              <term>       +       <expr>
                |                    |
             <pars>               <term>
                |                    |
             <var>              /         \
                |             *           <pars>
                x                           |
                                        ( <expr> )
                       <term>
                         |
                      <pars>        <term>    +    <expr>
                         |            |              |
                      <var>        <pars>         <term>
                         |            |              |
                         z         <var>          <pars>
                                      |              |
                                      x           <var>
                                                    |
                                                    y
```
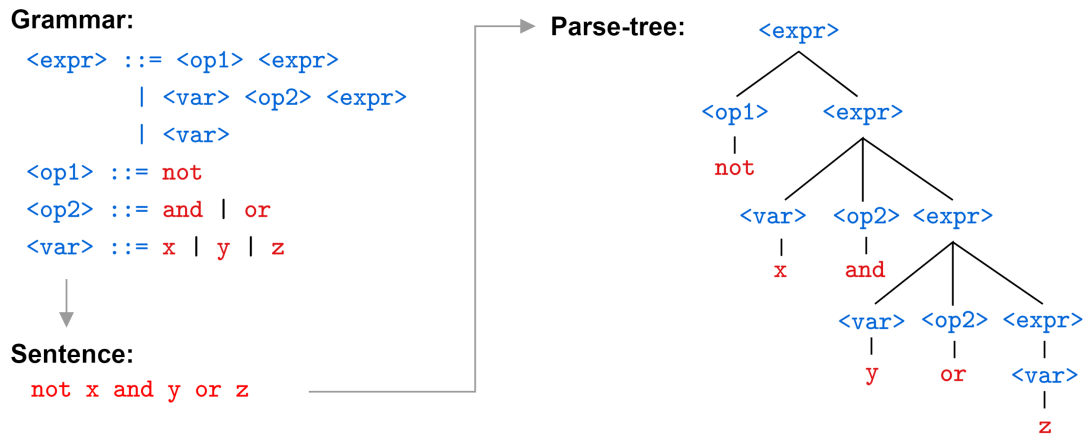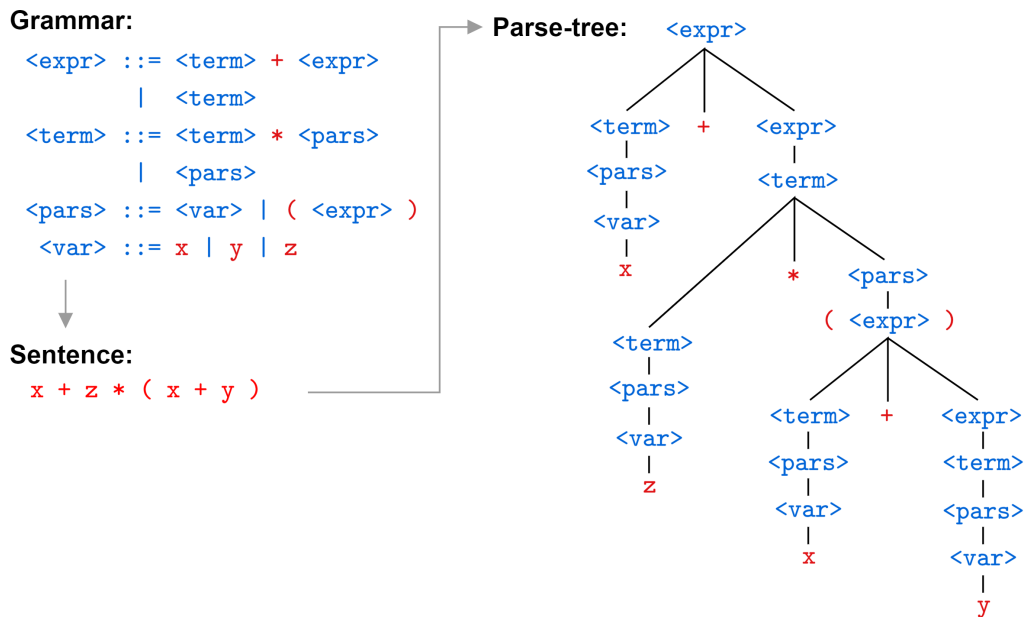
Figure 4.13: An unambiguous grammar for `+, *` expressions, by breaking symmetry and precedence.

Tip: The "Cult of Parentheses" in Lisp: Lisp-style languages rely heavily on **prefix notation**, where operators and function names appear **before** their arguments. This leads to deeply nested **parentheses**, often making Lisp code look overwhelming to newcomers.

For example, the Fibonacci function in Lisp is written as:

```
(defun fib (n)
  "Return the nth Fibonacci number."
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

Since **everything** in Lisp is an expression and follows a **uniform syntax**, parentheses are required for every function call, condition, and operation. This often leads to **excessive nesting**.

4.2   Formal: Parsing, Lexing, & Grammar Types

4.2.1   Parsing Generators & Grammar Types

Lexical analysis and parsing are closely linked to compiler design. Though instead of writing such parsers by hand, we will use **parser generators**:

Definition 2.1: Parser Generators

Parser generators are programs that take a representation of formal grammar as input and produces a parser as output.

We'll use **Menhir** for OCaml. There are other options but this suffices for our needs.

Definition 2.2: Menhir

Menhir is an LR(1) parser generator for OCaml. LR(1) stands for:

- **L**: Left-to-right scanning of the input.

- **R**: Rightmost derivation in reverse. Meaning, it builds the parse tree from the leaves up to the root.

- **(1)**: One symbol of lookahead. This means that the parser can look at one token ahead in the input stream to make parsing decisions.

Users input small OCaml snippets to define the production rules of the grammar.

A quick aside on Domain Specific Languages (DSLs):

Definition 2.3: Domain Specific Language (DSL)

A domain-specific language (DSL) is a programming language or specification language dedicated to a particular problem domain. An example of a DSL is SQL, as its only purpose is to query databases. This is in contrast to general-purpose languages like Python, which can be used for a wide range of applications.

Now we discuss **Extended BNF (EBNF)**, which allows us to be more expressive in our grammars with less noise.

Definition 2.4: Extended BNF (EBNF)

Extended Backus-Naur Form (EBNF) is a notation for specifying formal grammars, building upon BNF (Backus-Naur Form) by introducing syntactic extensions for greater clarity and brevity. EBNF includes the following additional constructs:

- `|` — alternation: denotes choice between patterns.
  - **E.g.,** `<letter> ::= A | B | C`, means that <letter> can be A, B, or C.
- `{…}` — repetition: the enclosed pattern may repeat **<u>zero</u> or more times**.
  - **E.g.,** `<digits> ::= {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}`, matches:
    > "0", "320", "11235813"
- `[…]` — optionality: the enclosed pattern may appear zero or one time.
  - **E.g.,** `<number> ::= ["-"] <digits>`, matches:
    > "1", "-320"
- `(…)` — grouping: used to group sequences or alternatives.
  - **E.g.,** `<sum> ::= <number> ("+" | "-") <number>`, matches:
  > "1 + 2 - 3", "-1 + 2", "5"
- `"…"` — terminal strings may be written in quotes. This clears ambiguity between terminal and non-terminal symbols.
  - **E.g.,** in `<digit> ::= A | B | C`, it's clear that A, B, and C refer to non-terminal symbols. However, in the expression `<expr> ::= <expr> | "(" <expr> ")"`, the parenthesis are **crucial** in indicating that it is not grouping notation.

Let's elaborate more on `<digits>` from above:

Example 2.1: Dealing with Optionality

In our previous grammar from Definition (2.4), we had:

```
<digit> ::= {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
```

which matched:

> "0", "320", "11235813"

But also unintentionally matched:

> ""   *(empty string)*

> "000", "0123"   *(leading zeros)*

Let's first fix the empty string case:

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digits> ::= <digit> {<digit>}
```

To restrict leading zeros (e.g., for natural numbers), we refine further:

```
<nonzero> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digits> ::= "0" | <nonzero> {<digit>}
```

Adding quotes around `0` is a stylistic choice and does not change the meaning of the grammar. We could have written:

```
<nonzero> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<digits> ::= "0" | <nonzero> {<digit>}
```

Or without quotes, it doesn't matter as long as there is no ambiguity.   ■

To be more precise with our language we briefly discuss **Automata Theory** and types of grammars:

Definition 2.5: Automata Theory

In mathematics, an automaton (Plural form: automata) is an abstract machine that follow a sequence of states (similar to a flowchart). A finite number of states describe a **finite automaton (FA)** or **finite-state machine (FSM)**.

Tip: Automata comes from the Greek word $\alpha\upsilon\tau\acute{o}\mu\alpha\tau\sigma\varsigma$ (autómatos), which means "self-acting, self-willed, self-moving"

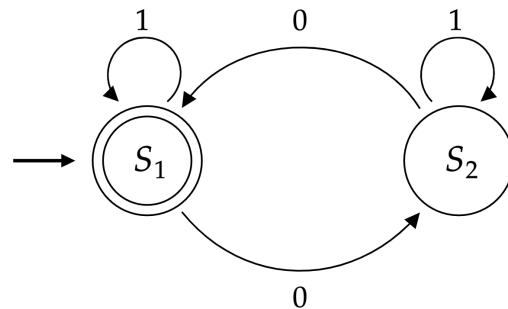Consider the following automaton described by a state machine diagram:

Figure 4.14: An automaton represented as a state-machine diagram. This state-machine generates binary strings with an even number of 0's. There are two states, S_1 and S_2. The automaton starts in state S_1 as shown by the arrow coming from blank space. The double circle indicates the **accepting state**, at which the automaton may terminate.

Definition 2.6: Grammar Types

In formal language theory, grammars are classified into a hierarchy based on their generative power. This classification, known as the **Chomsky hierarchy**, comprises four primary types that utilize BNF:

- **Type 3: Regular Grammars**
  Generate **regular languages**, which can be recognized by finite automata. These grammars can describe simple flat patterns, but cannot handle nested or recursive structures.

- **Type 2: Context-Free Grammars (CFGs)**
  Generate **context-free languages** and are recognized by pushdown automata. A **pushdown automaton** is a finite automaton extended with a stack—a Last-In, First-Out (LIFO) memory structure. This allows it to handle nested and recursive structures, such as balanced parentheses.

- **Type 1: Context-Sensitive Grammars**
  Generate **context-sensitive languages** and are recognized by linear bounded automata. A **linear bounded automaton** is a Turing machine where the tape is limited to a length proportional to the input size. This constraint gives it more power than pushdown automata but less than unrestricted Turing machines.

- **Type 0: Unrestricted Grammars**
  Generate **recursively enumerable languages** and are recognized by Turing machines. These grammars have no restrictions on production rules and can describe any computable language, though some of these languages are undecidable.
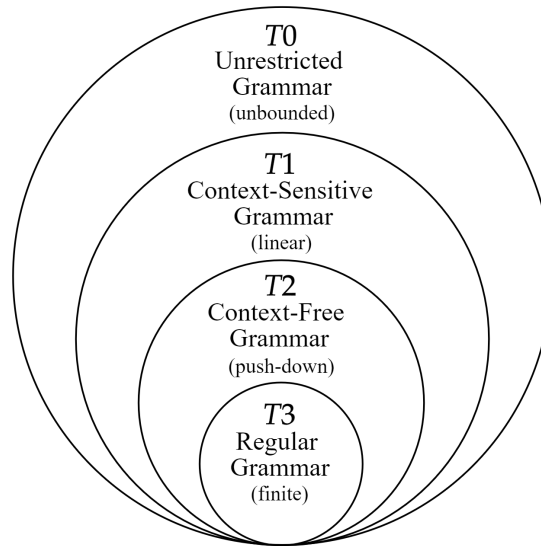
We can visualize the Chomsky hierarchy as follows:

Figure 4.15: The Chomsky hierarchy: T_3 finite automata (regular expressions), T_2 pushdown automata (context-free grammars), T_1 linear bounded automata (context-sensitive grammars), and T_0 Turing machines (unrestricted grammars).

Definition 2.7: Regular Expressions (Regex)

Regular expressions (Regex) provide a compact way to describe patterns in regular grammars:

- `a` — a single **terminal** symbol is itself a regex.

- `[t1 t2 …tk]` — character class: matches any one of the listed characters.

- `(e1 | e2 | …| ek)` — alternation: matches any one of the enclosed expressions.

- `exp*` — repetition: matches **zero or more occurrences** of `exp`.

- `exp+` — repetition: matches **one or more occurrences** of `exp`.

- `exp?` — optional: matches **zero or one occurrence** of `exp`.

E.g., the regex `(a|b)*c` matches any string of a's and b's followed by a c, such as:

> `"abc", "aabbbbc", "c"`

This is just a small subset of the Regex syntax. To learn more, consider the following resource: https://regexlearn.com/

4.2.2 Menhir: Simple Lexing & Parsing

To formally retouch on the definitions of lexing:

Definition 2.8: Lexical Analysis

Lexical analysis concerns itself with the **tokenization** of a program. In particular, converting a stream of characters into a stream of tokens (whitespace and comments are ignored).

Lexing (to lex) works by scanning the input for the next longest valid token. Then it returns that token and the remaining input to lex.

E.g., `" let@#_)($#@_J_@O#GKJ"` gives us `(LET, "@#_)($#@_J_@O#GKJ")`, where `LET` is the token and `"@#_)($#@_J_@O#GKJ"` is the remaining input, which is most likely to result in an error (unless the lexer is designed to handle such cases).

We will now create a lexer and parser for a simple toy-language that only has addition, subtraction, multiplication, and division. Menhir should already be installed from the onboarding Section (1.2.3).

Example 2.2: Simple Lexer & Parser (Part 1)

Let's create a simple lexer and parser for a toy language that only has addition, subtraction, multiplication, and division. We will use Menhir to generate the parser.

Let's create the project:

```
dune init proj small_arith
cd small_arith
```

This should create:

```
small_arith/
├── _build/
│   └── log
├── bin/
│   ├── dune
│   └── main.ml
├── dune-project
├── lib/
│   └── dune
├── small_arith.opam
└── test/
    ├── dune
    └── test_small_arith.ml
```

∎

Example 2.3: Simple Lexer & Parser (Part 2)

Create these new files, `lib/ast.ml` , `lib/lexer.mll` , and `lib/parser.mly` :

```
touch lib/dune lib/ast.ml lib/lexer.mll lib/parser.mly
```

- `ast.ml` — A regular OCaml source file (`.ml`) where we define the core data structures via ADTs, which will generate the **abstract syntax tree (AST)** for our language.

- `lexer.mll` — The (`.mll`) extension to signal it will be processed by **OCamllex**, a lexer generator. Here, we write rules using regular-expression-like syntax to convert raw strings into **tokens** like NUM, ADD, and LPAREN.

- `parser.mly` — The (`.mly`) extension is handled by **Menhir**. It defines the **grammar** of our language. The result of parsing is an AST defined in `ast.ml` .

Together: Input String → Lexer (mll) → Tokens → Parser (mly) → AST (ml)

Lets begin by defining the AST in `ast.ml` :

```ocaml
(* ast.ml *)
type bop = Add | Sub | Mul | Div

type expr =
  | Var of string
  | Num of int
  | Let of string * expr * expr
  | Bop of bop * expr * expr

type prog = expr
```

The above code defines the abstract syntax tree (AST) for our toy language. It includes:

- `bop` — a type for binary operators (addition, subtraction, multiplication, division).

- `expr` — a type for expressions, which can be a variable, number, let-binding, or binary operation.

- `prog` — a type for the program, which is just an expression.

This should be familiar from the BNF grammar we've discussing.   ∎

Example 2.4: Simple Lexer & Parser (Part 4)

Now let's define our lexer in `lib/lexer.mll`:

```
(* lexer.mll *)
{
    open Parser
}

(* Regex for tokens *)
let whitespace = [' ' '\t' '\n' '\r']+
let var = ['a'-'z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_' '\'']*
let num = ['0'-'9']+

rule read = parse
    | whitespace { read lexbuf }
    | "let"      { LET }
    | "in"       { IN }
    | "="        { EQ }
    | "+"        { ADD }
    | "-"        { SUB }
    | "*"        { MUL }
    | "/"        { DIV }
    | "("        { LPAREN }
    | ")"        { RPAREN }
    | num        { NUM (int_of_string (Lexing.lexeme lexbuf)) }
    | var        { VAR (Lexing.lexeme lexbuf) }
    | eof        { EOF }
```

- `lexbuf` : short for **"lexing buffer,"** is an object created by the OCaml stdlib. It stores the input string and tracks the position of the lexer as it reads through.

- `read lexbuf` : A recursive call to the lexer function. It's used to **skip over whitespace**. When the lexer sees a space or newline, it just calls itself again to scan the next part of the input.

- `Lexing.lexeme lexbuf` : This function returns the **exact substring** that was matched by our Regex. For example, if the input is "42 + 1" and we match `num`, then `Lexing.lexeme lexbuf` returns "42" as a string. We then convert it to an integer.

- `rule read = parse` : This defines the main lexer function. The `read` is the entry point of the lexer. The `parse` keyword indicates parsing rules for the input to match against.

∎

Example 2.5: Simple Lexer & Parser (Part 5)

Finally the parser in `lib/parser.mly` utilizing our `ast.ml` (Moreover on the next page.):

```
%{
  open Ast
%}

%token LET EQ IN
%token ADD SUB MUL DIV
%token LPAREN RPAREN
%token <int> NUM
%token <string> VAR
%token EOF

%left ADD SUB
%left MUL DIV

%start <Ast.prog> prog
%%

prog:
  | e = expr; EOF { e }

expr:
  | LET; x = var; EQ; e1 = expr; IN; e2 = expr
    { Let (x, e1, e2) }
  | e = expr1 { e }

%inline bop:
  | ADD { Add }
  | SUB { Sub }
  | MUL { Mul }
  | DIV { Div }

expr1:
  | e1 = expr1; op = bop; e2 = expr1
    { Bop (op, e1, e2) }
  | n = num { Num n }
  | v = var { Var v }
  | LPAREN; e = expr; RPAREN { e }

num:
  | n = NUM { n }

var:
  | x = VAR { x }
```

∎

Example 2.6: Simple Lexer & Parser (Part 5a)

The `parser.mly` file looks quite different from regular OCaml code because it follows the syntax of Menhir (an LR(1) parser generator). Let's walk through its structure:

- `%{ ... %}` — Opens a block of OCaml code. We use it to import our AST module.

- `%token` — Declares the tokens the parser will receive from the lexer. If tokens carry data integer or string, we write it as:

```
%token <int> NUM
%token <string> VAR
```

- `%left` — Defines left-associativity, and precedence is by order of declaration:

```
%left ADD SUB
%left MUL DIV
```

   means that `MUL` and `DIV` bind more tightly than `ADD` and `SUB`. All are left-associative. This prevents ambiguity in expressions like `3 + 4 * 2`.

- `%start <Ast.prog> prog` — The entry point of the parser is the rule `prog`. The return type is `Ast.prog` from our AST module.

- `<name>:` — Each rule has the form:

```
name:
  | pattern1 { action1 }
  | pattern2 { action2 }
```

   Think of it as EBNF, but instead we use OCaml pattern matching syntax.

- `%inline <name>:` — A helper rule that lets us **macro** grammar patterns into a single variable without generating a new nonterminal or type (cannot be recursive):

```
%inline bop:
  | ADD { Add }
  | SUB { Sub }
  | ...
```

∎

Example 2.7: Simple Lexer & Parser (Part 5b)

Let's look closely at the following grammar rule:

```
expr:
    | LET; x = var; EQ; e1 = expr; IN; e2 = expr
    { Let (x, e1, e2) }
```

- Each item before the '`{`' is a **token** (e.g., `LET`, `EQ`, `IN`) or a **nonterminal rule** (e.g., `var`, `expr`).

- When a nonterminal is matched, we can bind its result using `name = rule`. For example, `x = var` means: "run the `var` rule and bind its result to `x`."

- The block in `{ }` is an OCaml expression that builds an AST node using the bound variables. Notice, `Let (x, e1, e2)` our ADT constructor from `ast.ml`.

When we parse a string like:

```
let x = 3 + 4 in x * 2
```

The lexer produces a stream of tokens:

```
[LET; VAR("x"); EQ; NUM(3); ADD; NUM(4); IN; VAR("x"); MUL; NUM(2); EOF]
```

Then, the parser:

- Sees `LET`, then `VAR("x")` → runs the `var` rule, binds to `x`.

- Sees `EQ`, then parses the expression `3 + 4` using the `expr` rule, binds to `e1`.

- Sees `IN`, then parses the expression `x * 2`, binds to `e2`.

Finally, it constructs:

```
Let (x, e1, e2)
```

This is now a fully constructed OCaml value using the constructors we defined in `ast.ml`. ∎

Example 2.8: Simple Lexer & Parser (Part 6)

Now that we've written our AST, lexer, and parser, we need to set up Dune so we can compile everything and test it in `utop`.

First, open your `dune-project` file and add the following line at the bottom to enable Menhir:

```
(using menhir 3.0)
```

Your complete `dune-project` should now look like:

```
(lang dune 3.0)
(name small_arith)
(using menhir 3.0)
```

Next, open `lib/dune`:

```
(library
    (name small_arith))

(ocamllex lexer)
(menhir (modules parser))
```

Create a `lib/demo.ml` file to test the lexer and parser:

```
let parse s =
    try Some (Parser.prog Lexer.read (Lexing.from_string s))
    with _ -> None
```

From the root of your project, run:

```
dune build
dune utop lib
```

In `utop`, you can test the parser like this:

```
# open Demo;;
# parse "let x = 3 + 4 in x * 2";;
- : prog option =
Some
(Let ("x",
     Bop (Add, Num 3, Num 4),
     Bop (Mul, Var "x", Num 2)
))
```

∎

4.3   Semantic Evaluation

4.3.1   Small-step Semantics

In our previous derivations, we've been doing **Big-step semantics**:

Definition 3.1: Big-Step Semantics

Big-step semantics describes how well-formed expressions evaluate to a final value, without detailing each intermediate step.

Notation: We write $e \Downarrow v$ to mean that the expression e evaluates to the value v.
Example:
$$(\text{sub } 10 \ (\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3))) \Downarrow 2$$

Here, we now introduce **Small-step semantics**:

Definition 3.2: Small-Step Semantics

Small-step semantics describes how an well-formed expressions reduce one step at a time until a final value is reached. **Notation:** $e \to e'$, means that e reduces to e' in a single step. These semantics are housed in a **configuration** with some state-structure and program. We write:

$$\underbrace{\langle S, p \rangle}_{\text{configuration}} \quad \longrightarrow \quad \underbrace{\langle S', p' \rangle}_{\text{transformation.}}$$

Where S is the state of the program and p is the program. The rightarrow shows the **transformation** or **reduction** of the program. For purposes in FP there is no state, hence:

$$\langle \varnothing, p \rangle \longrightarrow \langle \varnothing, p' \rangle$$

Moving forward we <u>**shorthand**</u> this to $p \to p'$ for brevity. Inference rules outline the semantics of the language:

$$\frac{e_1 \to e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2} \ (\text{reduce-left-first})$$

This rule states that if the e_1 position is reducible ($e_1 \to e_1'$), reduce it first.

Let's try defining some small-step semantics for a toy-language:

Example 3.1: Defining Grammars in Small-Step Semantics

Say we have some grammar that can only add from 0 to 9:

```
<expr> ::= "(" <op> <expr> <expr> ")"
         | <int>
<op>   ::= add
<int>  ::= 0-9
```

Let's assume our language reads from **left-to-right** and define the semantics of `add`:

- **Both arguments are expressions:**

$$\frac{e_1 \to e_1'}{(\text{add } e_1 \ e_2) \to (\text{add } e_1' \ e_2)} \ (\text{add-left})$$

- **Left argument is an integer:**

$$\frac{n \text{ is an int} \qquad e_2 \to e_2'}{(\text{add } n \ e_2) \to (\text{add } n \ e_2')} \ (\text{add-right})$$

- **Both arguments are integers:**

$$\frac{n_1 \text{ and } n_2 \text{ are int} \qquad n_1 + n_2 = v \text{ (integer addition)}}{(\text{add } n_1 \ n_2) \to v} \ (\text{add-ok})$$

When writing semantics (in this case **add**) think, "What are all the possible argument states?" If we have (`add 5 e2`), we must figure out what e2 represents in order to add.

We can almost think of these terminal-symbols as **base cases**. Additionally, since we read left-to-right, we do not need a rule for (`add e1 n`), where $e1$ is an expression and n is an integer. This scenario is already covered by (add-left). ∎

Tip: States can represent data structures like stacks, making them ideal for modeling stack-oriented languages. For example (ϵ is the empty program):

$$(\varnothing, \texttt{push 2; push 3; add})$$
$$\to \quad (2 :: \varnothing, \texttt{push 3; add})$$
$$\to \quad (3 :: 2 :: \varnothing, \texttt{add})$$
$$\to \quad (5 :: \varnothing, \epsilon)$$

Definition 3.3: Multi-Step Semantics

Multi-step semantics captures the idea of reducing a configuration through **zero or more single-step reductions**. We write $C \to^\star D$ to mean that configuration C reduces to configuration D in zero or more steps. This relation is defined inductively with two rules:

$$\frac{}{C \to^\star C} \text{ (reflexivity)} \qquad\qquad \frac{C \to C' \quad C' \to^\star D}{C \to^\star D} \text{ (transitivity)}$$

These rules formalize:

- Every configuration reduces to itself   *(reflexivity)*

- Multi-step reductions can be extended by single-step reductions   *(transitivity)*

- If there are multiple ways to reduce $C \to^\star D$, then the semantics are **ambiguous**.

Example 3.2: Proving Multi-step Reductions

We show (add (add 3 4) 5)$\to^\star$ 14 based off semantics defined in Example (3.1). We will do multiple rounds of single-step reductions to yield a result. At each round, read bottom up:

$$1. \quad \frac{\dfrac{\dfrac{}{\text{add } 3\ 4 \to 7} \text{ (add-ok)}}{\text{add } 5\ (\text{add } 3\ 4) \to \text{add } 5\ 7} \text{ (add-right)}}{(\text{add } (\text{add } 5\ (\text{add } 3\ 4))\ 2) \to (\text{add } (\text{add } 5\ 7)\ 2)} \text{ (add-left)}$$

$$2. \quad \frac{\dfrac{}{(\text{add } 5\ 7) \to 12} \text{ (add-ok)}}{(\text{add } (\text{add } 5\ 7)\ 2) \to (\text{add } 12\ 2)} \text{ (add-left)}$$

$$3. \quad \frac{}{(\text{add } 12\ 2) \to 14} \text{ (add-ok)}$$

Thus, (add (add 3 4) 5) $\to^\star$ 14. Following rules, we match on the next outermost pattern. Our very first reduction matches with (add-left). In particular, $e_1 :=$ (add 5 (add 3 4)), and we see that that's our starting value the next layer up.

Moreover, the trailing 2 in (add (add 5 (add 3 4)) 2), is not evaluated until the very last step (3), as we read from left-to-right. Even though *we can see it*, our patterns do not.

■

Though there is a clear distinction between big-step and multi-step semantics:

Theorem 3.1: Multi-Step vs. Big-Step Semantics

Both Multi-step and Big-step semantics have the same outcome, but differ in meaning.

$$e \to^\star v \quad \approx \quad e \Downarrow v$$

Unlike big-step semantics, small-step semantics concerns all possible patterns, allowing us to choose how we reduce terms, and **in which order**. This eliminates possible ambiguity in in big-step pattern matching.

4.3.2 Lambda Calculus

We briefly touched on **Lambda Calculus** in a previous section when discussing the Anonymous Function Definition (3.4). Here we go more in-depth:

Definition 3.4: Lambda Calculus Syntax

Lambda calculus is a formal system for representing computations using only single-argument functions, avoiding the need for multiple parameters or state. Lambda calculus has three basic constructs:

- **Variables:** x, y, z, etc.

- **Abstraction:** $\lambda x.e$ (a function that takes an argument x and returns expression e).

- **Application:** $e_1 \, e_2$ (applying function e_1 to argument e_2).

The OCaml translation:

$$\lambda x.e \quad \equiv \quad \texttt{fun } x \texttt{ -> } e$$

Replacing 'fun' with 'λ', and '`->`' with '.'. In pure lambda calculus (no additional syntax), **the only value** is a lambda **abstraction**.

Definition 3.5: The Identity Function

The identity function is a function that returns its argument unchanged. Represented as: $\lambda x.x$. If we introduce numbers, $(\lambda x.x)\ 5 \to 5$ (application). In OCaml, this is represented as: `(fun x -> x) 5`. This is also know as the "I" combinator.

Before moving on we address a few notational conventions:

Definition 3.6: Symbols $\triangleq$ vs. :=

The symbol `:=` is used to define a variable or expression. The symbol $\triangleq$ is used to state that two expressions are equal **by definition**. For example, we may write a paper which reuses some large specific configuration $\langle \{ \ldots \}, \ldots \rangle$; Instead of writing it again and again, we assign one variable to represent such idea:

$$\Delta_\Pi^\star \triangleq \langle \{ \ldots \}, \ldots \rangle$$

Now throughout our paper, $\Delta_\Pi^\star$ signals to the reader that we are using this configuration. As opposed to `:=`, where we might temporarily assign the variable a to some value multiple times over the course of a document.

Next we look at what happens when we apply the identity function to itself:

Definition 3.7: The Diverging Term Ω

Lambda Calculus is capable of recursion; The backbone of which is the **diverging term**, for which we define as Ω (Omega):

$$\Omega \triangleq (\lambda x.x\,x)(\lambda x.x\,x)$$

The inner function $\lambda x.x\,x$ is sometimes called the **mockingbird combinator**, as it applies its argument to itself:

$$M \triangleq \lambda x.x\,x$$

Thus, $\Omega = M\,M$ creates an infinite loop of self-application.

Example 3.3: Showing Ω Divergence

We can show that Ω diverges by repeated self-application:

$$\begin{aligned}
\Omega &\triangleq (\lambda x.x\,x)(\lambda x.x\,x) \\
&\rightarrow (\lambda x.(\lambda x.x\,x)\,(\lambda x.x\,x)) \\
&\rightarrow (\lambda x.(\lambda x.(\lambda x.x\,x)\,(\lambda x.x\,x))) \\
&\rightarrow (\lambda x.(\lambda x.(\lambda x.(\lambda x.x\,x)\,(\lambda x.x\,x)))) \\
&\rightarrow \ldots
\end{aligned}$$

Hence, Ω diverges as it continues to apply itself indefinitely.   ∎

Application has a formal definition in lambda calculus:

Definition 3.8: Application & β-Reduction

A (beta) β**-reduction** is the process of applying a function as an argument in lambda calculus:

1.
$$\frac{e_1 \to e_1'}{e_1 \ e_2 \to e_1' \ e_2} \text{ (beta-left)}$$

2.
$$\frac{e_2 \to e_2'}{(\lambda x.e_1) \ e_2 \to (\lambda x.e_1) \ e_2'} \text{ (beta-right)}$$

3.
$$\frac{}{(\lambda x.e) \ (\lambda y.e') \to [(\lambda y.e')/x]e} \text{ (beta-ok)}$$

Where (1) we reduce the left-hand side of the application, (2) we reduce the right-hand side of the application, and (3) we apply a function to another function by substitution.

Note: (3) is our base case where **only values** are allowed to be substituted. We can take a different approach and not **eagerly** evaluate expressions.

1.
$$\frac{e_1 \to e_1'}{e_1 \ e_2 \to e_1' \ e_2} \text{ (beta-left)}$$

2.
$$\frac{}{(\lambda x.e) \ e' \to [e'/x]e} \text{ (beta-ok)}$$

Note: (2) though we don't evaluate e' immediately, we **still** expect it to be a value when evaluating the full expression. As when we evaluate, we are expecting a value, **not some unknown variable**.

Example 3.4: Simple β-Reduction

Consider the following example of β-reduction:

$$(\lambda x.x + 1) \ 2 \quad \to \quad [2/x](x + 1) \quad \to \quad 2 + 1 \to 3$$

Here, we supply the function with 2, substitute 2 for x in the body of the function, and then evaluate the expression, yielding 3. ∎

Though we must be wary of what we are substituting for:

Definition 3.9: α-Equivalence

Two expressions are (alpha) α**-equivalent** if they differ only variable names. This formalizes the **principle of name irrelevance**: renaming bound variables does not change the meaning of an expression.

$$\lambda x.\lambda y.x \ =_\alpha \ \lambda v.\lambda w.v$$

In OCaml-like syntax:

$$\texttt{let x = 2 in x} \ =_\alpha \ \texttt{let z = 2 in z}$$

Substitution should preserve α-equivalence. If $e_1 =_\alpha e_2$, then for any term v, we have:

$$[v/x]e_1 =_\alpha [v/x]e_2$$

To continue we make the following distinction:

Definition 3.10: Free & Bound Variables

In lambda calculus, a variable in an expression can be either **free** or **bound**:

- A variable is **bound** if it is defined by a λ abstraction in the expression.

    - **E.g.,** $(\lambda x.x + 1)$, the variable x is bound.

- A variable is **free** if it is not bound by any enclosing λ abstraction.

    - **E.g.,** $(\lambda x.y + 1)$, the variable y is free, **but** x is still bound.

Formally, the set of free variables in an expression e, written $FV(e)$, is defined inductively as:

$$
\begin{aligned}
FV(x) &= \{x\} & \textit{(Alone variable)} \\
FV(\lambda x.e) &= FV(e) \setminus \{x\} & \textit{(Function body)} \\
FV(e_1 \ e_2) &= FV(e_1) \cup FV(e_2) & \textit{(Application)}
\end{aligned}
$$

In just a moment, we will define substitution in a way that preserves α-equivalence. The high-level idea is that we should **avoid** substituting variables that are **bound** within an expression.

Now we define the semantics of substitution:

Definition 3.11: Substitution Semantics

Substitution replaces **free** occurrences of a variable with another expression. The rules are defined inductively as follows, where in $[v/x]e$, v is assumed to be a **value** (abstraction):

$$(1) \qquad [v/y]x = \begin{cases} v & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

$$(2) \quad [v/y](\lambda x.e) = \begin{cases} \lambda x.e & \text{if } x = y \\ \lambda z.[v/y]([z/x]e) & \text{if } x \in \text{FV}(v), \text{ choose a fresh} z \\ \lambda x.[v/y]e & \text{otherwise} \end{cases}$$

$$(3) \quad [v/y](e_1 \ e_2) = ([v/y]e_1) \ ([v/y]e_2)$$

Moreover on (2)'s middle condition. $FV(v)$ means free variables in v, so if $v := (\lambda w.y)$ then $FV(v) = \{y\}$. Then $[v/x](\lambda \ y.x)$ would be a major problem:

$$\lambda y.\lambda w.y \ \equiv \ \lambda y.y \ \neq_\alpha \ \lambda y.x$$

The middle condition avoids this by swapping in a **fresh variable** z that does not have any conflicts in the body. **E.g.,** $[v/x](\lambda y.x(\lambda z.y))$, $y \in FV(v)$, ignoring freshness of z:

$$\big(\lambda z.v(\lambda z.z)\big) \neq_\alpha \big(\lambda y.x(\lambda z.y)\big)$$

The abstraction $(\lambda z.y)$ **captures** the subbed z in $(\lambda z.z)$. Now we pick some arbitrary **fresh** variable z, such that no captures occur:

$$\big(\lambda u.y(\lambda z.u)\big) =_\alpha \big(\lambda y.x(\lambda z.y)\big)$$

Here we chose the variable u as it does not conflict with the rest of the expression.

.

Example 3.5: Multi-step β-Reductions

Consider the following derivations of $(\lambda f.\lambda x.fx)(\lambda y.y)$ using our substitution semantics:

1.
$$\frac{}{(\lambda f.\lambda x.fx)(\lambda y.y) \to [(\lambda y.y)/f](\lambda x.fx) = (\lambda x.(\lambda y.y)x)} \text{ (beta-ok)}$$

Yielding $(\lambda x.(\lambda y.y)x)$. We **cannot** evaluate inner-terms. I.e., let $w := (\lambda y.y)x$, then we have, $(\lambda x.w)$. Hence, there is no rule in Definition (3.8) that allows further reduction. ∎

There are times where we can't evaluate a term

Definition 3.12: Stuck Terms

A **stuck term** is a well-formed expression that cannot be reduced, yet is not a value. Applying a non-function value to an argument often causes such issue:

$$((\lambda x.yx)(\lambda x.x)) \to y(\lambda x.x)$$

Here, the variable y is free, and thus, cannot proceed with application. Hence, we are "stuck" in the evaluation. We can avoid such scenarios via **typing systems** (discussed in Sec. (4.4.1)).

There are two main evaluation strategies in program evaluation:

Definition 3.13: Call-by-Value vs. Call-by-Name

There are two main evaluation strategies:

- **Call-by-value (CBV)** evaluates the argument *before* substitution.

- **Call-by-name (CBN)** substitutes the argument expression *directly* without evaluating it first.

We may illustrate this with the following rules:

$$(\text{CBV}) \quad \frac{e_1 \Downarrow \lambda x.e_1' \quad e_2 \Downarrow v_2 \quad [v_2/x]e_1' \Downarrow v}{e_1\ e_2 \Downarrow v} \qquad (\text{CBN}) \quad \frac{e_1 \Downarrow \lambda x.e_1' \quad [e_2/x]e_1' \Downarrow v}{e_1\ e_2 \Downarrow v}$$

The benefit of CBV is that it **only evaluates an argument once** and is reused. With CBN, the argument is evaluated **every time** it needs to be computed. This is good if an expensive computation is passed around, but barely touched in execution (e.g., expensive edge-case).

We saw this before in Definition (3.8). The first semantics were CBV, while the latter was CBN.

Tip: There are many evaluation strategies optimizing different aspects of computation. In addition to **CBV** and **CBN** there are: **Call-by-need** (lazy eval)—like call-by-name, but avoids recomputation by memoizing results. Used in Haskell. **Call-by-reference**—used in languages with pointers (functions receive variable references). **Call-by-sharing**—also pointer focused langues (functions receive object references).

Definition 3.14: Well-Scopedness and Closedness

We expand on the idea of **scope** in respect to free and bound variables:

- An expression e is **closed** if it contains **no free variables**, also called a **combinator**.

- An expression e is **well-scoped** if the expression is closed or is bound under the configuration's state.

Every closed expression is well-scoped, **but** not every well-scoped expression is closed.

Note: The expression $x \mapsto y$ means that x is mapped to y. Hence, x is bound to y.

Example 3.6: Closed vs. Open Terms

Recall that abstractions bind to their argument variable:

- **Open Term:** $(\lambda x.y)$ is *not closed*, since y is free.

- **Closed Term & Well-scoped:** $(\lambda x.\lambda y.y)$ is *closed*, since both x and y are bound.

- **Well-scoped:** $\langle \{x \mapsto (\lambda y.y)\}, (\lambda w.x) \rangle$ is *well-scoped*, since x is bound in state, but not closed.

■

Definition 3.15: Lexical vs Dynamic Scope

A variable's **scope** determines where in the program the variable can be referenced.

- **Lexical (or static) scope:** Textual delimiters define the scope of a binding.

- **Dynamic scope:** Bindings are determined at runtime based on the call stack. I.e., the most recent binding in the call stack is used regardless was declared.

To understand the difference between lexical and dynamic scoping:

Example 3.7: Dynamic vs Lexical Scoping

Consider the following Bash code:

```
f() { x=23; g; }
g() {   y=$x; }
f
echo $y    # prints 23
```

In Bash, the variable x is not defined in g, but since f called g and x was defined in f, g sees it. This is **dynamic scoping**. In contrast, consider the following Python code:

```python
x = 0
def f():
    x = 1
    return x

assert f() == 1
assert x == 0
```

Now consider the following OCaml code:

```ocaml
let x = 0
let f () =
    let x = 1 in
    x

let _ = assert (f () = 1)
let _ = assert (x = 0)
```

Both Python and OCaml use **lexical scoping**, meaning each use of x refers to the closest enclosing definition in the source code, not the caller's environment. ∎

Lastly, we can model if statements in lambda calculus:

Theorem 3.2: Lambda Calculus – True-False Conditions

True,false, and if conditions are defined as:

$$\texttt{true} \triangleq \lambda x.\lambda y.x \qquad \texttt{false} \triangleq \lambda x.\lambda y.y$$

$$\texttt{IF} \triangleq \lambda cond.\lambda then.\lambda else.cond\ then\ else$$

E.g.,
(IF true 1 2) $\to [2/y][1/x](\lambda x.\lambda y.x) \to 1$.
(IF false 1 2) $\to [2/y][1/x](\lambda x.\lambda y.y) \to 2$.

4.3.3 Handling Lambda Recursion

We have Ω which allows us to do recursion, but we need self-referencing.

Definition 3.16: Fixed-point Combinator

A **fixed point** is a value unchanged by a transformation (e.g., the fixed point of f is some value x such that, $f(x) = x$). A fixed-point combinator is a higher-order function that satisfies:

$$\text{FIX } f = f(\text{FIX} f)$$

i.e., functions `FIX` and f when applied returns f whose argument is the original application. This enables recursion, as there is a self reference in scope. This unfolds to an infinite series of applications: $(\text{FIX } f = f(\text{FIX} f) = f(f(\text{FIX} f)) = f(f(f(\text{FIX} f))) = \ldots)$. Whether or not it converges depends on the behavior of f (i.e., a base-case).

Example 3.8: Writing Recursive Functions

Say we defined the following recursive factorial function, extending our lambda syntax:

$$\text{FACT} \triangleq \lambda n.\texttt{if } n = 0 \texttt{ then } 1 \texttt{ else } n * \text{FACT}(n-1)$$

To supply `FACT` with its own definition, we may preform an intermediary step:

$$\text{FACT'} \triangleq \lambda f.\lambda n.\texttt{if } n = 0 \texttt{ then } 1 \texttt{ else } n * (ff(n-1))$$

We define `FACT'`, which takes an additional function f to supply its recursive case. Now, we can apply `FACT'` to itself to render our desired `FACT` function:

$$\text{FACT} \triangleq \text{ FACT' FACT'}$$

For example, let's supply 3 to `FACT`:

$$
\begin{aligned}
\text{FACT } 3 &= (\text{FACT}' \text{ FACT}') \, 3 && \text{Definition of FACT} \\
&= ((\lambda f. \lambda n.\texttt{if } n = 0 \texttt{ then } 1 \texttt{ else } n \times (f \, f \, (n-1))) \, \text{FACT}') \, 3 && \text{Definition of FACT'} \\
&\to (\lambda n.\texttt{if } n = 0 \texttt{ then } 1 \texttt{ else } n \times (\text{FACT}' \, \text{FACT}' \, (n-1))) \, 3 && \text{Application to FACT'} \\
&\to \texttt{if } 3 = 0 \texttt{ then } 1 \texttt{ else } 3 \times (\text{FACT}' \, \text{FACT}' \, (3-1)) && \text{Application to } n \\
&\to 3 \times (\text{FACT}' \, \text{FACT}' \, (3-1)) && \text{Evaluating if} \\
&\to \ldots \\
&\to 3 \times 2 \times 1 \times 1 \\
&\to^* 6 && [3]
\end{aligned}
$$

∎

We make the following distinction to emphasize the meaning of a fixed-point:

Theorem 3.3: The identity function & fixed-points

Any function f is a fixed-point of the identity function I ($\lambda x.x$), i.e., $I\ f = f$.

Our previous implementation of `FACT` in Example (3.8) was manual. This would be quite tedious for every recursive function. We can automate this with the following fixed-point combinator:

Definition 3.17: Y-Combinator

In lambda calculus, the **Y combinator** is a fixed-point combinator of form:

$$\mathtt{Y} \triangleq \lambda f.\, (\lambda x.\, f(x\ x))\, (\lambda x.\, f(x\ x))$$

E.g., $Y\ f = (\lambda x.\, f(x\ x))\, (\lambda x.\, f(x\ x)) = f((\lambda x.f(xx)(\lambda x.f(xx)))) = f(f(\dots)) = \dots$.

Example 3.9: Factorial with Y-Combinator

We can now define `FACT` using the Y combinator:

$$
\begin{aligned}
\mathtt{FACT} &\triangleq \lambda f.\lambda n.\mathtt{if}\ n = 0\ \mathtt{then}\ 1\ \mathtt{else}\ n * (f(n-1)) && (\text{ Definition of FACT' })\\
\mathtt{Y\ FACT}\ 3 &= ((\lambda x.\, \mathtt{FACT}(x\ x))\, (\lambda x.\, \mathtt{FACT}(x\ x)))\ 3 && (\ [\mathtt{FACT}/f]\mathtt{Y}\ )\\
&= \mathtt{FACT}\ ((\lambda x.\, \mathtt{FACT}(x\ x))\ (\lambda x.\, \mathtt{FACT}(x\ x)))\ 3 && (\ [\lambda x.\, \mathtt{FACT}(x\ x)/x]\mathtt{FACT}(x\ x)\ )
\end{aligned}
$$

Remember that `FACT` still requires two arguments f and n, for which we now supply:

$$
\begin{aligned}
&= \mathtt{if}\ 3 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 3 * ((\lambda x.\, \mathtt{FACT}(x\ x))\ (\lambda x.\, \mathtt{FACT}(x\ x))(3-1))\\
&= \mathtt{if}\ 3 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 3 * \mathtt{FACT}((\lambda x.\, \mathtt{FACT}(x\ x))\ (\lambda x.\, \mathtt{FACT}(x\ x)))\ (3-1)\\
&= \mathtt{if}\ 3 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 3 * (\mathtt{Y\ FACT}(3-1))\\
&\ \ \vdots\\
&= \mathtt{if}\ 2 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 2 * (\mathtt{Y\ FACT}(2-1))\\
&= \mathtt{if}\ 1 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 1 * (\mathtt{Y\ FACT}(1-1))\\
&= \mathtt{if}\ 0 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 0 * (\mathtt{Y\ FACT}(0-1))
\end{aligned}
$$

We hit the base-case and then evaluate $3 * (2 * (1 * 1))$ to get 6.   ■

If we aren't careful, step three of our derivation in Example (3.9) could lead to an infinite loop:

Definition 3.18: Strict vs. Lazy Evaluation

Strict evaluation means that all arguments to a function are evaluated before the function is applied, i.e., CBV (call-by-value).
Lazy evaluation means that an argument to a function is not evaluated until it is actually used in the body of the function, i.e., CBN (call-by-name).

Theorem 3.4: Y-Combinator & Lazy-Evaluation

The Y-combinator only works in lazy-evaluation settings. In strict evaluation, the Y-combinator will infinitely reduce.

To stop this we introduce another type of combinator for eager evaluation:

Definition 3.19: Z-Combinator

The Z-combinator is a fixed-point combinator that works in strict evaluation settings:

$$Z \triangleq \lambda f. \left(\lambda x. f\left(\lambda v. x\,x\,v\right)\right)\left(\lambda x. f\left(\lambda v. x\,x\,v\right)\right).$$

Example 3.10: Factorial with Z-Combinator (Part-1)

We now define the `FACT` using the Z combinator:

$$\texttt{Z FACT 3} = ((\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\,(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v)))\ 3 \qquad (\ [\texttt{FACT}/f]\texttt{Z}\ )$$
$$= \texttt{FACT}(\lambda v.(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v)\ v))\ 3 \quad (\ \text{Application}\ )$$

The extra argument waiting for a value delays Z long enough to evaluate `FACT`:

$$= \texttt{if}\ 3 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3*(\lambda v.(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ v)\ (3-1)))$$
$$= \texttt{if}\ 3 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3*(\lambda v.(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ v)\ (2)))$$
$$= \texttt{if}\ 3 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3*(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ 2$$
$$= \texttt{if}\ 3 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3*\texttt{FACT}(\lambda v.(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ v)\ 2$$
$$= \texttt{if}\ 3 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3*(\texttt{Z FACT 2})$$

This assumes that truthy `if` expressions don't evaluate the `else` branch. ∎

Last example touches on the idea of short-circuiting:

Definition 3.20: Short-Circuiting

Short-circuiting is a semantic trick which skips additional computation of a boolean expressions if some former part of the expression is sufficient to determine the value of the entire expression. For example ($\mathbb{B}$ is the set of booleans):

$$\frac{e_1 \Downarrow \bot}{e_1 \;\&\&\; e_2 \Downarrow \bot} \;(\text{andEvalFalse}) \qquad \frac{e_1 \Downarrow \top \quad e_2 \Downarrow v, \; v \in \mathbb{B}}{e_1 \;\&\&\; e_2 \Downarrow v} \;(\text{andEvalTrue})$$

$$\frac{e_1 \Downarrow \top}{e_1 \;||\; e_2 \Downarrow \top} \;(\text{orEvalTrue}) \qquad \frac{e_1 \Downarrow \bot \quad e_2 \Downarrow v, \; v \in \mathbb{B}}{e_1 \;||\; e_2 \Downarrow v} \;(\text{orEvalFalse})$$

$$\frac{e_1 \Downarrow \top \quad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \;(\text{ifTrueEval}) \qquad \frac{e_1 \Downarrow \bot \quad e_3 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \;(\text{ifFalseEval})$$

Notice that in (andEvalFalse), (orEvalTrue), and (ifTrueEval) the second expression is never evaluated. This is a form of **short-circuiting**.

4.3.4   Environments: Variable Binding Data-Structure

Though Y and Z combinators allow us to write recursive functions, this method quickly grows unwieldy as the complexity of our programs increases. Things like variable bindings and jumping between scopes become difficult to manage. That's where environments come in:

Definition 3.21: Environment

An **environment** is a data-structure that keeps track of **variable bindings**, i.e., associations between variables and their corresponding values. Environments are written as finite mappings:

$$\{x \mapsto v, \; y \mapsto w, \; z \mapsto f\}$$

Where each variable is mapped to a value. We may use such data-structure for state configurations. For example $\langle \{x \mapsto \lambda y.y\}, x \rangle \Downarrow \lambda y.y$. We shall denote environments as $\mathcal{E}$. Though we track state in configurations, the program itself is **still functionally pure**.

Theorem 3.5: Substitution vs. Environment Model

When we care about the speed of our program, the substitution model quickly becomes inefficient. This is because we have to read our entire program to find free variables and handle additional logic.

Here are the following operations we can preform on environments:

Definition 3.22: Operations on Environments

Environments support basic operations for managing variable bindings, similar to a map:

- $\varnothing$ — represents the empty environment (OCaml: `empty`).

- $\langle \mathcal{E} \rangle$ — represents the current environment (OCaml: `env`).

- $\langle \mathcal{E}[x \mapsto v] \rangle$ — adds a new binding of variable x to value v (OCaml: `add x v env`).

- $\langle \mathcal{E}(x) \rangle$ — looks up the value of variable x (OCaml: `find_opt x env`).

- $\langle \mathcal{E}(x) = \bot \rangle$ — indicates that x is unbound in the environment
  (OCaml: `find_opt x env = None`).

Additionally, if a new binding is added for a variable that already exists, the new binding **shadows** the old one:
$$\mathcal{E}[x \mapsto v][x \mapsto w] = \mathcal{E}[x \mapsto w]$$

Definition 3.23: Semantic Closures

A snapshot of the environment and its bindings is called a **closure**. There are two types:

$$\underbrace{(\![\mathcal{E}, \cdot \mapsto \lambda x.\, e]\!)}_{\text{unnamed}} \quad \text{and} \quad \underbrace{(\![\mathcal{E}, f \mapsto \lambda x.\, e]\!)}_{\text{named}}$$

Unnamed and named closures: The former captures the environment and program. The latter includes the function name for safe self-referencing. Closures are **values.**

Example 3.11: Extended Lambda Calculus (Part-1)

We momentarily create a grammar, **Lambda Calculus$^+$ (LC$^+$)**, for demonstration:

```
<expr> ::= <expr><expr>
         | let <var> = <expr> in <expr>
         | let rec <var> <var> = <expr> in <expr>
         | <val>
<var> ::= [a-zA-Z]
<val> ::= λ<var>.<expr> | <num>
```

■

Let's give $\mathbf{LC}^+$ some semantics:

Example 3.12: $\mathbf{LC}^+$ Semantics (Part-2)

Values and variables

$$\overline{\langle \mathcal{E}, \lambda x.\, e \rangle \Downarrow (\!(\mathcal{E}, \lambda x.\, e)\!)} \qquad \frac{\langle \mathcal{E}, x \rangle \neq \bot}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)} \qquad \overline{\langle \mathcal{E}, n \rangle \Downarrow n}$$

Let expressions

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \qquad \langle \mathcal{E}[\, x \mapsto v_1], \, e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{ let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2} \qquad \frac{\langle \mathcal{E}[\, f \mapsto (\!(\mathcal{E}', \, f \mapsto \lambda x.\, e_1)\!)\,], \, e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{ let rec } f\, x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

Application (unnamed closure)

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow (\!(\mathcal{E}', \cdot \mapsto \lambda x.\, e)\!) \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \qquad \langle \mathcal{E}'[x \mapsto v_2], \, e \rangle \Downarrow v}{\langle \mathcal{E}, \, e_1\, e_2 \rangle \Downarrow v}$$

Application (named closure)

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow (\!(\, \mathcal{E}', \, f \mapsto \lambda x.\, e)\!) \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \qquad \langle \mathcal{E}'[\, f \mapsto (\!(\mathcal{E}', \, f \mapsto \lambda x.\, e)\!)\,][x \mapsto v_2]\, e \rangle \Downarrow v}{\langle \mathcal{E}, \, e_1\, e_2 \rangle \Downarrow v}$$

∎

Example 3.13: Named Closure Derivation (Part-3)

Let $\alpha := f \mapsto (\!(\{x \mapsto 0\}, f \mapsto \lambda y.x)\!)$, and β define the following premises:

- $\langle \{x \mapsto 1, \alpha\}, f \rangle \Downarrow (\!(\{x \mapsto 0\}, f \mapsto \lambda y.x)\!) \to (\!(\{x \mapsto 0\}, f \mapsto \lambda y.0)\!)$

- $\langle \{x \mapsto 1, \alpha\}, 2 \rangle \Downarrow 2$

- $\langle \{x \mapsto 1, \alpha\}, [y \mapsto 2]0 \rangle \Downarrow 0$

We derive the following:

$$\frac{\langle \{x \mapsto 0\}, \lambda y.x \rangle \Downarrow (\!(\{x \mapsto 0\}, \lambda y.x)\!) \qquad \dfrac{\langle \{x \mapsto 0, \alpha\}, 1 \rangle \Downarrow 1 \qquad \dfrac{\beta}{\langle \{x \mapsto 1, \alpha\}, f\, 2 \rangle \Downarrow 0}\ (\text{NC})}{\langle \{x \mapsto 0, \alpha\}, \text{let } x = 1 \text{ in } f\, 2 \rangle \Downarrow 0}\ (\text{L})}{\langle \{x \mapsto 0\}, \text{ let } f = \lambda y.\, x \text{ in let } x = 1 \text{ in } f\, 2 \rangle \Downarrow 0}\ (\text{L})$$

Shorthanded rule names, NC:= Named Closure, L:= Let.   ∎

4.4 Type Theory

4.4.1 Simply Typed Lambda Calculus

An additional way to protect and reduce ambiguity in programming languages is to use **types**:

Definition 4.1: A Type

A **type** is a syntactic object that describes the kind of values that an expression pattern is allowed to take. This happens before evaluation to safeguard unintended behavior.

Recall our work in Section (1.4.1). We add the following:

Definition 4.2: Contexts & Typing Judgments

Contexts: Γ is a finite mapping of variables to types. **Typing Judgments:** $\Gamma \vdash e : \tau$, reads "e has type τ in context Γ". It is said that e is **well-typed** if $\cdot \vdash e : \tau$ for some τ, where $(\cdot)$ is the **empty context**. Such types we may inductively define:

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$
$$x ::= vars$$
$$\tau ::= types$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}$$

In practice, a context is a set (or ordered list) of variable declarations (variable-type pairs). Our inference rules operate with these contexts to determine the type of an expression:

This leads us to an extension of lambda calculus:

Definition 4.3: Simply Typed Lambda Calculus (STLC)

The syntax of the Simply Typed Lambda Calculus (STLC) extends the lambda calculus by including types and a unit expression.

```
<e>  ::=  () | <v> | <e> <e>
      |   fun "("<v> : <ty>")" -> <e>
<ty> ::=  unit | <ty> -> <ty>
<v>  ::=  [a-zA-Z]
```

We include the unit type (arbitrary value/void) and that functions are now typed. We transition into a more mathematical notation:

$$e ::= \quad \bullet \mid x \mid \lambda x^\tau . e \mid e\, e$$
$$\tau ::= \quad \top \mid \tau \rightarrow \tau$$
$$x ::= \quad variables$$

This brings us to the typing rules for STLC:

Definition 4.4: Typing Rules for STLC

Typing Rules: The typing rules for STLC are as follows:

$$\frac{}{\Gamma \vdash \bullet : \top} \text{ (unit)} \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x^\tau . e : \tau \to \tau'} \text{ (abstraction)}$$

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash x : \tau} \text{ (variable)} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'} \text{ (application)}$$

Such rules enforce that application is only valid when the e_1 position is a function type and the e_2 position is a valid argument type.

When encountering notation, types are often omitted in some contexts:

Definition 4.5: Church vs. Curry Typing

There are two main styles of typing:

Curry-style typing: Typing is **implied** (extrinsic) via typing judgement:

```
fun x -> x
```

Church-style typing: Types are **explicitly** (intrinsic) annotated in the expression:

```
fun (x : unit) -> x
```

Important: Curry-style does not imply polymorphism, expressions are judgement-backed.

This leads us the an important lemma:

Definition 4.6: Lemma – Uniqueness of Types

Let Γ be a typing context and e a well-formed expression in STLC:

$$\text{If } \Gamma \vdash e : \tau_1 \text{ and } \Gamma \vdash e : \tau_2, \text{ then } \tau_1 = \tau_2.$$

I.e., typing in STLC is **deterministic** – a well-typed expression has a **unique type** under any fixed context.

To prove the above lemma we must recall structural induction:

Definition 4.7: Structural Induction

Structural induction is a proof technique used to prove properties of recursively defined structures. It consists of two parts:

- **Base case:** Prove the property for the simplest constructor (e.g., a variable or unit).

- **Inductive step:** Assume the property holds for immediate substructures, and prove it holds for the structure built from them.

This differs from standard mathematical induction over natural numbers, where the base case is typically $n = 0$ (or 1), and the inductive step proves $(n + 1)$ assuming (n).

In the context of **lambda calculus**, expressions are recursively defined and built from smaller expressions. Structural induction proceeds as:

- **Base case:** Prove the property for the simplest expressions (e.g., variables and units).

- **Inductive step:** Assume the property holds for sub-expressions $e_1, e_2, \ldots, e_k$, and prove it holds for a compound expression e (e.g., abstractions and application).

A proof of the previous lemma is as follows:

Proof 4.1: Lemma – Uniqueness of Types

We prove that if $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$ (Γ is a fixed typing context, and e a well-formed expression) via structural-induction:

Case 1: $e = \bullet$ (unit value). We define a generation lemma (*****): if $\Gamma \vdash \bullet : \tau \to \tau = \top$. Therefore, if $\Gamma \vdash \bullet : \tau_1$ and $\Gamma \vdash \bullet : \tau_2$, then $\tau_1 = \tau_2 = \top$, by lemma (*****).

Case 2: $e = x$ (a variable). We define a generation lemma (******): $\Gamma, x \vdash \tau \to x : \tau \in \Gamma$ Since $(x : \tau_1), (x : \tau_2) \in \Gamma$ and Γ is fixed, x maps to a single type. Thus, $\tau_1 = \tau_2$, by lemma (******).

Case 3: $e = \lambda x^\tau.e'$. Abstraction typings require the form $\tau \to \tau'$. Both derivations, $\Gamma \vdash \lambda x^\tau.e' : \tau_1$ and $\Gamma \vdash \lambda x^\tau.e' : \tau_2$, must have such form. Hence, by inductive hypothesis on e' (under $\Gamma, x : \tau$), we conclude $\tau_1 = \tau_2$.

Case 4: $e = e_1 e_2$ (application). Suppose $\Gamma \vdash e_1 e_2 : \tau_1$ and $\Gamma \vdash e_1 e_2 : \tau_2$. Then e_1 must have type $\tau' \to \tau_1$ and $\tau' \to \tau_2$ respectively, and e_2 type τ'. With likewise reasoning from Case 3, and through the inductive hypothesis on e_1 and e_2, we conclude $\tau_1 = \tau_2$.

Hence, by induction on the typing derivation, the type assigned to any expression is unique.
∎

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] Cornell University. Fixed-point combinators. Lecture Notes, 2020.

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