Functional Programming Language Design

Christian J. Rudder

January 2025

Contents

Contents **1**

1 Functional Programming **6**
1.1 Introduction . . . 6
1.2 Ocaml Basics: Syntax, Types, and Semantics . . . 11
Understanding Functions in OCaml . . . 14
If-Expressions . . . 19
Type Hinting in OCaml . . . 20
OCaml Data Structures: Arrays, Lists, and Tuples . . . 21
Pattern Matching & Switch-Case Absence . . . 24
Looping: Recursion, Tail-End Recursion . . . 25
Strings, Characters, and Printing in OCaml . . . 27
Conversions in OCaml . . . 30
Defining Custom Types: Variants and Records . . . 31
Print Debugging . . . 34
1.3 Formalizing Ocaml Expressions . . . 35

Bibliography **41**

This page is left intentionally blank.

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

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

Prerequisite Definitions

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

Definition 0.1: Token

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

Examples of tokens include:

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

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

Definition 0.2: Non-terminal and Terminal Symbols

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

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

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

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

Definition 0.3: Symbol ":="

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

For example:

$$x := 5$$

This means the variable x is assigned the value 5.

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

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

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

Definition 0.4: Substitution: $[v/x]e$

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

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

This means that every occurrence of x in e is replaced with v. We may string multiple substitutions together, such as:

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

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

— 1 —

Functional Programming

1.1 Introduction

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

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

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

Definition 1.1: Well-formed Expression

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

Definition 1.2: Programming Language

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

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

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

Example 1.1: Syntax for Addition

If e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1 + e_2$ is also a well-formed expression. ■

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

Definition 1.3: Production Rule (::=)

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

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

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

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

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

Example 1.2: Production Rule

Here are some possible production rules:

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

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

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

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

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

Example 1.3: Production Rule for Addition

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

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

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

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

Definition 1.4: Meta-variables

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

Example 1.4: Meta-variables:

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

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

Definition 1.5: Judgments

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

Definition 1.6: Context and Typing Environment

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

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

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

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

where f is a function taking n arguments $(\tau_1, \tau_2, \ldots, \tau_n)$, outputting type τ. We may add temporary variables to the context. For example:

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

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

Definition 1.7: Rule of Inference

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

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

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

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

Example 1.5: Typing Rule for Integer Addition

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

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

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

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

Example 1.6: Typing Rule for Function Application

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

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

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

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

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

Example 1.7: Evaluation Rule for Integer Addition (Semantics)

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

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

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 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 1.1: Example of Strong Typing

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

This will result in the following error:

Listing 1.2: Error Message

```
3 | let _ = x + y (* THIS IS NOT POSSIBLE *)
        ^

Error: This expression has type string but an expression was expected of
    type int
```

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

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

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

Table 1.1: Basic OCaml Types, Literals, and Operators

For emphasis:

Definition 2.1: 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 2.2: OCaml (in) Keyword

Consider the expression below:

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

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

This is similar to the prerequisite definition of the substitution operation (0.4). Mathematically, we can think of this as:

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

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

To illustrate this, Observe the diagram below:

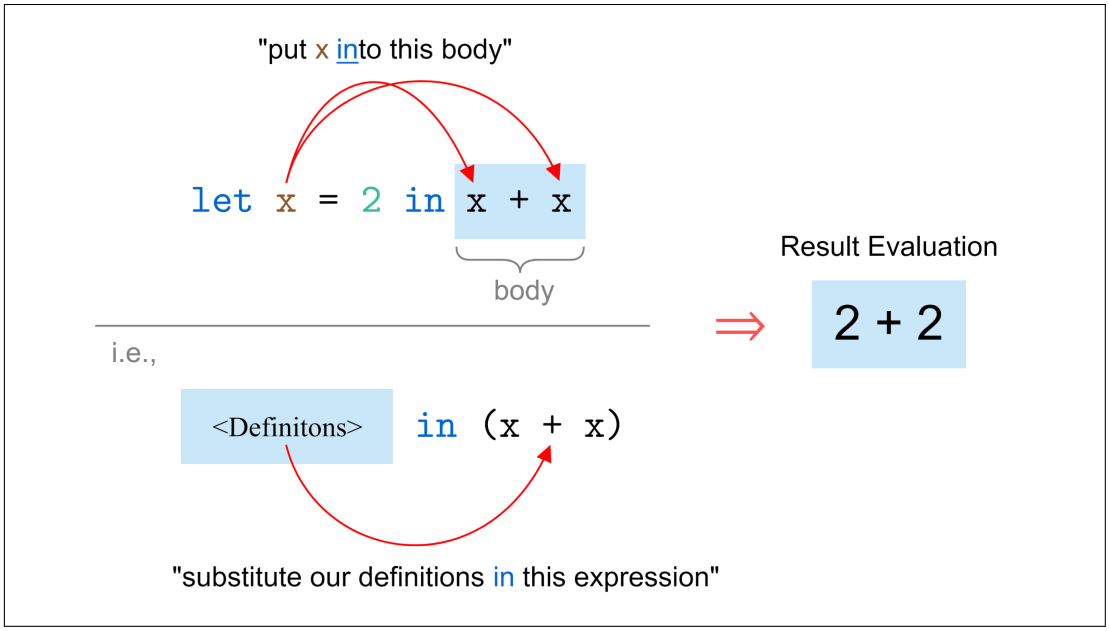

Figure 1.1: The `in` Keyword in OCaml

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

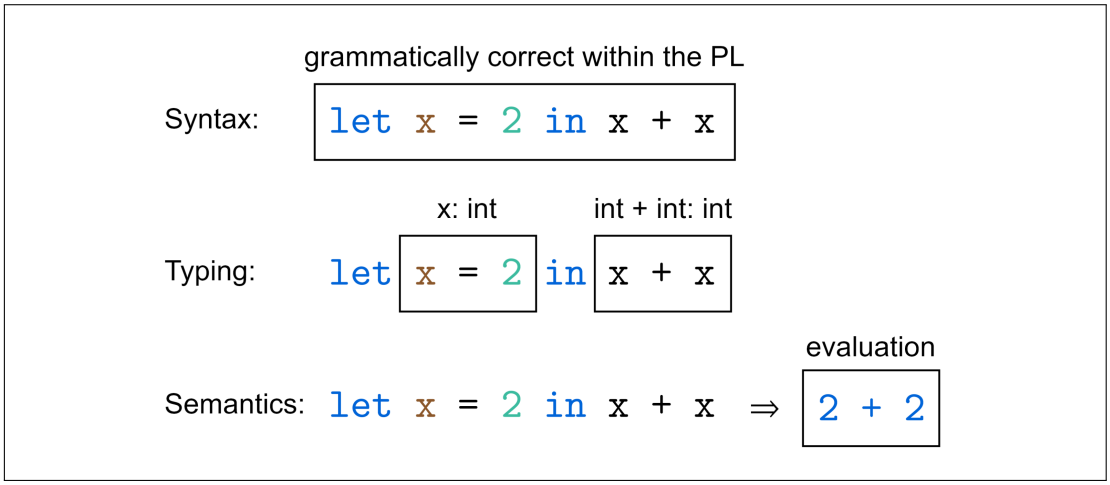

Figure 1.2: Syntax, Semantics, and Types in OCaml

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

Definition 2.3: 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 1.3: Whitespace Agnostic Example

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

and

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

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

Understanding Functions in OCaml

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

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

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

Definition 2.4: 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:

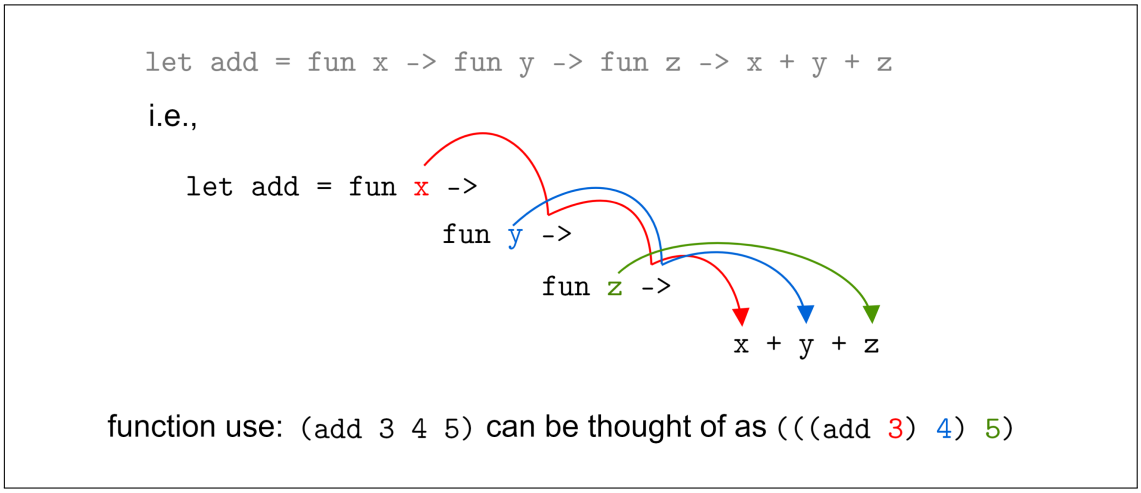

Figure 1.3: Anonymous Functions in OCaml

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

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

Figure 1.4: Nested Functions Pseudo-code Example

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

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

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

Definition 2.5: 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 2.6: 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 2.7: 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 1.4: 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 1.5: 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 2.8: 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 1.6: Unit Type and Value

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

Listing 1.7: 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 2.9: 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 1.8: 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 1.9: Using `let ()` in the Main Function

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

Definition 2.10: Skeleton Code

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

Listing 1.10: Skeleton Code Example

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

Definition 2.11: Multiple Function Arguments

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

Listing 1.11: 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 1.12: Correct Function Application

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

If-Expressions

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

Definition 2.12: 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 1.13: 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 1.14: 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 1.15: 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 2.13: Conditional Assignment

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

Listing 1.16: 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 *)
```

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 2.14: 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 1.17: 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 1.18: Adding Type Annotations to Variables

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

Listing 1.19: 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 1.20: Type Hinting in Conditional Expressions

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

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 2.15: 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 1.21: 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 1.22: 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 1.23: 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 1.24: 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 2.16: 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 1.25: Defining and Modifying an Array

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

Listing 1.26: 2D Array

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

Listing 1.27: Arrays.make: Prefill Length Array

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

Listing 1.28: Accessing Array Elements

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

Listing 1.29: 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 1.30: Mutating Array Elements

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

Listing 1.31: Length of Array

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

Definition 2.17: 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 1.32: Defining a Tuple

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

Listing 1.33: 2D Tuple

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

Listing 1.34: Mixed Type Tuple

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

Listing 1.35: 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 1.36: 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.

Pattern Matching & Switch-Case Absence

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

Definition 2.18: 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 1.37: 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 1.38: Matching an Integer

```
fun x ->
  match x with
  | 0 -> "zero"
  | 1 -> "one"
  | _ -> "other";;

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

Listing 1.39: 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.

Looping: Recursion, Tail-End Recursion

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 2.19: 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 1.40: Summing to n Using Recursion

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

Listing 1.41: Fibonacci Sequence Using Recursion

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

Listing 1.42: Sum an Array of Integers Using Recursion

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

Definition 2.20: 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 1.43: 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 1.44: 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 1.45: 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 1.46: 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. *)
```

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 2.21: Strings in OCaml

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

Listing 1.47: Creating and Concatenating Strings

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

Listing 1.48: Getting the Length of a String

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

Listing 1.49: Accessing a Character in a String

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

Listing 1.50: Slicing a String

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

Listing 1.51: Converting to Uppercase

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

Listing 1.52: String Equality

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

Definition 2.22: 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 1.53: Defining Characters

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

Listing 1.54: Comparing Characters

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

Listing 1.55: Converting Characters to Strings

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

Listing 1.56: Converting Characters to Integers

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

Listing 1.57: Converting Integers to Characters

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

Listing 1.58: Checking if a Character is Uppercase

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

Definition 2.23: Printing in OCaml

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

Listing 1.59: Printing a String

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

Listing 1.60: 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 1.61: 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 1.62: 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 1.63: 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 1.64: Using Printf.eprintf for Error Messages

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

Conversions in OCaml

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

Definition 2.24: Common Conversions in OCaml

Listing 1.65: Integer and Float Conversions

```
float_of_int : int -> float

int_of_float : float -> int
```

Listing 1.66: Character and Integer Conversions

```
int_of_char : char -> int

char_of_int : int -> char
```

Listing 1.67: Boolean and String Conversions

```
string_of_bool : bool -> string

bool_of_string : string -> bool

bool_of_string_opt : string -> bool option
```

Listing 1.68: String and Integer Conversions

```
string_of_int : int -> string

int_of_string : string -> int

int_of_string_opt : string -> int option
```

Listing 1.69: String and Float Conversions

```
string_of_float : float -> string

float_of_string : string -> float

float_of_string_opt : string -> float option
```

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 2.25: 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 1.70: Defining and Using Variants Correctly

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

(* 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 2.26: 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 1.71: 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 1.72: 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 1.73: 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 2.27: Variant (of) Keyword

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

Listing 1.74: Variants Carrying Record Fields

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

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

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

Definition 2.28: 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 1.75: Using Variants for Type Checking

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

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

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

Print Debugging

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

Listing 1.76: Print Debugging

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

Without the `stdlib` we can do the following:

Listing 1.77: Print Debugging Without Printf

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

Perhaps a quick print without care of formatting:

Listing 1.78: Dirty Print Debugging

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

Section Exercises:

Exercise 2.1: 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 2.2: Write an OCaml function that takes an integer x and evaluates to the first digit of x using only integer arithmetic operations.

Exercise 2.3: 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 2.4: Write an OCaml function implementation of the Fibonacci sequence using tail-end recursion.

1.3 Formalizing 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 3.1: 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 3.2: 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:** $$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau}$$

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

  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.

Thus, we have formalized the `let` expression in OCaml.

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

Definition 3.3: 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 3.4: Formalizing If-Expressions

Let Γ be the OCaml context, then:

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

  If e_1 is a well-formed expression, e_2 is a well-formed expression, and e_3 is a well-formed expression, then $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ 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 $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ 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 $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ 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 $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ 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 3.5: 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 3.6: 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.3.5) 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 3.7: Formalizing Tuples

Let Γ be the OCaml context, then:

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

  If e_1 is a well-formed expression and e_2 is a well-formed expression, then (e_1, e_2) is also a well-formed expression.

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

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

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

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

Definition 3.8: Formalizing Lists

Let Γ be the OCaml context, then:

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

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

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

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

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

  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 3.9: 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 3.10: 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 3.11: Weak Matching on Lists

Let Γ be the OCaml context, then:

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

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

- **Typing Rule:**

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

  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:**

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

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

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

  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.

Bibliography

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

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

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