Programming with OCaml

Christian Rinderknecht

Christian.Rinderknecht@nomadic-labs.com

Nomadic Labs (Paris)

20 October 2018

Tezos Masterclass

Introduction

- Rewrite systems are a good mathematical and mental model for understanding functional languages.
- It is time now to learn an actual functional language, so we can run programs.
- The aim of these lectures is to introduce the functional language OCaml, which is used for the core development at **Tezos** (network protocols, compilers and virtual machines for smart contracts).
- We are going to start with a mini language that is actually a subset of OCaml, then grow it step by step.

Growing a language

- The presentation is principled, that is, it relies on a systematic classification of the **syntactic constructs** and their **formal semantics**.
- When we reach a certain complexity, we will drop the formal descriptions.
- Both theoretical and pragmatic, our approach is complementary to those usually found in books.
- Let mini-ML be the name of the subset of OCaml we are going to grow.

Appending a stack

- The notation we introduced for stacks in the context of rewrite systems actually follows the OCaml convention.
- In the context of functional programming, stacks are often called **lists**.
- Here are different ways to express the same list in OCaml:
$$1::(2::3::(4::[])) = 1::2::3::4::[] = [1;2;3;4].$$
- Before starting, let us see how a familiar function defined with a rewrite system looks like in OCaml.
- Let us recall the appending of a stack:
$$\mathsf{cat}([],t) \xrightarrow{\alpha} t$$
$$\mathsf{cat}(x::s,t) \xrightarrow{\beta} x::\mathsf{cat}(s,t)$$

Appending a stack (continued)

- In OCaml, the function name is not repeated (once for each rule).
- Instead, it is introduced once by a **binder** called **`let`**, like so:
  `let` `cat =` **`function`** `...`
- The complete definition in OCaml is:

```
let rec cat = function
    [], t → t
  | x::s, t → x::cat(s,t)
```

- Note the addition of the keyword **`rec`**: this is meant so the call $\texttt{cat}(s,t)$ indeed refers to the function being defined.
- With rewrite systems, recursion is not an issue: any call matching a pattern can be rewritten.
- Note also how the two rules are introduced by the keyword **`function`** and are separated by a vertical line.

Sentences

- An OCaml program is a sequence of sentences.
- A **sentence**, or **global definition**, is defined by the following cases, where e denotes an expression, x is a variable, and **let** and **rec** are keywords:
  - *global definition* **let** $x = e$
  - *global recursive definition* **let rec** $x = e$
- Global definitions can be optionally followed by two semicolons: these are necessary when inputting the sentences in the toplevel loop (prompted after running the command `ocaml` from a Unix shell).
- Note that the keyword **rec** is necessary to enable recursion. (We will see later how that works precisely.)
- For e to be an expression means that e denotes a part of a sentence which is classified as an expression according to its **syntax**.

- We say that *e* is a **metavariable** because, being a name, it is a variable, but that variable does not belong to the language being described (the **object language**, here, OCaml) and, instead, exists only in the descriptive language (the **metalanguage**).
- In the following

  $$\textbf{let } x = e$$

  the metavariable *x* (mind the italics) denotes an infinite set of OCaml variables and we must not confuse it with the OCaml variable x (no italics). Similarly, the metavariable *e* denotes an infinity of expressions, as in the following:

```
let e = 3
let x = e
let a = x + e
```

Expressions

- An expression *e* is **recursively defined by** the following **cases**:
  - *variable* — my_var, x, y, z, etc.
  - *unit or integer constant* — () or 0 or 1 or 2, etc.
  - *function* (or *abstraction*) — **fun** $x \rightarrow e$
  - *call* (or *application*) — $e_1\ e_2$
  - *arithmetic operator* — (+) (-) (/) (*)
  - *arithmetic operation* — e_1 + e_2 or e_1 - e_2, etc.
  - *parenthesis* — (e)
  - *local definition* — **let** $x = e_1$ **in** e_2
- Note how the keywords **rec** and **function**, as well as the definition of multiple cases (as seen in the definition of "cat" earlier) is not found above: this is because we want to focus on other aspects of OCaml first, like higher-order functions.
- Instead we have **fun**, which enables only one case made of a variable.

Examples of expressions

- Note that what we print nicely as "$\rightarrow$" is actually written "`->`" in the source code.
- Here are examples of OCaml sentences making up a program, step by step.
- First, a numerical constant:
  `let x = 0`
- Note that constants can be defined by means of a rewrite system as functions taking the empty tuple:
  $$x() \rightarrow 0$$
- The next sentence is
  `let id = fun x -> x`
- This is equivalent to the rewrite system
  $$\mathrm{id}(x) \rightarrow x$$

Examples of expressions (continued)

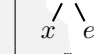
- Next we have another constant being defined by binding a variable to the value of a call:
  `let y = 2 in id y`
  This is equivalent to the call `id(2)`.
- Now
  `let x = (fun x → fun y → x + y) 1 2`
  is equivalent to
  `let x = 1 + 2`
  (We will see why later.)
- Finally
  `let z = x + 1`
  is obvious.

- Variables must start with a lowercase character.
- The arrow is right-associative, so the expression
$$\textbf{fun}\ x_1 \to \textbf{fun}\ x_2 \to \ldots \to \textbf{fun}\ x_n \to e$$
is equivalent to $\textbf{fun}\ x_1 \to (\textbf{fun}\ x_2 \to (\ldots \to (\textbf{fun}\ x_n \to e))\ \ldots)$.
- Function calls are left-associative, so
$$e_1\ e_2\ e_3\ \ldots\ e_n = (((\ldots(e_1\ e_2)\ e_3)\ldots)\ e_n).$$
- Function calls have higher priority than operator calls, for example,
$$f\,3 + 4 = (f\,3) + 4 = f(3) + 4.$$
- Operator calls have higher syntactic priority than abstractions, for instance,
$$\textbf{fun}\ x \to x + y = \textbf{fun}\ x \to (x + y).$$

Function calls

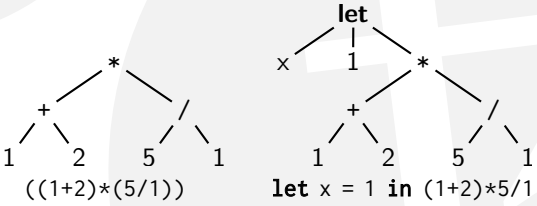
- Note the difference between `f x y` and `f(x,y)`.
- The former is to be understood as `(f(x))(y)` which means that there are *two* calls:
  1. First, the call `f(x)` is evaluated. Let us assume that its value is g.
  2. Second, the call `g(y)` is evaluated.
- In other words, the first case is equivalent to
  `let` `g = f x` **`in`** `g y`

- A graphical representation of programs (here, expressions) as trees proves very handy when studying their semantics (meaning):

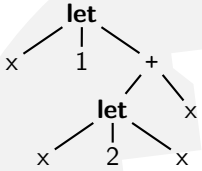
Expression	Tree
x	x
fun $x \rightarrow e$	
$e_1\ e_2$	

- Note that we needed some symbol for the root of the application tree (`$`). Traditionally, that symbol is (`@`), but OCaml uses it already for something else.
- When trees represent programs, they are called **abstract syntax trees** (AST).

- Intuitively, the method for constructing abstract syntax trees consists in fully parenthesising the expression.
- Each parenthesis corresponds to a subexpression and each subexpression corresponds to a subtree.
- The tree is built from the root, down to the leaves, by traversing subexpressions from the outermost to the innermost, that is, the most embedded ones.
- For instance, consider



- Another example:
  `let x = 1 in ((let x = 2 in x) + x)`
  corresponds to the abstract syntax tree



- This example features two local definitions of "x": is one the redefinition of the other? Is one hiding the other?
- There are also two uses of "x": which use corresponds to which definition?

- To answer theses questions, we must first understand
  - what a **binding** is,
  - what the **scope** of a binding is, and
  - what an **environment** is.
- The expression we called **local definition** earlier:

  $$\textbf{let } x = e_1 \textbf{ in } e_2$$

  is evaluated as follows:
  1. The expression e_1 is evaluated,
  2. assuming it has a value v_1, then v_1 is **bound** to the variable x,
  3. the expression e_2 is evaluated under the assumption that x is bound to v_1.

- A local definition

$$\textbf{let } x = e_1 \textbf{ in } e_2$$

  associates the value v_1 of expression e_1 to the variable denoted by x, making up a **binding** noted $x \mapsto v_1$, and then proceeds to evaluate e_2 with this binding available.
- The binding $x \mapsto v_1$ is not usable outside the evaluation of e_2.
- We say that the **scope** of the definition is made of the subtrees of the abstract syntax tree corresponding to e_2 where the binding of that definition is usable ("visible").
- A mapping from variables to values is called an **environment**.
- Let us note $\rho_\varnothing$ the empty environment.
- We write

$$x \mapsto v \oplus \rho$$

  to mean that the binding $(x \mapsto v)$ was added to the environment ρ.

- Consider again the program

```
let x = 0 in
let id = fun x → x in
let y = id x in
let x = (fun x → fun y → x + y) 1 2
in x + 1
```

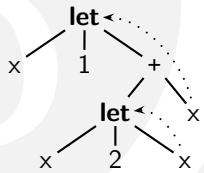
- The variable x in the expression x+1 denotes the value of the expression bound to the variable x in the previous line, not the first.
- Bindings are ordered in the environment **by the order of their definitions**.
- Let us follow the evolution of the environment.

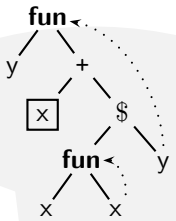
1. The environment is initially empty: $\rho_\varnothing$;
2. **let** x = 0 **in** …
   yields $(\mathsf{x} \mapsto 0) \oplus \rho_\varnothing$;
3. **let** id = **fun** x → x **in** …
   yields $(\mathsf{id} \mapsto \textbf{fun}\ \mathsf{x} \to \mathsf{x}) \oplus (\mathsf{x} \mapsto 0) \oplus \rho_\varnothing$;
4. **let** y = id x **in** …
   yields $(\mathsf{y} \mapsto 0) \oplus (\mathsf{id} \mapsto \textbf{fun}\ \mathsf{x} \to \mathsf{x}) \oplus (\mathsf{x} \mapsto 0) \oplus \rho_\varnothing$;
5. **let** x = …
   yields $(\underline{\mathsf{x} \mapsto 3}) \oplus (\mathsf{y} \mapsto 0) \oplus (\mathsf{id} \mapsto \textbf{fun}\ \mathsf{x} \to \mathsf{x}) \oplus (\mathsf{x} \mapsto 0) \oplus \rho_\varnothing$.
6. The expression x+1 is evaluated in $3 + 1 = 4$ because $\mathsf{x} \mapsto 3$ (the last binding) **hides** $\mathsf{x} \mapsto 0$ (the first binding).
7. In other words, the first definition of x was **out of scope** in x+1.

Free variables

- By definition, a variable is **free** in an expression if it is not bound by a **let** or a **fun**.
- We can understand this concept on a graphic representation of the abstract syntax tree.
- From each variable occurrence, we move up, towards the root.
- If we encounter a **let** that binds that variable (in the left child), we create an oriented edge from the variable to that **let**, and we stop: the variable is bound.
- Otherwise, if we reach the root (no such **let** has been found), then the variable is free.

Free variables





`let x = 1 in ((let x = 2 in x) + x)` `fun y → x + (fun x → x) y`

- A similar situation arises with functions: **fun** is a binder, just like **let**.
- In **fun** $x \rightarrow e$, the variable x (called a **parameter**) may shadow (hide) another variable x defined on the path to the root.
- See above for an example.
- We note $\mathcal{F}(e)$ the set of the variables free in e.

- It is easy to formally define $\mathcal{F}$ by case and recursively:

$$\mathcal{F}(x) = \{x\}, \quad \texttt{where } x \texttt{ is a variable}$$
$$\mathcal{F}(c) = \varnothing, \quad \texttt{where } c \texttt{ is a constant}$$
$$\mathcal{F}(\textbf{fun } x \to e) = \mathcal{F}(e) \setminus \{x\}$$
$$\mathcal{F}(e_1\ e_2) = \mathcal{F}(e_1) \cup \mathcal{F}(e_2)$$
$$\mathcal{F}(\textbf{let } x = e_1 \textbf{ in } e_2) = \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \setminus \{x\}$$
$$\mathcal{F}(e_1 + e_2) = \mathcal{F}(e_1) \cup \mathcal{F}(e_2)$$

- Now, instead of drawing abstract syntax trees, let us use these equations to compute the free variables of our previous examples.

$$
\begin{aligned}
&\mathcal{F}(\textbf{let}\ \mathsf{x} = 1\ \textbf{in}\ (\textbf{let}\ \mathsf{x} = 2\ \textbf{in}\ \mathsf{x}) + \mathsf{x}) \\
&\quad = \quad \mathcal{F}(1) \cup \mathcal{F}((\textbf{let}\ \mathsf{x} = 2\ \textbf{in}\ \mathsf{x}) + \mathsf{x}) \backslash \{\mathsf{x}\} \\
&\quad = \quad \varnothing \cup (\mathcal{F}(\textbf{let}\ \mathsf{x} = 2\ \textbf{in}\ \mathsf{x}) \cup \mathcal{F}(\mathsf{x})) \backslash \{\mathsf{x}\} \\
&\quad = \quad (\mathcal{F}(2) \cup \mathcal{F}(\mathsf{x}) \backslash \{\mathsf{x}\} \cup \mathcal{F}(\mathsf{x})) \backslash \{\mathsf{x}\} \\
&\quad = \quad \mathcal{F}(\mathsf{x}) \backslash \{\mathsf{x}\} \\
&\quad = \quad \varnothing.
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{F}(\textbf{fun}\ \mathsf{y} \rightarrow \mathsf{x} + (\textbf{fun}\ \mathsf{x} \rightarrow \mathsf{x})\ \mathsf{y}) \\
&\quad = \quad \mathcal{F}(\mathsf{x} + (\textbf{fun}\ \mathsf{x} \rightarrow \mathsf{x})\ \mathsf{y}) \backslash \{\mathsf{y}\} \\
&\quad = \quad (\mathcal{F}(\mathsf{x}) \cup \mathcal{F}((\textbf{fun}\ \mathsf{x} \rightarrow \mathsf{x})\ \mathsf{y})) \backslash \{\mathsf{y}\} \\
&\quad = \quad (\{\mathsf{x}\} \cup \mathcal{F}(\textbf{fun}\ \mathsf{x} \rightarrow \mathsf{x}) \cup \mathcal{F}(\mathsf{y})) \backslash \{\mathsf{y}\} \\
&\quad = \quad (\{\mathsf{x}\} \cup \mathcal{F}(\mathsf{x}) \backslash \{\mathsf{x}\} \cup \{\mathsf{y}\}) \backslash \{\mathsf{y}\} \\
&\quad = \quad \{\mathsf{x},\mathsf{y}\} \backslash \{\mathsf{y}\} \\
&\quad = \quad \{\mathsf{x}\}.
\end{aligned}
$$

Closed expressions

- A **closed** expression is an expression without free variables.
- That is, an expression e is closed if, and only if, $\mathcal{F}(e) = \varnothing$.
- Only a closed expression can be evaluated (executed).
- By extension, we say that e is **closed in an environment** ρ if $\mathcal{F}(e) \subseteq \text{dom } \rho$ (the free variables are bound in ρ).
- That is why the first static analysis performed by compilers consists in determining the variables which are free in expressions: if the program is not closed, it is rejected.
- In the case of `fun y -> x + (fun x -> x) y`, the OCaml compiler prints

  ```
  Unbound value x
  ```

  and stops. It is useful that this open expression be rejected at compile-time and does not cause a run-time error.

Evaluation

- A **semantics** is a partial function from expressions to values. An **interpreter** or **evaluator** is a program implementing a semantics.
- A partial function models the fact that an evaluation may not terminate or end in an error.
- The **values** of mini-ML are defined recursively by the following cases:
  - *unit or integer constant* $()$ or 0 or 1 or 2, etc.
  - *closure* $((\textbf{fun}\ x \rightarrow e), \rho)$, where ρ is an environment.
  - *n-tuple* $v_1, \ldots, v_n$
- A **closure** is a pair made of a function and an environment.
- That environment is the one at the location where the function is defined and the function must be closed in that environment (this is called **static scoping**, or **lexical scoping**).

Evaluation

- Let $\rho(x)$ be the value most recently bound to x in the environment ρ, if any.
- We have $(x \mapsto v \oplus \rho)(x) = v$.
- We equivalently say that the environment ρ was **extended** with the binding $x \mapsto v$.
- The evaluation of an expression must take place in the context of an environment, where any free variable must be bound (in other words, the expression must be closed in that environment).
- For example, when evaluating the expression x in
  `let x = 1 in x`
  we need the previous environment extended with the binding $x \mapsto 1$.

Evaluation

Let us proceed to define a semantics for our mini-ML, by associating to each expression e its value v, assuming the environment ρ:

• x	Look up the **last** value bound to x in ρ.
• **fun** $x \to e$	The value is $((\textbf{fun}\ x \to e), \rho)$.
• `+ - / *`	The value is $((+), \rho)$, etc.
• e_1 + e_2, etc.	Evaluate e_1 and e_2 in ρ, then add, etc.
• `()` or `0` or `1` or `2`, etc.	The value is `()` or `0` or `1`, etc.
• (e)	Evaluate e into v in ρ.
• **let** $x = e_1$ **in** e_2	Evaluate e_1 into v_1 in ρ; evaluate e_2 into v in $x \mapsto v_1 \oplus \rho$.
• $e_1\ e_2$	Evaluate e_1 and e_2 into v_1 and v_2 in ρ (order left unspecified); v_1 must be of the form $((\textbf{fun}\ x \to e), \rho_1)$; evaluate e in $x \mapsto v_2 \oplus \rho_1$: the value is v.

Example of evaluation

1. The environment is initially empty. Let us evaluate
   ```
   let x = 0 in
   let id = fun x → x in
   let y = id x in
   let x = (fun x → fun y → x + y) 1 2
   in x + 1
   ```
2. The evaluation of `let x = 0 in` … starts with the evaluation of `0`, which, obviously, yields 0.
3. We create the binding $\mathtt{x} \mapsto 0$, with which we extend $\rho_\varnothing$, that is, $\mathtt{x} \mapsto 0 \oplus \rho_\varnothing$.
4. We evaluate the elided subexpression within this new environment.

Example of evaluation

- The evaluation of **let** `id = fun x` $\rightarrow$ `x in` ... is performed within the environment $\mathtt{x} \mapsto 0 \oplus \rho_\varnothing$.
- The value v of `id` is then the closure $((\textbf{fun}\ \mathtt{x} \rightarrow \mathtt{x}), \mathtt{x} \mapsto 0 \oplus \rho_\varnothing)$.
- We extend the current environment with $\mathtt{id} \mapsto v$ and we evaluate the elided subexpression with it.
- The evaluation of **let** `y = id x in` ... is done within the environment $\rho = (\mathtt{id} \mapsto ((\textbf{fun}\ \mathtt{x} \rightarrow \mathtt{x}), \mathtt{x} \mapsto 0 \oplus \rho_\varnothing)) \oplus (\mathtt{x} \mapsto 0) \oplus \rho_\varnothing$.
- We evaluate first (`id x`) in ρ.
- In order to do so, we evaluate `id` and `x` separately.
- These are both variables, thus we look them up in the environment to retrieve their last corresponding binding: $\rho(\mathtt{id}) = ((\textbf{fun}\ \mathtt{x} \rightarrow \mathtt{x}), \mathtt{x} \mapsto 0 \oplus \rho_\varnothing)$, and $\rho(\mathtt{x}) = 0$.
- We need to evaluate the body `x` in the environment $\rho = (\mathtt{x} \mapsto 0) \oplus (\mathtt{x} \mapsto 0) \oplus \rho_\varnothing$, which yields $\rho(\mathtt{x}) = 0$, etc.

- If we note $[\![e]\!]\rho$ the value obtained by evaluating the expression e in the environment ρ, then the evaluation of mini-ML is:

$$[\![\overline{n}]\!]\rho = n, \text{ with } \overline{n} \text{ an OCaml integer and } n \in \mathbb{N}$$
$$[\![e_1 + e_2]\!]\rho = [\![e_1]\!]\rho + [\![e_2]\!]\rho \in \mathbb{N}$$
$$[\![(e)]\!]\rho = [\![e]\!]\rho$$
$$[\![x]\!]\rho = \rho(x)$$
$$[\![\textbf{fun } x \to e]\!]\rho = (\textbf{fun } x \to e), \rho$$
$$[\![e_1\ e_2]\!]\rho = [\![e_3]\!](x \mapsto [\![e_2]\!]\rho \oplus \rho_3), \text{ where } [\![e_1]\!]\rho = (\textbf{fun } x \to e_3), \rho_3$$
$$[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]\rho = [\![(\textbf{fun } x \to e_2)\ e_1]\!]\rho.$$

- Note that we write "x,y" instead of "(x,y)" for brevity.
- Evaluation means applying the equations from left to right.

Partial applications

- We can easily work out that
$$[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]\rho = [\![e_2]\!](x \mapsto [\![e_1]\!]\rho \oplus \rho).$$
- Since closures are values, they can be the value of a function call:

```
let add = fun x → fun y → x + y in
let incr = add 1
in incr 5
```

- The call (add 1) is a **partial application**, as opposed to a **complete application** like (add 1 5).
- A partial application, by definition, always evaluates in a closure.
- Operations are functions denoted by a symbol used in infix position (between the operands). As such, they can be partially evaluated:

```
let incr = (+) 1 in incr 5
```

where the parentheses in `(+)` mean that the operator has to be considered in **prefix** position.

- Our mini-ML is **Turing-complete**, just as OCaml is.
- For instance, we can write the following non-terminating program:

  `let omega = fun f -> f f in omega omega`
- The function `omega` is an **auto-applicative function**.
- It enables the following non-terminating evaluation:

$$\begin{aligned}
&[\![\texttt{let omega = fun f} \to \texttt{f f in omega omega}]\!]\rho \\
&\quad = [\![\texttt{omega omega}]\!](\texttt{omega} \mapsto [\![\texttt{fun f} \to \texttt{f f}]\!]\rho \oplus \rho) \\
&\quad = [\![\texttt{f f}]\!](\texttt{f} \mapsto ((\texttt{fun f} \to \texttt{f f}), \rho) \oplus \rho) \\
&\quad = \textit{idem}.
\end{aligned}$$

- To demonstrate the expressive power of our mini-ML, let us see how we can simulate recursive functions by means of the `omega` function:

  ```
  let omega = fun f → f f
  ```

- Let us define a function `fix`, called the **Y combinator** in λ-calculus:

  ```
  let fix = fun g → omega (fun h → fun x → g (h h) x)
  ```

- We can show that, for any function *f*,

$$[\![f(\mathsf{fix}\, f)]\!]\rho = [\![\mathsf{fix}\, f]\!]\rho.$$

- In other words, we have

$$f(\mathsf{fix}\, f) \equiv \mathsf{fix}\, f.$$

- The fixed point x of a (continuous) function g satisfies $g(x) = x$.
- Therefore, the fixed point of f (if it exists) is the value of $(\mathsf{fix}\, \mathsf{f})$.

Fixed points

- Let us consider the following definitions:

```
let pre_fact =
  fun f → fun n → if n = 1 then 1 else n * f(n-1)
let fact = fix pre_fact
```

- We see that `fact` is the fixed point of `pre_fact`:

$$\begin{aligned}
[\![\texttt{fact}]\!]\rho &= [\![\texttt{fix pre_fact}]\!]\rho \\
&= [\![\texttt{pre_fact (fix pre_fact)}]\!]\rho \\
&= [\![\texttt{pre_fact fact}]\!]\rho \\
&= [\![\texttt{fun n} \rightarrow \texttt{if n = 1 then 1 else n * fact(n-1)}]\!]\rho
\end{aligned}$$

- It is as if the definition of "`fact`" were recursive... but it is not.

- The function "`fact`" coincides pointwise with the factorial function.
- Let us extend mini-ML with a native recursive binder, using the "`fix`" function:

$$[\![\textbf{let rec } f = e_1 \textbf{ in } e_2]\!]\rho = [\![\textbf{let } f = \mathsf{fix}(\textbf{fun } f \to e_1) \textbf{ in } e_2]\!]\rho.$$

Extensions

- Let us extend our mini-ML with the following expressions.
  - *Boolean constants* **true** or **false**
  - *Boolean operators* (&&) or (||) or **not**
  - *n-tuple* $e_1,\ldots,e_n$
  - *conditional* **if** e_0 **then** e_1 **else** e_2
  - *local recursive binding* **let rec** $f = e_1$ **in** e_2
- Note that parentheses are recommended around tuples, but are not mandatory.
- Let us distinguish the variables occurring after **let** and **fun**, because they are **irrefutable patterns**, noted $\overline{p}$:
  - *function* **fun** $\overline{p} \rightarrow e$
  - *local definition* **let** [**rec**] $\overline{p} = e_1$ **in** e_2

Irrefutable patterns

- An irrefutable pattern $\overline{p}$ is defined recursively by the following cases:
  - *variable* $f,g,h,x,y,z,$ etc.
  - *unit* $()$
  - *n-tuple* $\overline{p}_1,\ldots,\overline{p}_n$
  - *parenthesis* $(\overline{p})$
  - *underscore* $\_$
- Note that, from the syntactic point of view, irrefutable patterns are special cases of expressions, except the underscore, which is a special case avoiding the creation of a binding.
- The sentence e is equivalent to **let** $\_ = e$.

- The comma takes priority over the arrow:
$$\textbf{fun}\ x \to x,y = \textbf{fun}\ x \to (x,y).$$
- In order to alleviate the notation $\textbf{fun}\ \overline{p}_1 \to \ldots \to \textbf{fun}\ \overline{p}_n \to e$, we define the equivalent expressions:
  - **let** [**rec**] $f = \textbf{fun}\ \overline{p}_1\ \overline{p}_2 \ldots \overline{p}_n \to e$ (new expression)
  - **let** [**rec**] $f\ \overline{p}_1\ \overline{p}_2 \ldots \overline{p}_n = e$ (new sentence)
- For example, we would write
```
let mult = fun x y → x * y in
let eq x y = (x = y) in
let rec fact n = if eq n 1 then 1 else mult n (fact(n-1))
in fact 9
```
- Note the two different meanings of `(=)`: one is definitional, the other is operational (a Boolean operator).

- We also extend the syntax to alleviate certain expressions. By definition,

  $$\textbf{let } \overline{p}_1 = e_1 \textbf{ and } \overline{p}_2 = e_2 \ldots \textbf{ and } \overline{p}_n = e_n \textbf{ in } e$$

  is equivalent to

  $$\textbf{let } \overline{p}_1,\ldots,\overline{p}_n = e_1,\ldots,e_n \textbf{ in } e$$

  but the latter requires more memory (for the tuples in the heap).
- These are called **parallel definitions** because they occur within the same environment and could be conceived as happening in parallel.
- The two following sentences are equivalent:

```
let x = 1 and y = 2 in x + y
let x,y = 1,2 in x + y
```

Mutually recursive definitions

- Parallel definitions are useful for **mutually recursive definitions**:
  - **let rec** $\overline{p}_1 = e_1$ **and** $\overline{p}_2 = e_2 \ldots$ **and** $\overline{p}_n = e_n$ **in** e (expression)
  - **let rec** $\overline{p}_1 = e_1$ **and** $\overline{p}_2 = e_2 \ldots$ **and** $\overline{p}_n = e_n$ (sentence)

- A silly example:

```
let rec even n = if 2*(n/2) = n then true else odd (n-1)
and odd n = if 2*(n/2) = n then false else even (n-1)
```

where `(/)` is the Euclidean division.

- Let us consider the case where the irrefutable patterns are variables:

$$\textbf{let } x = e_1 \textbf{ and } y = e_2 \textbf{ in } e$$

  where $x \neq y$.

- If x is not free in e_2, i.e., $x \notin \mathcal{F}(e_2)$, then the previous construct is equivalent to

$$\textbf{let } x = e_1 \textbf{ in let } y = e_2 \textbf{ in } e.$$

- If x is free in e_2, that is, $x \in \mathcal{F}(e_2)$, then the previous construct is defined as being equivalent to

$$\begin{array}{l} \underline{\textbf{let } z = x \textbf{ in}} \\ \quad \textbf{let } x = e_1 \textbf{ in} \\ \quad \textbf{let } y = \underline{\textbf{let } x = z \textbf{ in}}\ e_2 \\ \textbf{in } e \end{array}$$

  where $z \notin \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \cup \mathcal{F}(e)$, in order to avoid being captured by e_1, e_2 or e.

Mutually recursive definitions

- The parallel **let rec** (with **and**) can always be reduced to a simple **let rec** (with **in**) by parameterising one of the definitions by the other.
- Consider the following sentence

  $$\textbf{let rec } x = e_1 \textbf{ and } y = e_2 \textbf{ in } e$$

  where $x \neq y$.
- It is equivalent, by definition, to

  $$\begin{array}{l}\textbf{let rec } x = \underline{\textbf{fun } y \rightarrow} \; e_1 \textbf{ in} \\ \quad \textbf{let rec } y = \underline{\textbf{let } x = x\, y \textbf{ in}} \; e_2 \textbf{ in} \\ \quad \underline{\textbf{let } x = x\, y} \\ \underline{\textbf{in}} \; e\end{array}$$

- Note how $(x\, y)$ is evaluated twice.

- The aim of **static typing** is to detect and reject at compile-time some kinds of programming errors which would otherwise result into an error at run-time, like `(1 2)` or `(fun x → x) + 1`.
- With this aim, a **type** is assigned by the compiler or the programmer to each subexpression of an expression (in particular, the whole program).
- For example, *int* for an arithmetic expression, or $int \to int$ for a function from the integers to the integers.
- The **coherence** of these types is checked by the OCaml compiler, that is, if we think of a type as an assumption, no assumption must contradict another one.

- It is undecidable to determine all possible run-time errors for all programs before running them.
- **Type systems**, on the other hand, are often **decidable**, because it is considered a good design to ensure that the compiler terminates on all input programs.
- Therefore, it is impossible for the compiler to reject only those programs that would "go wrong" at run-time.
- In other words, all type systems reject innocent programs, and that is why the quest for better type systems, with better compromises between unfair rejection and guaranteed safety, will never end.

Types

- **Types are terms** associated to expressions, either by the programmer or the compiler, and they capture some **invariant properties** which can be automatically composed and checked.
- A type t is a term defined recursively by the following cases:
  - *simple* char, bool, int, string, float, unit
  - *Cartesian product* $t_1 \times \ldots \times t_n$
  - *functional* $t_1 \rightarrow t_2$
  - *parenthesis* (t)
  - *free variable* α, β, γ, etc.
  - *parameterised* α list
- Until now, we have not encountered values of the type float, char or string, but they are fairly obvious.
- We write $\times$, α, β, etc. what is written *, 'a, 'b, etc. in the source code.

Type expressions

- The Cartesian product takes multiple arguments, instead of being binary like in mathematics, and $(\times)$ is not associative in OCaml:
$$t_1 \times t_2 \times t_3 \neq (t_1 \times t_2) \times t_3 \neq t_1 \times (t_2 \times t_3).$$
- The arrow is used in types too, where it is also right-associative:
$$t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_{n-1} \rightarrow t_n$$
is equivalent to
$$t_1 \rightarrow (t_2 \rightarrow (\ldots(t_{n-1} \rightarrow t_n))\ldots).$$
- The Cartesian product has priority over the arrow:
$$t_1 \times t_2 \rightarrow t_3$$
is equivalent to
$$(t_1 \times t_2) \rightarrow t_3.$$

Type expressions

- The OCaml compiler computes a type for each expression: this is called **static type inference**.
- For the simple constants, we have

Type	Values	Some functions
unit	()	
bool	**true false**	&& || not
int	1 2 max_int, etc.	+ - * /, etc.
float	1.0 2. 1e4, etc.	+. -. *. /. cos, etc.
char	'a' '\n' '\097', etc.	Char.code Char.chr, etc.
string	"a\tb\010c\n", etc.	(^) s.[i], etc.

- Operations on floating point numbers are written differently from their equivalent counterparts on integers.

Type definitions

- We further extend the syntax of the sentences to allow the binding of a type to a name, or **type definition**:
  - *type definition* **type** [**nonrec**] $q = t$
  - *parallel type definition* **type** [**nonrec**] $q_1 = t_1$ [**and** $q_2 = t_2$ ...]
  - *type variable* q

  In particular:
  - *type aliasing* **type** $q_0 = q_1$
- By default, type definitions can be recursive (we will see an example later). If not, use the keyword **nonrec**.
- We can now write, for instance, the following:

```
type abscissa = float
type ordinate = float
type point = abscissa × ordinate
type origin = point
```

- Tuples contain data of the same type and their arity is part of their type.
- For instance, the pair (1,2) has type int × int, and the triplet (1,2,3) has type int × int × int, thus are incompatible.
- Consider also the following session at the OCaml toplevel:

```
# let middle x y = (x+y)/2;;
val middle : int → int → int

# let middle (x,y) = (x+y)/2;;
val middle : int × int → int
```

- In the above, the OCaml compiler figures out in both cases that, since we compute (x+y), that implies that x and y must have type *int*.

- When a variable is not constrained by its uses, like x in (**fun** x $\rightarrow$ x), it is assigned the most general type, that is, a **type variable**:
  # let id = **fun** x $\rightarrow$ x;;
  val *id : $\alpha \rightarrow \alpha = \langle fun \rangle$*
- This means that the above identity function can be applied to any kind of value, and that the type of the call is that of the argument.
- This property of a function to accept values of different types is called **parametric polymorphism**. Consider more examples:
  # fun (x,y,z) $\rightarrow$ x;;
  - *: $\alpha \times \beta \times \gamma \rightarrow \alpha$.*
  # let compose f g = **fun** x $\rightarrow$ f (g x);;
  val *compose : $(\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \alpha) \rightarrow \gamma \rightarrow \beta = \langle fun \rangle$*
  # let rec power f n =
  if n $\leqslant$ 0 **then** id **else** compose f (power f (n-1));;
  val *power : $(\alpha \rightarrow \alpha) \rightarrow$ int $\rightarrow (\alpha \rightarrow \alpha) = \langle fun \rangle$*

```
# let compose f g = fun x → f (g x);;
```
val compose : $(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta = \langle fun \rangle$

The type of the function **compose** is inferred as follows:

- the first argument **f** is an arbitrary function, hence of type $\alpha \to \beta$;
- the second argument **g** is a function whose result is the argument of **f**, therefore of type α;
- the domain of **g** is arbitrary, hence **g** has type $\gamma \to \alpha$;
- the function **compose** has an argument **x** which is passed to **g**, hence has type γ;
- finally, the result of **compose** is returned by **f**, hence of type β.

- The **equality operator** in OCaml is polymorphic and built in:

```
# (=);;
- : 'a -> 'a -> bool = <fun>
```

- We must be very careful that it coincides with our intended notion of equality, because the compiler will always accept it on **non-functional values**.
- It consists in the **syntactical equality**: two values are equal if they have the same structure and if their respective parts are equal, recursively. It fails on closures.

```
# 1 = 1 && [1;2] = [1;2] && "yes" = "yes";;
- : bool = true
# (fun x -> x) = (fun x -> x);;
Exception: Invalid_argument "equal: functional value".
```

Pattern matching

- Let us extend expressions with **matching** a value against **patterns**. Let p_i be patterns in

$$\textbf{match } e \textbf{ with } p_1 \to e_1 \mid \ldots \mid p_n \to e_n.$$

- Patterns are recursively defined by the following cases:
  - *variable* f,g,h (functions) and x,y,z (others).
  - *unit or constant* $()$ or 0 or **true**, etc.
  - *n-tuple* $p_1,\ldots,p_n$
  - *parenthesis* (p)
  - *underscore* _
- Note that irrefutable patterns are patterns.
- **Pattern matching** is used to define functions case by case:

```
let rec fib n =
  match n with
    0 → 1
  | 1 → 1
  | _ → fib(n-1) + fib(n-2)
```

- As in mathematics, the order of the cases is that of the writing, and the previous definition reads like so:

  > If the value of `n` has the shape `0`, then `fib(n)` has the value 1; else, if the value of `n` has the shape 1, then `fib(n)` has the value 1; for any other value of `n`, `fib(n)` has the value of `fib(n-1) + fib(n-2)`.

- What is the meaning of the relation "The value *v* has the shape *p*"? Or, equivalently: "The pattern *p* matches the value *v*"?
- A constant, including `()`, has the shape of itself in a pattern (constants are both values and patterns).
- A *n*-tuple (value) has the shape of a *n*-tuple in a pattern.
- Any value has the shape of a variable in a pattern or of the underscore '`_`'.

- Note that patterns do not match closures, which means that the e in "**match** e **with**" must not have a closure as a value. In the case of constants, including $()$, matching coincides with equality.
- Matching is the evaluation of a **match** expression.
- Informally, the evaluation of

  $$\textbf{match } e \textbf{ with } p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

  begins with that of e into v.
- Next, v is compared to the patterns p_i for increasing values of i.
- If p_i is the first pattern to match v, then the value of the matching is the value of e_i.
- Here is a definition of the logical disjunction:

```
let disj (x,y) = match x,y with
                   false, false → false
                 | _ → true
```

Variant types

- **Variant types** are a generalisation of enumerations.
- For instance, Boolean values can be (re)defined as
  `type boolean = True | False`
  `let true' = True and false' = False`
- Note that **value constructors**, like `True` or `False`, must begin with an uppercase letter.
- Pattern matching examines the values of a variant type:
  `let int_of_boolean b = match b with True → 1 | False → 0`
- Value constructors can also carry information beyond their mere name. For instance, a pack of playing cards can be defined as
  `type suit = Heart | Diamond | Spade | Club`
  `type face = Ace | King | Queen | Jack | Simple of int`
  `type ordinary = suit × face`
  `type card = Card of ordinary | Joker`

Variant types

Let us define the cards and the functions creating them:

```
# let jack_of_spade = Card (Spade, Jack);;
val jack_of_spade : card = Card (Spade, Jack)

# let card f c = Card (c,f);;
val card : face → suit → card = ⟨fun⟩

# let king = card King;;
val king : suit → card = ⟨fun⟩

# let value c =
    match c with
      Card Ace         → 14
    | Card King        → 13
    | Card Queen       → 12
    | Card Jack        → 11
    | Card (Simple k) → k
    | Joker            → 0;;
```

Pattern matching (resumed)

- A pattern can capture several cases at once with an **underscore**:

```
let is_simple c =
  match c with
    Card (Simple _) → true
  | _ → false
```

- A pattern matching is incomplete if there exists at least a value which cannot be matched by any pattern.
- In that case, a warning is printed at compile-time because it is strongly advised to **avoid incomplete pattern matchings**:

```
# let simple c = match c with Card (_, Simple k) → k;;
Characters 15-51
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: Joker
val simple : card → int = ⟨fun⟩
```

Pattern matching (resumed)

- A variable cannot occur twice or more in the same pattern, because this requires to call the polymorphic equality during pattern matching, which may be costly and even loop on circular values (and we want compilers to terminate).
- Such a pattern is said to be **non-linear**. (We have seen these in the context of rewrite systems.)

```
# fun (x,y) → match x,y with Card z, z → true;;
Characters 40-41
This variable z is bound several times in this matching.
```

- Lists can be defined as a polymorphic, **recursive variant type**:

  type α `list' = Nil | Cons of` $\alpha \times \alpha$ `list'`

- The value constructors `Nil` and `Cons` are traditional in the community of functional languages. The former denotes the empty list; the latter represents a non-empty list.
- We used the same naming in the context of rewrite systems.
- A non-empty list is then modelled as a pair whose first component is an element (of type α) of the list and the second component is the remainder of the list, or a sublist (of type α `list'`). For instance:

```
let empty = Nil
let single_char = Cons ('a', empty)
let single_int = Cons (4, empty)
let long = Cons (1, Cons (2, Cons (3, single_int)))
```

- By default, the OCaml system predefines a type α `list`, whose constructor for the empty list is `[]`, and those of the non-empty list is `(::)`, used in infix position.
- The operator **catenating** two lists is also predefined and noted **`(@)`**. Note that it is *not* a constructor. Here are equivalent lists:

```
let l = 1::(2::3::(4::[]))
let l = 1::2::3::4::[]
let l = [1;2;3;4]
```

- As an example, consider the **reversal** of a list:

```
let reverse l =
  let rec aux acc l =
    match l with
        [] → acc
    | h::t → aux (h::acc) t
  in aux [] l
```

Polymorphic iterators on lists

- Let us consider writing an **iterator on lists**, that is, a function that applies a given function to the elements of a list.
- For example, given a list of integers, we may want to compute the list made of these integers incremented by one, or doubled, etc.
- This is called a **map**.
- We would like to do

```
# map (fun n -> n+1) [4;4;5;0];;
- : int list = [5;5;6;1]
```

- In that aim, we need to leverage **polymorphism**, because we do not want to limit ourselves to lists of integers, or to functions applicable only to integers.

- We begin with the name of the function and its parameters:

```
let map f l = ...
```

- Next, we destructure the list parameter `l` and decide what to do for each case:

```
let map f l =
  match l with
      [] → ...
  | h::t → ...
```

- If the list is empty, the map should do nothing, otherwise, we apply the iterated function `f` to the first element `h` and we recur on the rest of the list `t`:

```
let rec map f l = (* Note the keyword rec *)
  match l with
      [] → []
  | h::t → f h :: map f t
```

- In the case of `map`, each call of the iterated function does not inform another. We may want something more general, and **accumulate** those individual calls. The iterated function then would both operate on a particular list element, and also update the accumulator.
- For instance, we may want to sum all the integers in a list, in which case the initial value for the accumulator would be 0.
- This general kind of iterator is a **fold**.
- Let us start with the parameters:

  ```
  let fold f l a = ...
  ```

  where `f` is the iterated function, `l` is the list and `a` is the accumulator.
- For summing all integers in a list, we would then write:

  ```
  # fold (+) [4;4;5;0] 0;;
  - : int = 13
  ```

- A map is then a special case of a fold:

  ```
  # let map f l = fold (fun e a → f e :: a) l [];;
  val map : ('α → 'β) → 'α list → 'β list = <fun>
  # map [4;4;5;0];;
  - : int list = [5;5;6;1]
  ```

- If we want to define **map** that way, we are implicitly specifying that the fold is **leftwards**, that is, it iterates from right to left in the list.
- Indeed, when the iterated function was `(+)`, the direction of iteration did not matter, because

  $$4+(4+(5+(0+\mathbf{0}))) = (((\mathbf{0}+4)+4)+5)+0.$$

  (The initial value of the accumulator is **0**.)

- When the iterated function is `(::)`, that is no longer the case: if we push `(f a)` on previously transformed elements, we must chose if those were visited coming from the top of the list or from the end.
- The definition of **map** we gave above:

  let `map f l = fold (`**fun** `e a` $\rightarrow$ `f e :: a) l []`

  returns the transformed elements in the **same order** as in the input list `l`.
- If `fold` visited the list elements from left to right, then the result of the call `map [4;4;5;0]` would be `[1;6;5;5]`, which is the reverse of what we expect, that is, `[5;5;6;1]`.
- For that, we need to push the transformed elements from the end of `l` first, all the way to its beginning.

- Let us see how `fold` is made:

```
let rec fold f l a =
  match l with
      [] → ...
  | h::t → ...
```

- The case of the empty list does **not** yield the empty list, but the final value of the accumulator:

```
let rec fold f l a =
  match l with
      [] → a (* Not [] *)
  | h::t → ...
```

- Finally, the general case:

```
# let rec fold f l a =
    match l with
        [] → a
    | h::t → f h (fold f t a);;
```

val fold : $(\alpha \to \beta \to \beta) \to \alpha\ list \to \beta \to \beta = \langle fun \rangle$

- Note that the recursive call in the definition of `fold` is not in tail position. (We saw that concept in the lecture on a theory of programming.)
- In this case, this means that all the calls to the iterated function `f` will be waiting for all the calls to `fold` to be finished (as well as all the calls to `f` on the previous elements of the list).
- Generally speaking, the call stack will grow proportionally to the length of the list.

- In OCaml, the **module List** exports the function fold under the fully qualified name List.fold_right.
- The **rightwards iterator** is called List.fold_left.
- It is defined as follows:

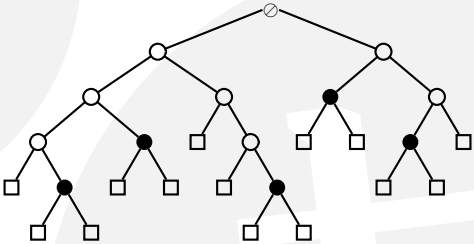
```
let rec fold_left f a l =
  match l with
      [] → a
  | h::t → fold_left f (f a h) t
```

- The type is different from List.fold_right:
  val `fold_left :` $(\alpha \to \beta \to \alpha) \to \alpha \to \beta$ *list* $\to \alpha =$ ⟨*fun*⟩

- The order in which the iterated function takes its arguments is not fundamental here, but the OCaml library makes a distinction between List.fold_left and List.fold_right.
- In the former, the accumulator is passed before the element of the list (`f a h`), while its the reverse in the latter (`f h (...)`).
- Note that the recursive call in the definition of fold_left is in **tail position**, which means that the call stack will only grow a constant amount during iteration if the calls to the iterated function also grow a constant amount.
- This is a good property when running programs on operating systems or hardware that limit the size of the call stack, in which case it is advised to use List.fold_left over List.fold_right whenever possible, even if that means that you have to reverse the final value of the list accumulator.

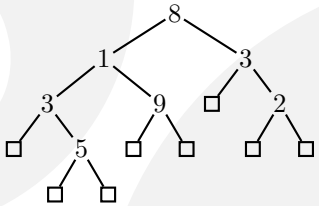
Binary trees

- A **binary tree** is a special case of tree, where each node has either no children or exactly two.

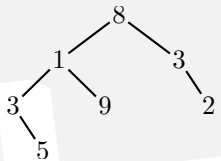


- Nodes are of two kinds: **internal** (⊘, ○ and ●) or **external** (□).
- The **root** is the only internal node without a parent (⊘).
- **Leaves** (●) are internal nodes whose children are two external nodes (□).

- Internal nodes are usually associated with some kind of information, whilst external nodes are not, like



Extended binary tree



Pruned binary tree

- To define a binary tree in OCaml means to write a type whose values are all possible binary trees. We need a polymorphic type, so the information stored in the tree can be of any type:

```
type α tree = ...
```

Binary trees in OCaml

- We need to ponder how many shapes a binary tree can take, just like lists.
- Each shape as a value constructor and maybe some data.
- The empty tree is an external node, without information:
  type α `tree = Ext | ...`
- A non-empty tree is an internal node carrying some information, and which has two children:
  type α `tree = Ext | Int of` $\alpha \times \alpha$ `tree` $\times$ α `tree`
- The tree in the first figure corresponds to the OCaml value

```
Int (8, Int (1, Int (3, Ext,
                        Int (5, Ext, Ext)),
                 Int (9, Ext, Ext)),
         Int (3, Ext,
                 Int (2, Ext, Ext)))
```

Tree traversals

- We may want to write an OCaml function that sums the integers in a binary tree like so:

```
let rec add t =
  match t with
    Int (n, left, right) → n + add left + add right
  | Ext → 0
```

- The only issue is that we wanted a function that does *not* apply to empty trees (it is partial), but we need the value 0 for the recursion to work.

The option type

- We need another function that checks whether the **whole tree** is empty, and, if, returns a special value that means "No sum is possible".
- The solution is to use a predefined type called the **option type**:

  type α `option = None | Some` **of** α
- For example, `(Some 3)` means "There was a value, and it is 3", whereas `None` means "There was no value."

- We can now finish our definition:

```
let rec add t =
  match t with
    Int (n,left,right) → n + add left + add right
  | Ext → 0

let add t =
  match t with
    Int (n,left,right) → Some (n + add left + add right)
  | Ext → None
```

- Note that the definition of the second function "add" hides the first one and uses it, because it is not recursive.

A shortcut

- The **design pattern**

```
let rec add t =
  match t with
    ... → ...
```

where `t` is not used on the right-sides is quite common.

- That is why OCaml provides a shortcut with a new keyword:

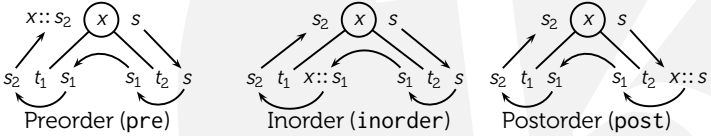
```
let rec add = function
  Int (n,left,right) → n + add left + add right
| Ext → 0

let add = function
  Int (n,left,right) → Some (n + add left + add right)
| Ext → None
```

Tree traversals (resumed)

Let `t` be the tree `Int(x,l,r)`.

- The **preorder traversal** of `t` is the list whose first element is `x`, followed by the elements of the preorder traversal of `l`, followed by the elements of the preorder traversal of `r`. The previous tree yields `[8;1;3;5;9;3;2]`.
- The **postorder traversal** of `t` is the list made of the elements of the postorder traversal of `l`, followed by the elements of the postorder traversal of `r`, followed by `x`. The previous tree yields `[5;3;9;1;2;3;8]`.
- The **inorder traversal** of `t` is the list made of the elements of the inorder traversal of `l`, followed by `x`, followed by the elements of the inorder traversal of `r`. The previous tree yields `[3;5;1;9;8;3;2]`.

- We want to build the traversals only by pushing (consing).
- We must add an **auxiliary parameter**, originally set to the empty list, on which the contents of the nodes are pushed in the proper order. For preorder:
  `let pre t s = ...`
  `let pre t = pre t []`
- This kind of extra parameter is called an **accumulator**, because it accumulates parts of the total result until the main structure (`t`) is fully traversed.
- We should interpret this accumulator when considering the call $(\mathsf{pre}\, t\, s)$. Let us consider the tree $t = \mathsf{Int}(x, t_1, t_2)$ in the figures



Preorder traversal

- Each arrow is a step in the traversal and connects different stages of the accumulator: a downwards arrow points to the argument of a recursive call on the corresponding child; an upwards arrow points to the value of the call on the parent.
- For instance, the leftmost figure, the subtree t_2 corresponds to the recursive call $(\mathsf{pre}\ t_2\ s)$ whose value is named s_1, which we note
$$\mathsf{pre}\ t_2\ s \twoheadrightarrow s_1$$
- Likewise, we have $\mathsf{pre}\ t_1\ s_1 \twoheadrightarrow s_2$. Therefore
$$\mathsf{pre}\ t_1\ (\mathsf{pre}\ t_2\ s) \twoheadrightarrow s_2$$
- The root is associated with the evaluation
$$\mathsf{pre}\ t\ s \twoheadrightarrow x :: s_2$$
Consequently,
$$\mathsf{pre}\ t\ s \equiv x :: \mathsf{pre}\ t_1\ (\mathsf{pre}\ t_2\ s).$$
- The rules for external nodes are not shown and simply consist in leaving the accumulator invariant.

Preorder traversal

- The OCaml function for the preorder traversal of a binary tree is

```
let rec pre t s =
  match t with
                Ext → s
  | Int (x,t1,t2) → x :: pre t1 (pre t2 s)

let pre t = pre t []
```

- If we swap the parameters `t` and `s` above, we can use the **function** construct because `t` is not used on the right-hand sides:

```
let rec pre s = function
             Ext → s
| Int (x,t1,t2) → x :: pre (pre s t2) t1

let pre t = pre [] t
```

- Similarly, we can derive the OCaml function for the inorder traversal of a binary tree:

```
let rec inorder s = function
            Ext → s
| Int (x,t1,t2) → inorder (x :: inorder s t2) t1

let inorder t = inorder [] t
```

- The same goes for the postorder traversal:

```
let rec post s = function
            Ext → s
| Int (x,t1,t2) → post (post (x::s) t2) t1

let post t = post [] t
```

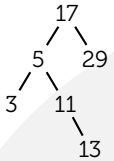
- Searching in a binary tree can be costly because, in the worst case, the whole tree must be traversed.
- To improve upon the worst case, two scenarios are desirable:
  1. the binary tree should be as **balanced** as possible, that is, the nodes should lie as close as possible to the root (for a given notion of distance), and
  2. at a given node, **only the left or the right subtree should be visited**, a decision taken solely by comparing the key at the root and the sought key.
- The simplest solution consists in satisfying the second goal, and see later how to also reach the first.
- A **binary search tree** is either `Empty`, or it is a value `BST(`x`,`t_1`,`t_2`)`, such that the key x is greater than or equal to the keys in the left subtree t_1, and lower than the keys in t_2.

- The definition of an OCaml type for such trees is isomorphic to that of binary trees, as the ordering constraint cannot be expressed at the type level:

  type α `bst = Empty | BST of` $\alpha \times \alpha$ `bst` $\times \alpha$ `bst`
  type α `tree = Ext | Int of` $\alpha \times \alpha$ `tree` $\times \alpha$ `tree`

- Therefore, let us retain the type `tree`.
- The **total ordering of the keys** (that is, the fact that any two keys must be comparable) can be expressed at the **module level**, which is an advanced feature of OCaml.
- We will not present modules in this lecture, although they are very important to write **composable software** that is type safe (libraries).
- A consequence is that the inorder traversal of a binary search tree is the list of its keys sorted in non-decreasing order.

- Here is an example:



- The OCaml value for that tree is:

```
let t =
  Int (17, Int (5, Int (3, Ext, Ext),
                   Int (11, Ext,
                            Int (13, Ext, Ext))),
           Int (29, Ext, Ext))
```

- The inorder traversal is:

```
# let l = inorder t;;
val l : int list = [3;5;11;13;17;29]
```

- Let consider checking the **membership** of a key k in a binary search tree.
- Any key is absent from the empty tree.
- If the tree is not empty, it has the shape `Int`(x, t_1, t_2).
- We compare k with x: if equal, we found it; if lower, we resume the search on t_1, else on t_2.
- In OCaml:

```
let rec mem k = function
  Ext → false
| Int (x,t1,t2) → if k < x then mem k t1
                  else if x < k then mem k t2
                       else true
```

- Since all unsuccessful searches end at an external node, it is tempting to start the insertion of a unique key by an unsuccessful search and then grow a leaf with the new key at the external node we reached.
- If the key is already present in the tree, we decide to leave the tree unchanged.
- Let us modify the OCaml definition for membership into

```
let rec add_leaf k = function
  Ext → Int (k, Ext, Ext) (* Was "false" *)
| Int (x,t1,t2) →
    if k < x then Int (x, add_leaf k t1, t2)
    else if x < k then Int (x, t1, add_leaf k t2)
                  else Int (x,t1,t2) (* Was "true" *)
```

- If you recall **data sharing** in the context of rewrite systems, you would have noticed that when the sought key is already in the tree, a new internal node is created identical, instead of sharing the existing one.
- Moreover, all the nodes visited before from the root also have been duplicated for nothing.
- We can remedy the first problem by using a feature of OCaml patterns called **aliases**, whereby we can name a subpattern and use that name on the right hand-sides of the rules, like so:

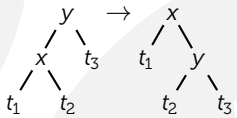
```
let rec add_leaf k = function
  Ext → Int (k, Ext, Ext)
| Int (x,t1,t2) as t → (* t = Int (x,t1,t2) *)
    if k < x then Int (x, add_leaf k t1, t2)
    else if x < k then Int (x, t1, add_leaf k t2)
                  else t (* Was Int (x,t1,t2) *)
```

Root insertion

- If recently inserted keys are looked up, the cost is relatively high because these keys are leaves or close to a leaf.
- In this scenario, instead of inserting a key as a leaf, it is better to insert it as a root.
- The idea is to perform a leaf insertion and, on the way back to the root (that is to say, after the recursive calls are evaluated, one after the other), we perform rotations to bring the inserted node up to the root.
- More precisely, if the node was inserted in a left subtree, then a **right rotation** brings it one level up, otherwise a **left rotation** has the same effect.
- The composition of these rotations brings the leaf to the root.

Right rotation

- The right rotation can be depicted as

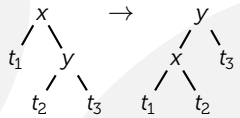


- We can see that x goes "up" and y goes "down" one level (this is a local transformation).
- It is defined in OCaml as

```
let rotr = function
  Ext → Ext
| Int (y, Int(x,t1,t2), t3) → Int (x, t1, Int(y,t2,t3))
```

Left rotation

- The left rotation can be depicted as



- We can see that x goes "down" and y goes "up" one level (this is a local transformation).
- It is defined in OCaml as

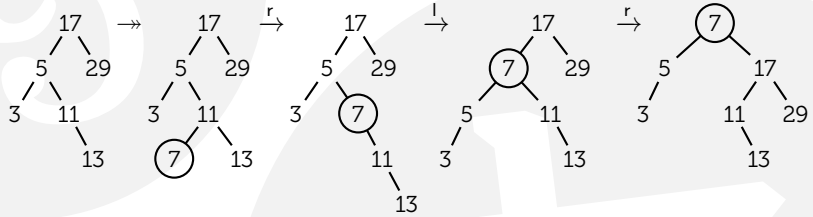
```
let rotl = function
  Ext → Ext
| Int (x, t1, Int(y,t2,t3)) → Int (y, Int(x,t1,t2), t3)
```

- Left and right rotations are the **inverse** of each other:

$$\forall t.\mathsf{rotl}\ (\mathsf{rotr}\ t) \equiv \mathsf{rotr}\ (\mathsf{rotl}\ t) \equiv t.$$

Root insertion (resumed)

Let us consider an example, where 7 is first added as a leaf, and then a series of rotations, left and right, bring it to the root:



- Remark that the transitive closure $(\twoheadrightarrow)$ captures the preliminary leaf insertion, $(\xrightarrow{r})$ is a right rotation and $(\xrightarrow{l})$ is a left rotation, both applied to the node containing 7.
- Note also that the inorder traversal of all those trees remains invariant and a sorted list.

Root insertion

It is now a simple matter to modify the definition of add_leaf so it becomes root insertion add_root as follows:

```
let rec add_root k = function
  Ext → Int (k, Ext, Ext)
| Int (x,t1,t2) as t →
    if k < x then rotr (Int (x, add_root k t1, t2))
    else if x < k then rotl (Int (x, t1, add_root k t2))
                  else t
```

We simply added a right rotation just after performing a left insertion, and a left rotation after a right insertion.

Balancing a binary search tree

- Whether we insert a root or a leaf to a binary search tree, we may still obtain **degenerate trees**, that is, trees isomorphic to lists (each internal node has exactly one internal child, except the last).
- You may recall that we mentioned page 83 the usefulness of balancing a search tree, so that these extreme cases do not happen.
- There exist different criteria to balance a tree, the two most common relying on
  1. the **distance** of a node to the root (the path length),
  2. the **weight** of a node (the number of nodes in its subtrees).
- Unfortunately, discussing and implementing these strategies would lead us too far in this lecture, but you can refer to the literature.

Sorting with Binary Search Trees

- We already noticed how insertions leave the inorder traversal of a binary search tree unchanged.
- Also the inorder traversal of a binary search tree is the list of its keys sorted in non-decreasing order.
- Combining these two facts (which can be made formal claims and proved), we deduce that we can use binary search trees to sort a list of keys chosen from a totally ordered set.
- We start by inserting each key in an empty binary search tree.
- Next, we collate the inorder traversal of the final tree.

```
let sort l = inorder (fold add_leaf l Ext)
```