Smart Contracts in LIGO

Christian Rinderknecht
rinderknecht@free.fr
Ligo LANG

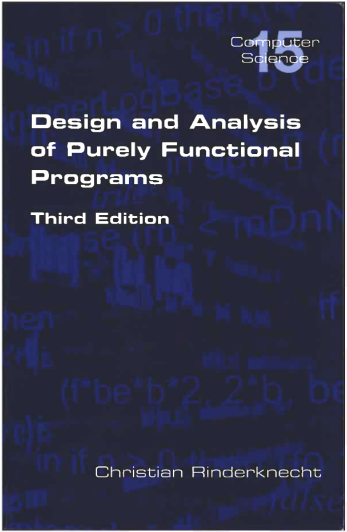
12 February 2020

Nomadic Labs Training

- My alma mater is *Université Pierre et Marie Curie* (UPMC, a.k.a. Paris 6).
- I did my doctoral studies at INRIA, one of the most prestigious research institutes in informatics in France.
- I was a member of the team that developed the programming language OCaml.
- I went on to work as an engineer, a researcher and a professor for many years, across several countries (France, Korea, Hungary, Sweden), both in academia and private companies.
- In 2018, I joined Nomadic Labs, where a lot of the maintenance of the Tezos blockchain is done. My expertise is in compiler construction and functional programming. I have been working on a high-level language for writing smart contracts on Tezos.

A personal introduction

My book about functional programming is published in London!



- At Nomadic Labs, I had two colleagues, Suzanne Dupéron and Gabriel Alfour.
- Gabriel founded a spin-off company mid-2019 and Suzanne and I joined him.
- It is funded by the Tezos Foundation to develop tools to ease the creation of distributed applications on Tezos.
- Beyond LIGO-the-language, we have produced a webIDE and a VSCode plug-in.
- The next target for LIGO-the-company is profitability.
- LIGO-the-company also helps with the training of Tezos users.

- Like Michelson, LIGO is a programming language for writing smart contracts on Tezos.
- Unlike Michelson, LIGO is a language akin to what a mainstream programmer would expect.
- This means that LIGO features variables, expressions, function calls, data types, pattern matching etc.
- Currently, LIGO is a DSL for the Tezos blockchain, but there is a plan to make LIGO a general-purpose language, or, at least, chain-agnostic.
- It is hosted here: `https://ligolang.org/`
- It is an open source, collective project (under the MIT licence).
- (If you guessed why the name “Michelson”, you will guess why “LIGO”.)

- Perhaps the most striking feature of LIGO is that it comes in different concrete syntaxes, and even different programming paradigms.
- In other words, LIGO is not defined by one syntax and one paradigm, like imperative versus functional.
- There is PascaLIGO, which is inspired by Pascal, hence is an imperative language with lots of keywords, where values can be locally mutated from within the scope where they have been declared.
- There is CameLIGO, which is inspired by the pure subset of OCaml, hence is a functional language with few keywords, where values cannot be mutated, but still require type annotations (unlike OCaml, whose compiler performs almost full type inference).
- There is ReasonLIGO, which is inspired by the pure subset of ReasonML, which is a JavaScript syntax on top of OCaml.

- Even within PascaLIGO, two styles are possible: terse or verbose. We illustrate the terse style here and in the documentation. We plan to offer automatic style checking and two-way conversion by pretty-printing.
- At the call site of a function, the arguments and the environment are always copied, therefore any mutation (in PascaLIGO) will have no effect on the caller's arguments or the environment at the call site.
- LIGO features higher-order functions, that is, functions can be passed as arguments to others.

Tooling for LIGO

- Several tools are currently being developed, aiming at facilitating the adoption of LIGO.
- A VSCode plug-in is available, featuring
  - syntax highlighting,
  - one-click compilation to Michelson,
  - *dry runs* to locally execute contracts on a sandbox.
- The architecture of VSCode, with its Language Server Protocol, hopefully opens the door to plugins written in any programming language, e.g., static analysis in OCaml.
- A web-based IDE with the same set of features.
- Start here: `https://ligolang.org/`

Research & Development on the LIGO compiler

- We are working on a more powerful type system which will enable the writing of more expressive contracts, featuring more type inference (less annotations) and enabling a greater variety of programming paradigms (e.g., object-oriented).
- We are writing a certified backend in Coq, that is, a Michelson code generator proven correct and extracted to OCaml from its specification.
- Those endeavours are not just engineering, they are instances of applied research and require a strong background on programming language theory.

Structure of a LIGO contract

- A LIGO contract is a series of constant and function declarations.
- The scope of those is called **top-level**, to distinguish declarations that may occur within functions.
- In particular, even in PascaLIGO, you cannot have mutable variables at the top-level.
- As a design pattern, there is usually one special function, which we call **main function** that is called with a parameter when the contract is invoked.
- The main function calls other functions according to the value of the contract parameter.
- Those functions are called **entrypoints**, following the Michelson convention.

Constant declarations

- LIGO lifts the basic types of Michelson, so we have string, nat, int etc.
- In PascaLIGO:
```
type breed is string
const dog_breed : breed = "Saluki"
```
- In CameLIGO:
```
type breed = string
let dog_breed : breed = "Saluki"
```
- In ReasonLIGO:
```
type breed = string;
let dog_breed : breed = "Saluki";
```

Numerical Types

- LIGO offers three built-in numerical types: `int`, `nat` and `tez`.
- Literals for natural numbers are positive integers immediately followed by `n`, like so: `13n`, `0n` etc.
- Literals for token amounts can have three forms:
  1. integral amounts of tez: `2tez`, `0tez`;
  2. fractional amounts of tez: `0.0003tez`;
  3. integral amounts of millionth of tez: `340000mutez`.
- For the sake of readability, you can insert underscores to group digits in the integral or fractional amounts, for example `340_000mutez`, or `34_0000mutez` (in Korea), or `0.000_3tez`.
- This also works for integers and natural numbers: `-10_245`, `19_355n`.

Addition in PascaLIGO

- LIGO lifts the constraints of Michelson arithmetics.

```
const a : int = 5 + 10                    // int + int yields int
const b : int = 5n + 10                   // nat + int yields int
const c : tez = 5mutez + 0.000_010tez // tez + tez yields tez

// tez + int or tez + nat is invalid
// const d : tez = 5mutez + 10n

const e : nat = 5n + 10n // two nats yield a nat

// nat + int yields an int: invalid
// const f : nat = 5n + 10;

const g : int = 1_000_000
```

Addition in CameLIGO

- LIGO lifts the constraints of Michelson arithmetics.

```
let a : int = 5 + 10            // int + int yields int
let b : int = 5n + 10           // nat + int yields int
let c : tez = 5mutez + 10mutez // tez + tez yields tez

// tez + int or tez + nat is invalid
// let d : tez = 5mutez + 10n

let e : nat = 5n + 10n // two nats yield a nat

// nat + int yields an int: invalid
// let f : nat = 5n + 10

let g : int = 1_000_000
```

Addition in ReasonLIGO

- LIGO lifts the constraints of Michelson arithmetics.

```
let a : int = 5 + 10;           // int + int yields int
let b : int = 5n + 10;          // nat + int yields int
let c : tez = 5mutez + 10mutez; // tez + tez yields tez

// tez + int or tez + nat is invalid:
// let d : tez = 5mutez + 10n;

let e : nat = 5n + 10n;  // two nats yield a nat

// nat + int yields an int: invalid
// let f : nat = 5n + 10;

let g : int = 1_000_000;
```

Subtraction in PascaLIGO

- LIGO lifts the constraints of Michelson arithmetics.

```
const a : int = 5 - 10

// Subtraction of two nats yields an int
const b : int = 5n - 2n

// Therefore the following is invalid
// const c : nat = 5n - 2n

const d : tez = 5mutez - 1mutez
```

Subtraction in CameLIGO

- LIGO lifts the constraints of Michelson arithmetics.

```
let a : int = 5 - 10

// Subtraction of two nats yields an int
let b : int = 5n - 2n

// Therefore the following is invalid
// let c : nat = 5n - 2n

let d : tez = 5mutez - 1mutez
```

Subtraction in ReasonLIGO

- LIGO lifts the constraints of Michelson arithmetics.

```
let a : int = 5 - 10;

// Subtraction of two nats yields an int
let b : int = 5n - 2n;

// Therefore the following is invalid
// let c : nat = 5n - 2n;

let d : tez = 5mutez - 1mutez;
```

Multiplication and Division in PascaLIGO

- LIGO lifts the constraints of Michelson arithmetics.

```
const a : int = 5 * 5
const b : nat = 5n * 5n
const c : tez = 5n * 5mutez // You can also multiply nat and tez

const d : int = 10 / 3
const e : nat = 10n / 3n
const f : nat = 10mutez / 3mutez
```

Multiplication and Division in CameLIGO

- LIGO lifts the constraints of Michelson arithmetics.

```
let a : int = 5 * 5
let b : nat = 5n * 5n
let c : tez = 5n * 5mutez // You can also multiply nat and tez

let d : int = 10 / 3
let e : nat = 10n / 3n
let f : nat = 10mutez / 3mutez
```

Multiplication and Division in ReasonLIGO

- LIGO lifts the constraints of Michelson arithmetics.

```
let a : int = 5 * 5;
let b : nat = 5n * 5n;
let c : tez = 5n * 5mutez; // You can also multiply nat and tez

let d : int = 10 / 3;
let e : nat = 10n / 3n;
let f : nat = 10mutez / 3mutez;
```

Casts

- You can cast an int to a nat and vice versa.
- In PascaLIGO:
```
const a : int = int (1n)
const b : nat = abs (1)
```
- In CameLIGO:
```
let a : int = int (1n)
let b : nat = abs (1)
```
- In ReasonLIGO:
```
let a : int = int (1n);
let b : nat = abs (1);
```

Checks

- You can check if a value is a nat by using a predefined cast function which accepts an int and returns an optional nat: if the result is not None, then the provided integer was indeed a natural number, and not otherwise.
- We will revisit this when we present the **variant** types.
- In PascaLIGO:

```
const is_a_nat : option (nat) = is_nat (1)
```

- In CameLIGO:

```
let is_a_nat : nat option = is_nat (1)
```

- In ReasonLIGO:

```
let is_a_nat : option (nat) = is_nat (1);
```

- In PascaLIGO:
```
const name : string = "Alice"
const greeting : string = "Hello"
const full_greeting : string = greeting ^ " " ^ name
```

- In CameLIGO:
```
let name : string = "Alice"
let greeting : string = "Hello"
let full_greeting : string = greeting ^ " " ^ name
```

- In ReasonLIGO:
```
let name : string = "Alice";
let greeting : string = "Hello";
let full_greeting : string = greeting ++ " " ++ name;
```

Booleans

- The type of the booleans is `bool`.
- In PascaLIGO:

```
const a : bool = True   // true also
const b : bool = False  // false also
```

- In CameLIGO:

```
let a : bool = true
let b : bool = false
```

- In ReasonLIGO:

```
let a : bool = true;
let b : bool = false;
```

Comparing Values

- In LIGO, only values of the same type can be compared. Moreover, not all values of the same type can be compared, only those with **comparable types**, which is a concept lifted from Michelson.
- Comparable types include, for instance, `int`, `nat`, `string`, `tez`, `timestamp`, `address`, etc. As an example of non-comparable types: maps, sets or lists are not comparable: if you wish to compare them, you will have to write your own comparison function.

Comparing Strings

- In PascaLIGO:
```
const a : string = "Alice"
const b : string = "Alice"
const c : bool = (a = b) // True
```

- In CameLIGO:
```
let a : string = "Alice"
let b : string = "Alice"
let c : bool = (a = b) // true
```

- In ReasonLIGO:
```
let a : string = "Alice";
let b : string = "Alice";
let c : bool = (a == b); // true
```

Comparing Numbers in PascaLIGO

```
const a : int  = 5
const b : int  = 4
const c : bool = (a = b)
const d : bool = (a > b)
const e : bool = (a < b)
const f : bool = (a <= b)
const g : bool = (a >= b)
const h : bool = (a =/= b)
```

Comparing Numbers in CameLIGO

```
let a : int  = 5
let b : int  = 4
let c : bool = (a = b)
let d : bool = (a > b)
let e : bool = (a < b)
let f : bool = (a <= b)
let g : bool = (a >= b)
let h : bool = (a <> b)
```

Comparing Numbers in ReasonLIGO

```
let a : int  = 5;
let b : int  = 4;
let c : bool = (a == b);
let d : bool = (a > b);
let e : bool = (a < b);
let f : bool = (a <= b);
let g : bool = (a >= b);
let h : bool = (a != b);
```

Tuples

- Tuples gather a given number of values in a specific order and those values, called **components**, can be retrieved by their index (position).
- Probably the most common tuple is the **pair**. For example, if we were storing coordinates on a two dimensional grid we might use a pair `(x,y)` to store the coordinates x and y.
- There is a **specific order**, so `(y,x)` is not equal to `(x,y)` in general.
- The number of components is part of the type of a tuple, so, for example, we cannot add an extra component to a pair and obtain a triple of the same type: `(x,y)` has always a different type from `(x,y,z)`, whereas `(y,x)` might have the same type as `(x,y)`.
- Like record fields, tuple components can be of arbitrary types.

Tuple Definition

- Unlike a record, tuple types do not have to be defined before they can be used. However below we will give them names by **type aliasing**.
- In PascaLIGO:
```
type full_name is string * string  // Alias
const full_name : full_name = ("Alice", "Johnson")
```
- In CameLIGO:
```
type full_name = string * string  // Alias
// Optional parentheses:
let full_name : full_name = ("Alice", "Johnson")
```
- In ReasonLIGO:
```
type full_name = (string, string);  // Alias
let full_name : full_name = ("Alice", "Johnson");
```

Accessing Tuple Components

- Accessing the components of a tuple in OCaml is achieved by pattern matching. LIGO currently supports tuple patterns only in the parameters of functions, not in pattern matching. However, we can access components by their position in their tuple, which cannot be done in OCaml. Components are zero-indexed, that is, the first has index 0.
- In PascaLIGO:
  ```
  const first_name : string = full_name.0
  ```
- In CameLIGO:
  ```
  let first_name : string = full_name.0
  ```
- In ReasonLIGO:
  ```
  let first_name : string = full_name[0];
  ```

- Lists are linear collections of elements of the same type. Linear means that, in order to reach an element in a list, we must visit all the elements before (sequential access).
- Elements can be repeated, as only their order in the collection matters. The first element is called the **head**, and the sub-list after the head is called the **tail**.
- For those familiar with algorithmic data structure, you can think of a list a **stack**, where the top is written on the left.
- Lists are needed when returning operations from a smart contract's main function.

Defining Lists

- In PascaLIGO:
```
const empty_list : list (int) = nil        // Or: list []
const my_list : list (int) = list [1; 2; 2] // The head is 1
```

- In CameLIGO:
```
let empty_list : int list = []
let my_list : int list = [1; 2; 2] // The head is 1
```

- In ReasonLIGO:
```
let empty_list : list (int) = [];
let my_list : list (int) = [1, 2, 2]; // The head is 1
```

Adding to Lists

- Lists can be augmented by adding an element before the head (or, in terms of stack, by **pushing an element on top**). This operation is usually called **consing** in functional languages.
- In PascaLIGO, the **cons operator** is infix and noted #. It is not symmetric: on the left lies the element to cons, and, on the right, a list on which to cons. (The symbol is helpfully asymmetric to remind you of that.)

```
const larger_list : list (int) = 5 # my_list // [5;1;2;2]
```

Adding to Lists

- In CameLIGO, the **cons operator** is infix and noted `::`. It is not symmetric: on the left lies the element to cons, and, on the right, a list on which to cons.
  ```
  let larger_list : int list = 5 :: my_list // [5;1;2;2]
  ```

- In ReasonLIGO, the **cons operator** is infix and noted "`, ...`". It is not symmetric: on the left lies the element to cons, and, on the right, a list on which to cons.
  ```
  let larger_list : list (int) = [5, ...my_list]; // [5,1,2,2]
  ```

- Sets are unordered collections of values of the same type, like lists are ordered collections. Like the mathematical sets and lists, sets can be empty and, if not, elements of sets in LIGO are **unique**, whereas they can be repeated in a list.
- In PascaLIGO, the notation for sets is similar to that for lists, except the keyword **set** is used before:
  ```
  const my_set : set (int) = set []
  ```
- In CameLIGO, the empty set is denoted by the predefined value `Set.empty`:
  ```
  let my_set : int set = Set.empty
  ```
- In ReasonLIGO, the empty set is denoted by the predefined value `Set.empty`:
  ```
  let my_set : set (int) = Set.empty;
  ```

Adding to Sets

- In PascaLIGO, the notation for non-empty sets follows that for lists:
```
const my_set : set (int) = set [3; 2; 2; 1]
```

- In CameLIGO, to add to a set, use the predefined function `Set.add`:
```
let my_set : int set =
  Set.add 3 (Set.add 2 (Set.add 2 (Set.add 1 (Set.empty : int set))))
```

- In ReasonLIGO, to add to a set, use the predefined function `Set.add`:
```
let my_set : set (int) =
  Set.add (3,
           Set.add (2,
                    Set.add (2,
                             Set.add (1, Set.empty : set (int)))));
```

Set Membership

- PascaLIGO features a special keyword **`contains`** that operates like an infix operator checking membership in a set:
  ```
  const contains_3 : bool = my_set contains 3
  ```

- In CameLIGO, the predefined predicate `Set.mem` tests for membership in a set as follows:
  ```
  let contains_3 : bool = Set.mem 3 my_set
  ```

- In ReasonLIGO, the predicate is `Set.mem` as well:
  ```
  let contains_3 : bool = Set.mem (3, my_set);
  ```

- In PascaLIGO, the predefined function `size` returns the number of elements in a given set as follows:
  ```
  const set_size : nat = Set.size (my_set)
  ```

- In CameLIGO, the predefined function `Set.size` returns the number of elements in a given set as follows:
  ```
  let set_size : nat = Set.size my_set
  ```

- In ReasonLIGO, the predefined function is `Set.size` too:
  ```
  let set_size : nat = Set.size (my_set);
  ```

Updating Sets in PascaLIGO

- In PascaLIGO, there are two ways to update a set, that is to add or remove from it. Either we create a new set from the given one, or we modify it in-place. First, let us consider the former way:

```
const larger_set  : set (int) = Set.add (4, my_set)
const smaller_set : set (int) = Set.remove (3, my_set)
```

- If we are in a block, we can use an instruction to modify the set bound to a given variable. This is called a **patch**. It is only possible to add elements by means of a patch, not remove any: it is the union of two sets.

```
function update (var s : set (int)) : set (int) is block {
 patch s with set [4; 7]
} with s

const new_set : set (int) = update (my_set)
```

Updating Sets in CameLIGO and ReasonLIGO

- In CameLIGO, we update a given set by creating another one, with or without some elements:

```
let larger_set  : int set = Set.add 4 my_set
let smaller_set : int set = Set.remove 3 my_set
```

- In ReasonLIGO, the update is similar to CameLIGO:

```
let larger_set  : set (int) = Set.add (4, my_set);
let smaller_set : set (int) = Set.remove (3, my_set);
```

Functions in PascaLIGO

- LIGO functions are the basic building block of contracts. For example, entry-points are functions.
- There are two ways in PascaLIGO to define functions: with or without a **block**.
- Blocks enable the sequential composition of instructions into an isolated scope. Each block needs to include at least one instruction.
  `block { a := a + 1 }`
- If we need a placeholder, we use the instruction **skip** which leaves the state unchanged. The rationale for **skip** instead of a truly empty block is that it prevents you from writing an empty block by mistake.
  `block { skip }`
- Blocks are more versatile than simply containing instructions: they can also include **declarations** of values, like so:
  `block { const a : int = 1 }`

Block-based Functions in PascaLIGO

- Functions in PascaLIGO are defined using the **function** keyword followed by their **name**, **parameters** and **return** type definitions.
- Here is how you define a basic function that computes the sum of two integers:

```
function add (const a : int; const b : int) : int is
  block {
    const c : int = a + b
  } with c
```

- The function body consists of two parts:
  1. `block { <instructions and declarations> }` is the logic of the function;
  2. `with <value>` is the value returned by the function.

Blockless Functions in PascaLIGO

- Functions that can contain all of their logic into a single **expression** can be defined without the need of a block:

```
// Bad! Empty block not needed!
function identity (const n : int) : int is block { skip } with n

// Better
function identity (const n : int) : int is n  // Blockless
```

- The value of the expression is implicitly returned by the function.

```
function add (const a: int; const b : int) : int is a + b
```

- Functions in CameLIGO are defined using the let keyword, like other values. The difference is that a succession of parameters is provided after the value name, followed by the return type. This follows OCaml syntax.
  ```
  let add (a : int) (b : int) : int = a + b
  ```

- CameLIGO is a little different from other syntaxes when it comes to function parameters. In OCaml, functions can only take one parameter. To get functions with multiple arguments like we are used to in imperative programming languages, a technique called currying is used.

Functions in CameLIGO

- Currying essentially translates a function with multiple arguments into a series of single argument functions, each returning a new function accepting the next argument until every parameter is filled. This is useful because it means that CameLIGO supports **partial application**.
- Currying is however **not** the preferred way to pass function arguments in CameLIGO. While this approach is faithful to the original OCaml, it is costlier in Michelson than naive function execution accepting multiple arguments. Instead, for most functions with more than one parameter, we should gather the arguments in a **tuple** and pass the tuple in as a single parameter.
- Here is how you define a basic function that accepts two integers and returns an integer as well:

```
let add (a, b : int * int) : int = a + b // Uncurried
let add_curry (a : int) (b : int) : int = add (a, b) // Curried
let increment (b : int) : int = add_curry 1 // Partial application
```

- The function body is a single expression, whose value is returned.

- Functions in ReasonLIGO are defined using the **let** keyword, like other values. The difference is that a tuple of parameters is provided after the value name, with its type, then followed by the return type.
- Here is how you define a basic function that sums two integers:

```
let add = ((a, b): (int, int)) : int => a + b;
```

- As in CameLIGO and with blockless functions in PascaLIGO, the function body is a single expression, whose value is returned.
- If the body contains more than a single expression, you use a **block** between braces:

```
let myFun = ((x, y) : (int, int)) : int =>
{
  let doubleX = x + x;
  let doubleY = y + y;
  doubleX + doubleY
};
```

Anonymous functions (a.k.a. lambdas)

- It is possible to define functions without assigning them a name. They are useful when you want to pass them as arguments, or assign them to a key in a record or a map.
- In PascaLIGO:

```
function increment (const b : int) : int is
  (function (const a : int) : int is a + 1) (b)
const a : int = increment (1); // a = 2
```

- In CameLIGO:

```
let increment (b : int) : int = (fun (a : int) -> a + 1) b
let a : int = increment 1 // a = 2
```

- In ReasonLIGO:

```
let increment = (b : int) : int => ((a : int) : int => a + 1) (b);
let a : int = increment (1); // a == 2
```

Anonymous functions (a.k.a. lambdas)

- If the example above seems contrived, here is a more common design pattern for lambdas: to be used as parameters to functions. Consider the use case of having a list of integers and mapping the increment function to all its elements.
- In PascaLIGO:

```
function incr_map (const l : list (int)) : list (int) is
  List.map (function (const i : int) : int is i + 1, l)
```

- In CameLIGO:

```
let incr_map (l : int list) : int list =
  List.map (fun (i : int) -> i + 1) l
```

- In ReasonLIGO:

```
let incr_map = (l : list (int)) : list (int) =>
  List.map ((i : int) => i + 1, l);
```

The unit Type

- The `unit` type in Michelson or LIGO is a predefined type that contains only one value that carries no information. It is used when no relevant information is required or produced.
- In PascaLIGO, the unique value of the `unit` type is **Unit**:

```
const n : unit = Unit // Or unit
```

- In CameLIGO, the unique value of the `unit` type is `()`, following the OCaml convention:

```
let n : unit = ()
```

- In ReasonLIGO, the unique value of the `unit` type is `()`, following the ReasonML convention:

```
let n : unit = ();
```

Functional Iterations over Lists

- A **functional iterator** is a function that traverses a data structure and calls in turn a given function over the elements of that structure to compute some value. (Another approach is possible in PascaLIGO: **loops**.)
- There are three kinds of functional iterations over LIGO lists:
  1. the **iterated operation**,
  2. the **mapped operation** (not to be confused with the *map data structure*),
  3. and the **folded operation**.

- The first, the **iterated operation**, is an iteration over the list with a unit return value. It is useful to enforce certain invariants on the element of a list, or fail. For example you might want to check that each value inside of a list is within a certain range, and fail otherwise.
- In PascaLIGO, the predefined functional iterator implementing the iterated operation over lists is called `List.iter`.
- In the following example, a list is iterated to check that all its elements (integers) are greater than 3:

```
function iter_op (const l : list (int)) : unit is
  block {
    function iterated (const i : int) : unit is
      if i > 3 then Unit else (failwith ("Below range.") : unit)
  } with List.iter (iterated, l)
```

- In CameLIGO, the predefined functional iterator implementing the iterated operation over lists is called `List.iter`.

```
let iter_op (l : int list) : unit =
  let predicate = fun (i : int) -> assert (i > 3)
  in List.iter predicate l
```

- In ReasonLIGO, the predefined functional iterator implementing the iterated operation over lists is also called `List.iter`:

```
let iter_op = (l : list (int)) : unit => {
  let predicate = (i : int) => assert (i > 3);
  List.iter (predicate, l);
};
```

Map Operations over Lists

- We may want to change all the elements of a given list by applying to them a function. This is called a **map operation**, not to be confused with the map data structure.
- In PascaLIGO, the predefined functional iterator implementing the map operation over lists is called `List.map`:

```
function increment (const i : int): int is i + 1
// Creates a new list with all elements incremented by 1
const plus_one : list (int) = List.map (increment, larger_list)
```

- In CameLIGO, the predefined functional iterator implementing the map operation over lists is called `List.map`:

```
let increment (i : int) : int = i + 1
// Creates a new list with all elements incremented by 1
let plus_one : int list = List.map increment larger_list
```

- In ReasonLIGO, the predefined functional iterator implementing the map operation over lists is called `List.map`:

```
let increment = (i : int) : int => i + 1;
// Creates a new list with all elements incremented by 1
let plus_one : list (int) = List.map (increment, larger_list);
```

Folded Operations over Lists

- A **folded operation** is the most general of iteration. The folded function takes two arguments: an **accumulator** and the structure **element** at hand, with which it then produces a new accumulator. This enables having a partial result that becomes complete when the traversal of the data structure is over.
- In PascaLIGO, the predefined functional iterator implementing the folded operation over lists is called `List.fold`:

```
function sum (const acc : int; const i : int): int is acc + i
const sum_of_elements : int = List.fold (sum, my_list, 0)
```

- In CameLIGO, the predefined functional iterator implementing the folded operation over lists is called `List.fold`:
```
let sum (acc, i: int * int) : int = acc + i
let sum_of_elements : int = List.fold sum my_list 0
```

- In ReasonLIGO, the name of the iterator is also `List.fold`:
```
let sum = ((result, i): (int, int)): int => result + i;
let sum_of_elements : int = List.fold (sum, my_list, 0);
```

Functional Iterations over Sets

- Like for lists, there are three kinds of functional iterations over LIGO sets:
  1. the **iterated operation**,
  2. the **mapped operation** (not to be confused with the *map data structure*),
  3. and the **folded operation**.

- In PascaLIGO, the predefined functional iterator implementing the iterated operation over sets is called `Set.iter`.
- In the following example, a set is iterated to check that all its elements (integers) are greater than 3:

```
function iter_op (const s : set (int)) : unit is
  block {
    function iterated (const i : int) : unit is
      if i > 3 then Unit else (failwith ("Below range.") : unit)
  } with Set.iter (iterated, s)
```

- In CameLIGO, the predefined functional iterator implementing the iterated operation over sets is called `Set.iter`.

```
let iter_op (s : int set) : unit =
  let predicate = fun (i : int) -> assert (i > 3)
  in Set.iter predicate s
```

- In ReasonLIGO, the predefined functional iterator implementing the iterated operation over sets is called `Set.iter` too.

```
let iter_op = (s : set (int)) : unit => {
  let predicate = (i : int) => assert (i > 3);
  Set.iter (predicate, s);
};
```

Folded Operations over Sets

- In PascaLIGO, the predefined functional iterator implementing the folded operation over sets is called `Set.fold`.
```
function sum (const acc : int; const i : int): int is acc + i
const sum_of_elements : int = Set.fold (sum, my_set, 0)
```

- In CameLIGO, the predefined fold over sets is called `Set.fold`:
```
let sum (acc, i : int * int) : int = acc + i
let sum_of_elements : int = Set.fold sum my_set 0
```

- In ReasonLIGO, the predefined fold over sets is called `Set.fold` as well:
```
let sum = ((acc, i) : (int, int)) : int => acc + i;
let sum_of_elements : int = Set.fold (sum, my_set, 0);
```

Variant Types

- A variant type is a user-defined or a built-in type (in case of options) that defines a type by cases, so a value of a variant type is either this, or that or... and nothing else. The simplest variant type is equivalent to the enumerated types found in Java.
- In PascaLIGO:

```
type coin is Head | Tail
const head : coin = Head // Equivalent to Head (Unit)
const tail : coin = Tail // Equivalent to Tail (Unit)
```

- In CameLIGO:

```
type coin = Head | Tail
let head : coin = Head
let tail : coin = Tail
```

- In ReasonLIGO:

```
type coin = Head | Tail;
let head : coin = Head;
let tail : coin = Tail;
```

Variant Types

- The names `Head` and `Tail` in the definition of the type `coin` are called **data constructors**, or **variants**.
- In general, it is interesting for variants to carry some information, and thus go beyond enumerated types.
- In the following, we show how to define different kinds of users of a system.

Variants in PascaLIGO

```
type id is nat

type user is
  Admin   of id
| Manager of id
| Guest

const u : user = Admin (1000n)
const g : user = Guest        // Equivalent to Guest (Unit)
```

```
type id = nat

type user =
  Admin   of id
| Manager of id
| Guest

let u : user = Admin 1000n
let g : user = Guest
```

Variants in ReasonLIGO

```
type id = nat;

type user =
| Admin   (id)
| Manager (id)
| Guest;

let u : user = Admin (1000n);
let g : user = Guest;
```

Conditionals

- Conditional logic enables forking the control flow depending on the state.
- In PascaLIGO:

```
type magnitude is Small | Large
function compare (const n : nat) : magnitude is
  if n < 10n then Small else Large
```

- In CameLIGO:

```
type magnitude = Small | Large
let compare (n : nat) : magnitude = if n < 10n then Small else Large
```

- In ReasonLIGO:

```
type magnitude = Small | Large;
let compare = (n : nat) : magnitude =>
  if (n < 10n) { Small; } else { Large; };
```

Conditionals

- When the branches of the conditional are not a single expression, as above, we need a block:

```
if x < y then
  block {
    const z : nat = x;
    x := y; y := z
  }
  else skip;
```

- As an exception to the rule, the blocks in a conditional branch do not need to be introduced by the keyword **block**, so, we could have written instead:

```
if x < y then {
  const z : nat = x;
  x := y; y := z
}
else skip;
```

Conditionals

- Like in OCaml, conditionals in CameLIGO can omit "**`else ()`**", yielding the *dangling else* parsing issue, which is solved by associating each **`else`** to the closest previous **`then`**.
- For example,

```
let iter_op (s : int set) : unit =
  let predicate = fun (i : int) ->
     if i <= 2 then failwith "Below range." // No else
  in Set.iter predicate s
```

- The same is possible in ReasonLIGO, but **not** in PascaLIGO.

Optional Values

- The `option` type is a predefined variant type that is used to express whether there is a value of some type or none.
- This is especially useful when calling a **partial function**, that is, a function that is not defined for some inputs.
- In that case, the value of the `option` type would be `None`, otherwise `Some (v)`, where v is some meaningful value **of any type**.

Optional Values

- An example in arithmetic is the division operation follows.
- In PascaLIGO:
```
function div (const a : nat; const b : nat) : option (nat) is
  if b = 0n then (None: option (nat)) else Some (a/b)
```
- In CameLIGO:
```
let div (a, b : nat * nat) : nat option =
  if b = 0n then (None: nat option) else Some (a/b)
```
- In ReasonLIGO:
```
let div = ((a, b) : (nat, nat)) : option (nat) =>
  if (b == 0n) { (None: option (nat)); } else { Some (a/b); };
```

- **Pattern matching** is similiar to the switch construct in Javascript, and can be used to route the program's control flow based on the value of a variant. Consider for example the definition of a function `flip` that flips a coin.
- In PascaLIGO:

```
type coin is Head | Tail

function flip (const c : coin) : coin is
  case c of
    Head -> Tail // Equivalent to Tail (Unit)
  | Tail -> Head // Equivalent to Head (Unit)
  end
```

Pattern Matching in CameLIGO and ReasonLIGO

- In CameLIGO:

```
type coin = Head | Tail

let flip (c : coin) : coin =
  match c with
    Head -> Tail
  | Tail -> Head
```

- In ReasonLIGO:

```
type coin = Head | Tail;

let flip = (c : coin) : coin =>
  switch (c) {
  | Head => Tail
  | Tail => Head
  };
```

Maps

- **Maps** are a data structure which associate values (called **keys**) of the same type to values of the same type. Together they make up a **binding**. The type of the keys must be **comparable**, in the Michelson sense.
- Here is how a custom map from addresses to a pair of integers is defined in PascaLIGO:
  ```
  type move is int * int
  type register is map (address, move)
  ```
- In CameLIGO:
  ```
  type move = int * int
  type register = (address, move) map
  ```
- In ReasonLIGO:
  ```
  type move = (int, int);
  type register = map (address, move);
  ```

- In PascaLIGO:

```
const moves : register =
  map [("tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx" : address) -> (1,2);
       ("tz1gjaF81ZRRvdzjobyfVNsAeSC6PScjfQwN" : address) -> (0,3)]
```

- In CameLIGO:

```
let moves : register =
  Map.literal [
    (("tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx" : address), (1,2));
    (("tz1gjaF81ZRRvdzjobyfVNsAeSC6PScjfQwN" : address), (0,3))]
```

- In ReasonLIGO:

```
let moves : register =
  Map.literal ([
    ("tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx" : address, (1,2)),
    ("tz1gjaF81ZRRvdzjobyfVNsAeSC6PScjfQwN" : address, (0,3))]);
```

- In PascaLIGO, we can use the postfix [] operator to read the move value associated to a given key (address here) in the register.

```
const my_balance : option (move) =
  moves [("tz1gjaF81ZRRvdzjobyfVNsAeSC6PScjfQwN" : address)]
```

- In CameLIGO:

```
let my_balance : move option =
  Map.find_opt
    ("tz1gjaF81ZRRvdzjobyfVNsAeSC6PScjfQwN" : address) moves
```

- In ReasonLIGO:

```
let my_balance : option (move) =
  Map.find_opt
    (("tz1gjaF81ZRRvdzjobyfVNsAeSC6PScjfQwN" : address), moves);
```

Accessing Map Bindings in PascaLIGO

Notice how the value we read is an optional value: this is to force the reader to account for a missing key in the map. This requires **pattern matching**.

```
function force (const key: address; const moves: register) : move is
  case moves[key] of
    Some (move) -> move
  | None -> (failwith ("No move.") : move)
  end
```

Accessing Map Bindings in CameLIGO and ReasonLIGO

- In CameLIGO:

```
let force (key, moves : address * register) : move =
  match Map.find_opt key moves with
    Some move -> move
  | None -> (failwith "No move." : move)
```

- In ReasonLIGO:

```
let force = ((key, moves) : (address, register)) : move => {
  switch (Map.find_opt (key, moves)) {
  | Some (move) => move
  | None => failwith ("No move.") : move
  }
};
```

- Given a map, we may want to add a new binding, remove one, or modify one by changing the value associated to an already existing key. We may even want to retain the key but not the associated value. All those operations are called **updates**.
- The values of a PascaLIGO map can be updated using the usual assignment syntax

```
<map variable>[<key>] := <new value>
```

- For instance:

```
function assign (var m : register) : register is
  block {
    m [("tz1gjaF81ZRRvdzjobyfVNsAeSC6PScjfQwN": address)] := (4,9)
  } with m
```

- If multiple bindings need to be updated, PascaLIGO offers a **patch** instruction for maps.

```
function assignments (var m : register) : register is
  block {
    patch m with map [
      ("tz1gjaF81ZRRvdzjobyfVNsAeSC6PScjfQwN" : address) -> (4,9);
      ("tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx" : address) -> (1,2)
    ]
  } with m
```

Updating Maps in CameLIGO and ReasonLIGO

- We can update a binding in a map in CameLIGO by means of the `Map.update` built-in function:

```
let assign (m : register) : register =
  Map.update
    ("tz1gjaF81ZRRvdzjobyfVNsAeSC6PScjfQwN" : address)
    (Some (4,9)) m
```

- Notice the optional value `Some (4,9)` instead of `(4,9)`. If we had use `None` instead, that would have meant that the binding is removed.
- In ReasonLIGO:

```
let assign = (m : register) : register =>
  Map.update
    (("tz1gjaF81ZRRvdzjobyfVNsAeSC6PScjfQwN" : address),
     Some ((4,9)), m);
```

Removing a Binding

- In PascaLIGO, there is a special instruction to remove a binding from a map.

```
function del (const key : address; var moves : register)
 : register is
  block {
    remove key from map moves
  } with moves
```

- In CameLIGO, we use the predefined function `Map.remove`:

```
let del (key, moves : address * register) : register =
  Map.remove key moves
```

- In ReasonLIGO, we use `Map.remove` too:

```
let delete = ((key, moves) : (address, register)) : register =>
  Map.remove (key, moves);
```

- There are three kinds of functional iterations over LIGO maps:
  1. the **iterated operation**,
  2. the **mapped operation** (not to be confused with the *map data structure*),
  3. and the **folded operation**.

- In PascaLIGO, the predefined functional iterator implementing the iterated operation over maps is called `Map.iter`. In the following example, the register of moves is iterated to check that the start of each move is above 3.

```
function iter_op (const m : register) : unit is
  block {
    function iter (const i : address; const j : move) : unit is
      if j.1 > 3 then Unit else (failwith ("Below range.") : unit)
  } with Map.iter (iter, m)
```

- In CameLIGO, the predefinded functional iterator implementing the iterated operation over maps is called `Map.iter`.

```
let iter_op (m : register) : unit =
  let predicate = fun (i,j : address * move) -> assert (j.0 > 3)
  in Map.iter predicate m
```

- In ReasonLIGO:

```
let iter_op = (m : register) : unit => {
  let predicate = ((i,j) : (address, move)) => assert (j[0] > 3);
  Map.iter (predicate, m);
};
```

- We may want to change all the bindings of a map by applying to them a function. This is called a **map operation**, not to be confused with the map data structure.
- In PascaLIGO, the predefined functional iterator implementing the map operation over maps is called `Map.map`:

```
function map_op (const m : register) : register is
  block {
    function incr (const i : address; const j : move) : move is
      (j.0, j.1 + 1);
  } with Map.map (incr, m)
```

- In CameLIGO, the predefined functional iterator implementing the map operation over maps is called `Map.map`:

```
let map_op (m : register) : register =
  let incr (i,j : address * move) = j.0, j.1 + 1
  in Map.map incr m
```

- In ReasonLIGO, the iterator is called `Map.map` too:

```
let map_op = (m : register) : register => {
  let incr = ((i,j): (address, move)) => (j[0], j[1] + 1);
  Map.map (incr, m);
};
```

Folded Operations over Maps

- In PascaLIGO, the predefined functional iterator implementing the folded operation over maps is called `Map.fold`:

```
function fold_op (const m : register) : int is block {
  function folded (const j: int; const cur: address*move) : int is
    j + cur.2.2
  } with Map.fold (folded, m, 5)
```

- In CameLIGO, the predefined functional iterator implementing the folded operation over maps is called `Map.fold`:

```
let fold_op (m : register) : register =
  let folded (i,j : int * (address * move)) = i + j.1.1
  in Map.fold folded m 5
```

- In ReasonLIGO, the iterator is called `Map.fold` too:

```
let fold_op = (m : register) : register => {
  let folded = ((i,j): (int, (address, move))) => i + j[1][1];
  Map.fold (folded, m, 5);
};
```

Big Maps

- Ordinary maps are fine for contracts with a finite lifespan or a bounded number of entries. If a map is meant to hold many entries, the cost of loading those into the environment each time a user executes the contract would eventually become too expensive were it not for **big maps**.
- In PascaLIGO:

```
type move is int * int
type register is big_map (address, move)
```

- In CameLIGO:

```
type move = int * int
type register = (address, move) big_map
```

- In ReasonLIGO:

```
type move = (int, int);
type register = big_map (address, move);
```

Big Maps

- Basically, change `map` into `big_map` for types and `Map` into `Big_map` in CameLIGO and ReasonLIGO.
- All syntaxes and predefined functions for maps apply to big maps.
- For loops over big maps, use the keyword **`big_map`** instead of **`map`**.

General Iteration in PascaLIGO

- General iteration in PascaLIGO takes the shape of general loops, which should be familiar to programmers of imperative languages as while loops.
- Those loops are of the form

  `while <condition> <block>`
- Their associated block is repeatedly evaluated until the condition becomes true, or never evaluated if the condition is false at the start. The loop never terminates if the condition never becomes true.
- Because we are writing smart contracts on Tezos, when the condition of a `while` loops fails to become true, the execution will run out of gas and stop with a failure anyway.
- Loops make sense only in PascaLIGO because the conditional expression needs to be mutated by the body of the loop.

- Here is how to compute the greatest common divisors of two natural numbers by means of Euclid's algorithm:

```
function gcd (var x : nat; var y : nat) : nat is block {
  if x < y then {
    const z : nat = x;
    x := y; y := z
  }
  else skip;
  var r : nat := 0n;
  while y =/= 0n block {
    r := x mod y;
    x := y;
    y := r
  }
} with x
```

- CameLIGO is a functional language where user-defined values are constant, therefore it makes no sense in CameLIGO to feature loops, which we understand as syntactic constructs where the state of a stopping condition is mutated, as with `while` loops in PascaLIGO.
- Instead, CameLIGO implements a **folded operation** by means of a predefined function named `Loop.fold_while`.
- It takes an initial value of a certain type, called an **accumulator**, and repeatedly calls a given function, called **folded function**, that takes that accumulator and returns the next value of the accumulator, until a condition is met and the fold stops with the final value of the accumulator.
- The iterated function needs to have a special type: if the type of the accumulator is `t`, then it must have the type `(bool * t)` (not simply `t`). It is the boolean value that denotes whether the stopping condition has been reached.

- Here is how to compute the greatest common divisors of two natural numbers by means of Euclid's algorithm:

```
let iter (x,y : nat * nat) : bool * (nat * nat) =
  if y = 0n then false, (x,y) else true, (y, x mod y)

let gcd (x,y : nat * nat) : nat =
  let x,y = if x < y then y,x else x,y in
  let x,y = Loop.fold_while iter (x,y)
  in x
```

- To ease the writing and reading of the iterated functions (here, `iter`), two predefined functions are provided: `Loop.continue` and `Loop.stop`:

```
let iter (x,y : nat * nat) : bool * (nat * nat) =
  if y = 0n then Loop.stop (x,y) else Loop.resume (y, x mod y)

let gcd (x,y : nat * nat) : nat =
  let x,y = if x < y then y,x else x,y in
  let x,y = Loop.fold_while iter (x,y)
  in x
```

General Iteration in ReasonLIGO

```
let iter = ((x,y) : (nat, nat)) : (bool, (nat, nat)) =>
  if (y == 0n) { Loop.stop ((x,y)); } else { Loop.resume ((y, x mod y)); }

let gcd = ((x,y) : (nat, nat)) : nat => {
  let (x,y) = if (x < y) { (y,x); } else { (x,y); };
  let (x,y) = Loop.fold_while (iter, (x,y));
  x
};
```

Bounded Loops

- In addition to general loops, PascaLIGO features a specialised kind of loop to iterate over bounded intervals.
- These loops are familiarly known as for loops and they have the form

  ```
  for <variable assignment> to <upper bound> <block>
  ```

  as found in imperative languages.
- Consider how to sum the natural numbers up to *n*:

  ```
  function sum (var n : nat) : int is block {
    var acc : int := 0;
    for i := 1 to int (n) block { acc := acc + i }
  } with acc
  ```

- (Please do not use that function: there exists a closed form formula.)

- PascaLIGO `for` loops can also iterate through the contents of a collection, that is, a list, a set or a map.
- This is done with a loop of the form
  `for <element var> in <collection type> <collection var> <block>`
  where `<collection type>` is any of the following keywords: **list**, **set** or **map**.
- Here is an example where the integers in a list are summed up.

```
function sum_list (var l : list (int)) : int is block {
  var total : int := 0;
  for i in list l block { total := total + i }
} with total
```

- Summing all the integers in a set:

```
function sum_set (var s : set (int)) : int is block {
  var total : int := 0;
  for i in set s block { total := total + i }
} with total
```

- Loops over maps are loops over the bindings noted (key -> value).

```
function sum_map (var m : map (string,int)) : string*int is block {
  var string_total : string := "";
  var int_total : int := 0;
  for key -> value in map m block {
    string_total := string_total ^ key;
    int_total := int_total + value
  }
} with (string_total, int_total)
```

Records

- Records are one way data of different types can be packed into a single type. A record is made of a set of **fields**, which are made of a **field name** and a **field type**. Given a value of a record type, the value bound to a field can be accessed by giving its field name to a special operator `(.)`.
- Record declaration in PascaLIGO:
  ```
  type user is record [id : nat; is_admin : bool; name : string]
  ```
- Record declaration in CameLIGO:
  ```
  type user = {id : nat; is_admin : bool; name : string}
  ```
- Record declaration in ReasonLIGO:
  ```
  type user = {id : nat, is_admin : bool, name : string};
  ```

- And here is how a record value is defined
- in PascaLIGO:
  ```
  const alice : user = record [id=1n; is_admin = True; name = "Alice"]
  ```
- in CameLIGO:
  ```
  let alice : user = {id = 1n; is_admin = true; name = "Alice"}
  ```
- in ReasonLIGO:
  ```
  let alice : user = {id : 1n, is_admin : true, name : "Alice"};
  ```

- If we want the contents of a given field, we use the special infix operator `(.)`
- In PascaLIGO:
```
const alice_admin : bool = alice.is_admin
```
- In CameLIGO:
```
let alice_admin : bool = alice.is_admin
```
- In ReasonLIGO:
```
let alice_admin : bool = alice.is_admin;
```

Functional Updates

- Given a record value, it is a common design pattern to update only a small number of its fields. Instead of copying the fields that are unchanged, LIGO offers a way to only update the fields that are modified.
- One way to understand the update of record values is the functional update.
- The idea is to have an expression whose value is the updated record. The shape of that expression is, in PascaLIGO,

  <record variable> with <record value>

  The record variable is the record to update and the record value is the update itself.
- In CameLIGO, the shape follows that of OCaml:

  {<record variable> with <field assignments>}

Functional Updates in PascaLIGO

- Let us consider defining a function that translates three-dimensional points on a plane.
- In PascaLIGO:

```
type point is record [x : int; y : int; z : int]
type vector is record [dx : int; dy : int]

const origin : point = record [x = 0; y = 0; z = 0]

function xy (var p : point; const vec : vector) : point is
  p with record [x = p.x + vec.dx; y = p.y + vec.dy]
```

- You have to understand that p has not been changed by the functional update: a namless new version of it has been created and returned by the blockless function.

Functional Updates in CameLIGO

- The syntax for the functional updates of record in CameLIGO follows that of OCaml:

```
type point = {x : int; y : int; z : int}
type vector = {dx : int; dy : int}

let origin : point = {x = 0; y = 0; z = 0}

let xy (p, vec : point * vector) : point =
  {p with x = p.x + vec.dx; y = p.y + vec.dy}
```

Functional Updates in ReasonLIGO

- The syntax for the functional updates of record in ReasonLIGO follows that of ReasonML:

```
type point = {x : int, y : int, z : int};
type vector = {dx : int, dy : int};

let origin : point = {x : 0, y : 0, z : 0};

let xy = ((p, vec) : (point, vector)) : point =>
  {...p, x : p.x + vec.dx, y : p.y + vec.dy};
```

- Another way to understand what it means to update a record value is to make sure that any further reference to the value afterwards will exhibit the modification.
- This kind of imperative update is called in LIGO a patch and this is only possible in PascaLIGO, because a patch is an instruction, therefore we can only use it in a block.
- Similarly to a functional update, a patch takes a record to be updated and a record with a subset of the fields to update, then applies the latter to the former (hence the name "patch").

Record Patches

Let us consider defining a function that translates three-dimensional points on a plane.

```
type point is record [x : int; y : int; z : int]
type vector is record [dx : int; dy : int]

const origin : point = record [x = 0; y = 0; z = 0]

function xy (var p : point; const vec : vector) : point is
  block {
    patch p with record [x = p.x + vec.dx];
    patch p with record [y = p.y + vec.dy]
  } with p
```

Of course, we can actually translate the point with only one **patch**, as the previous example was meant to show that, after the first patch, the value of p indeed changed. So, a shorter version would be

```
type point is record [x : int; y : int; z : int]
type vector is record [dx : int; dy : int]

const origin : point = record [x = 0; y = 0; z = 0]

function xy (var p : point; const vec : vector) : point is
  block {
    patch p with record [x = p.x + vec.dx; y = p.y + vec.dy]
  } with p
```

Record patches can actually be simulated with functional updates. All we have to do is **declare a new record value with the same name as the one we want to update** and use a functional update.

```
type point is record [x : int; y : int; z : int]
type vector is record [dx : int; dy : int]

const origin : point = record [x = 0; y = 0; z = 0]

function xy (var p : point; const vec : vector) : point is
  block {
    const p : point =
       p with record [x = p.x + vec.dx; y = p.y + vec.dy]
  } with p
```

The hiding of a variable by another (here p) is called **shadowing**.

Packing and Unpacking

- Michelson provides the `PACK` and `UNPACK` instructions for data serialization. The former converts Michelson data structures into a binary format, and the latter reverses that transformation. This functionality can be accessed from within LIGO.
- `PACK` and `UNPACK` are Michelson instructions that are intended to be used by people that really know what they are doing.
- There are several risks and failure cases, typically casting the result to the wrong type. Do not use the corresponding LIGO functions without doing your homework first.

- In PascaLIGO:

```
function id_string (const p : string) : option (string) is
  block {
    const packed : bytes = Bytes.pack (p)
  } with (Bytes.unpack (packed) : option (string))
```

- In CameLIGO:

```
let id_string (p : string) : string option =
  let packed : bytes = Bytes.pack p in
  (Bytes.unpack packed : string option)
```

- In ReasonLIGO:

```
let id_string = (p : string) : option (string) => {
  let packed : bytes = Bytes.pack (p);
  (Bytes.unpack (packed) : option (string));
};
```

Hashing Keys

- It is often desirable to hash a public key. In Michelson, certain data structures such as maps will not allow the use of the key type.
- Even if this were not the case, hashes are much smaller than keys, and storage on blockchains comes at a cost premium.
- You can hash keys with a predefined function returning a value of type `key_hash`.

- In PascaLIGO:

```
function check (const kh1 : key_hash; const k2 : key)
  : bool * key_hash is
  block {
    var ret : bool := False;
    var kh2 : key_hash := Crypto.hash_key (k2);
    if kh1 = kh2 then ret := True else skip
  } with (ret, kh2)
```

- In CameLIGO:

```
let check (kh1, k2 : key_hash * key) : bool * key_hash =
  let kh2 : key_hash = Crypto.hash_key k2 in
  if kh1 = kh2 then true, kh2 else false, kh2
```

- In ReasonLIGO:

```
let check = ((kh1, k2) : (key_hash, key)) : (bool, key_hash) => {
  let kh2 : key_hash = Crypto.hash_key (k2);
  if (kh1 == kh2) { (true, kh2); } else { (false, kh2); }
};
```

Checking Signatures

- Sometimes a contract will want to check that a message has been signed by a particular key.
- For example, a point-of-sale system might want a customer to sign a transaction so it can be processed asynchronously.
- You can do this in LIGO using the `key` and `signature` types.
- There is no way to **generate** a signed message in LIGO, because that would require storing a private key on-chain, at which point it is not... private anymore.

Checking Signatures

- In PascaLIGO:
```
function check_signature
    (const pub_key : key;
     const signed  : signature;
     const msg     : bytes) : bool
is Crypto.check (pub_key, signed, msg)
```

- In CameLIGO:
```
let check_signature (pub_key, signed, msg : key * signature * bytes)
  : bool = Crypto.check pub_key signed msg
```

- In ReasonLIGO:
```
let check_signature =
  ((pub_key, signed, msg) : (key, signature, bytes)) : bool =>
  Crypto.check (pub_key, signed, msg);
```

- Often you want to get the address of the contract being executed. You can do it with `Tezos.self_address`.
- Due to limitations in Michelson, `Tezos.self_address` in a contract is only allowed at the top-level. Using it in an embedded function will cause an error.
- In PascaLIGO:

```
const current_addr : address = Tezos.self_address
```

- In CameLIGO:

```
let current_addr : address = Tezos.self_address
```

- In ReasonLIGO:

```
let current_addr : address = Tezos.self_address;
```

- LIGO gives access to the timestamp of the block, as Michelson does.
- You can obtain the current “time” using a predefined value.
- In PascaLIGO:
  ```
  const today : timestamp = Tezos.now
  ```
- In CameLIGO:
  ```
  let today : timestamp = Tezos.now
  ```
- In ReasonLIGO:
  ```
  let today : timestamp = Tezos.now;
  ```

Timestamp Arithmetics

- In LIGO, timestamps can be added to integers, allowing you to set time constraints on your smart contracts. Consider the following scenario "In 24h".
- In PascaLIGO:

```
const today : timestamp = Tezos.now
const one_day : int = 86_400
const in_24_hrs : timestamp = today + one_day
const some_date : timestamp = ("2000-01-01T10:10:10Z" : timestamp)
const one_day_later : timestamp = some_date + one_day
```

- In CameLIGO:

```
let today : timestamp = Tezos.now
let one_day : int = 86_400
let in_24_hrs : timestamp = today + one_day
let some_date : timestamp = ("2000-01-01t10:10:10Z" : timestamp)
let one_day_later : timestamp = some_date + one_day
```

Timestamp Arithmetics

- In ReasonLIGO:

```
let today : timestamp = Tezos.now;
let one_day : int = 86_400;
let in_24_hrs : timestamp = today + one_day;
let some_date : timestamp = ("2000-01-01t10:10:10Z" : timestamp);
let one_day_later : timestamp = some_date + one_day;
```

- Consider now the scenario “24h ago”.
- In PascaLIGO:

```
const today : timestamp = Tezos.now
const one_day : int = 86_400
const in_24_hrs : timestamp = today - one_day
```

- In CameLIGO:

```
let today : timestamp = Tezos.now
let one_day : int = 86_400
let in_24_hrs : timestamp = today - one_day
```

- In ReasonLIGO:

```
let today : timestamp = Tezos.now;
let one_day : int = 86_400;
let in_24_hrs : timestamp = today - one_day;
```

Timestamps Arithmetics

- You can compare timestamps using the same comparison operators applying to numbers.
- In PascaLIGO:
  ```
  const not_tommorow : bool = (Tezos.now = in_24_hrs)
  ```
- In CameLIGO:
  ```
  let not_tomorrow : bool = (Tezos.now = in_24_hrs)
  ```
- In ReasonLIGO:
  ```
  let not_tomorrow : bool = (Tezos.now == in_24_hrs);
  ```

Addresses

- The type `address` in LIGO denotes Tezos addresses (tz1, tz2, tz3, KT1, ...). Currently, addresses are created by casting a string to the `address`. Beware of failures if the address is invalid. Consider the following examples.
- In PascaLIGO:

```
const my_account : address =
  ("tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx" : address)
```

- In CameLIGO:

```
let my_account : address =
  ("tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx" : address)
```

- In ReasonLIGO:

```
let my_account : address =
  ("tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx" : address);
```

Signatures

- The `signature` type in LIGO datatype is used for Tezos signatures (edsig, spsig). Signatures are created by casting a string. Beware of failure if the signature is invalid.
- In PascaLIGO:

```
const my_sig : signature =
  ("edsigthTzJ8X7MPmNeEwybRAvdxS1...D8C7" : signature)
```

- In CameLIGO:

```
let my_sig : signature =
   ("edsigthTzJ8X7MPmNeEwybRAvdxS1...D8C7" : signature)
```

- In ReasonLIGO:

```
let my_sig : signature =
  ("edsigthTzJ8X7MPmNeEwybRAvdxS1...D8C7" : signature);
```

- The key type in LIGO is used for Tezos public keys. Do not confuse them with map keys. Keys are made by casting strings. Beware of failure if the key is invalid.
- In PascaLIGO:
```
const my_key : key = ("edpkuBknW28nW72KG6Ro...DVC9yav" : key)
```
- In CameLIGO:
```
let my_key : key = ("edpkuBknW28nW72KG6Ro...DVC9yav" : key)
```
- In ReasonLIGO:
```
let my_key : key = ("edpkuBknW28nW72KG6Ro...DVC9yav" : key);
```

Inclusion of Contracts

- Let us say that we have a contract that is getting too big. If it has a modular structure, you might find it useful to use the `#include` statement to split the contract up over multiple files.
- Yes, this is the same preprocessing directive that C-like programming languages offer.
- We plan to implement a real module system in the future.

Main Functions

- A LIGO contract is made of a series of constant and function declarations. Only functions having a special type can be called when the contract is activated: we called them main functions.
- A main function takes two parameters, the contract parameter and the on-chain storage, and returns a pair made of a list of operations and a (new) storage.
- When the contract is originated (that is, deployed, in Tezos parlance), the initial value of the storage is provided. When a main function is later called, only the parameter is given, but the type of a main function contains both.
- The type of the contract parameter and of the storage are up to the contract designer, but the type of list operations is not.

Access Functions

- The return type of an acces function is as follows, assuming that the type `storage` has been defined elsewhere. (Note that you can use any type with any name for the storage.)
- In PascaLIGO:

```
type storage is ...  // Any name, any type
type return is list (operation) * storage
```

- In CameLIGO:

```
type storage = ...  // Any name, any type
type return = operation list * storage
```

- In ReasonLIGO:

```
type storage = ...;  // Any name, any type
type return = (list (operation), storage);
```

- The contract storage can only be modified by activating the main function. It is important to understand what that means.
- What it does **not** mean is that some global variable holding the storage is modified by the main function.
- Instead, what it **does** mean is that, given the state of the storage **on-chain**, a main function specifies how to create another state for it, depending on a parameter.

- Here is an example where the storage is a single natural number that is updated by the parameter.
- In PascaLIGO:

```
type parameter is nat
type storage is nat
type return is list (operation) * storage

function save (const action : parameter; const store : storage)
  : return is ((nil : list (operation)), action)
```

Access Functions

- In CameLIGO:

```
type parameter = nat
type storage = nat
type return = operation list * storage
let save (action, store: parameter * storage) : return =
  (([] : operation list), store)
```

- In ReasonLIGO:

```
type parameter = nat;
type storage = nat;
type return = (list (operation), storage);
let main = ((action, store): (parameter, storage)) : return =>
  (([] : list (operation)), store);
```

- In LIGO, the design pattern is to have **one** main function, called `main`, that dispatches the control flow according to its parameter. Those functions called for those actions are called **entrypoints**.
- As an analogy, in the C programming language, the `main` function is the unique main function and any function called from it would be an entrypoint.
- The parameter of the contract is then a variant type, and, depending on the constructors of that type, different functions in the contract are called. In other terms, the unique main function dispatches the control flow depending on a **pattern matching** on the contract parameter.
- In the following example, the storage contains a counter of type `nat` and a name of type `string`. Depending on the parameter of the contract, either the counter or the name is updated.

Entrypoints in PascaLIGO

```
type parameter is Action_A of nat | Action_B of string
type storage is record [counter : nat; name : string]
type return is list (operation) * storage

function entry_A (const n : nat; const store : storage) : return is
  ((nil : list (operation)), store with record [counter = n])

function entry_B (const s : string; const store : storage) : return is
  ((nil : list (operation)), store with record [name = s])

function main (const action : parameter;
               const store : storage) : return is
  case action of
    Action_A (n) -> entry_A (n, store)
  | Action_B (s) -> entry_B (s, store)
  end
```

Entrypoints in CameLIGO

```
type parameter =
  Action_A of nat
| Action_B of string

type storage = {counter : nat; name : string}
type return = operation list * storage

let entry_A (n, store : nat * storage) : return =
  ([] : operation list), {store with counter = n}

let entry_B (s, store : string * storage) : return =
  ([] : operation list), {store with name = s}

let main (action, store : parameter * storage) : return =
  match action with
    Action_A n -> entry_A (n, store)
  | Action_B s -> entry_B (s, store)
```

Entrypoints in ReasonLIGO

```
type parameter = | Action_A (nat) | Action_B (string);
type storage = {counter : nat, name : string};
type return = (list (operation), storage);

let entry_A = ((n, store) : (nat, storage)) : return =>
  (([] : list (operation)), {...store, counter : n});

let entry_B = ((s, store) : (string, storage)) : return =>
  (([] : list (operation)), {...store, name : s});

let main = ((action, store) : (parameter, storage)) : return =>
  switch (action) {
  | Action_A (n) => entry_A ((n, store))
  | Action_B (s) => entry_B ((s, store))
  };
```

A crowdfunding contract in PascaLIGO

- A simple crowdfunding contract can be called in three manners:
  1. a backer sends some funds before the deadline has passed, and only once;
  2. a backer claims their funds after the deadline has passed and the goal has not been reached;
  3. the owner withdraws the funds after the deadline has passed and the goal has been reached;
  4. all other cases are errors.
- First, we define the storage:

```
type storage is record
   owner    : address;
   goal     : tez;
   deadline : timestamp;
   backers  : map (address, tez);
   funded   : bool;          // Optional terminator semicolon
 end
```

```
type return is list (operation) * storage

function back (var action : unit; var store : storage) : return is
  block {
    if Tezos.now > store.deadline then
      failwith ("Deadline passed.")
    else case store.backers[Tezos.sender] of
            None -> store.backers[Tezos.sender] := amount
          | Some (x) -> failwith ("Already backed.")
          end
  } with ((nil : list (operation)), store)
```

```
function claim (var action : unit; var store : storage) : return is
  block {
    var op : list (operation) := nil;
    if Tezos.now <= store.deadline then failwith ("Too early.")
    else
      case store.backers[Tezos.sender] of
         None -> failwith ("Not a backer.")
      | Some (assets) ->
          if Tezos.balance >= store.goal or store.funded then
            failwith ("Goal reached: no refund.")
          else {
            const dest : contract (unit) =
               Tezos.get_contract (Tezos.sender);
            op := list [Tezos.transaction (Unit, assets, dest)];
            remove Tezos.sender from map store.backers }
      end
  } with (op, store)
```

```
function withdraw (var action : unit; var store : storage) : return is
  block {
    var op : list (operation) := nil;
    if Tezos.sender = store.owner then
      if Tezos.now >= store.deadline then
        if Tezos.balance >= store.goal then {
          const dest : contract (unit) = Tezos.get_contract (Tezos.sender)
          store.funded := True;
          op := list [Tezos.transaction (Unit, balance, dest)]
        }
        else failwith ("Below target.")
      else failwith ("Too early.")
    else failwith ("Not the owner.")
  } with (op, store)
```

- The Michelson generator of the LIGO compiler performs several kinds of optimisations.
- One of them is **inlining**, that is, the expansion of the body of a function at its call site (with its parameters also expanded with the arguments).
- Inlining is controlled by **attributes** of constant and function declarations.
- In PascaLIGO:

```
function fst (const p : nat * nat) : nat is p.0;
attributes ["inline"];

function main (const p : nat * nat; const s : nat * nat)
  : list (operation) * (nat * nat) is
  ((nil : list (operation)), (fst (p.0,p.1), fst (p.1,p.0)))
```

Inlining

- Without the inlining attribute:
  `$ ligo measure-contract src/test/contracts/inlining.ligo main`
  `170 bytes`

- With the inlining attribute:
  `$ ligo measure-contract src/test/contracts/inlining.ligo main`
  `66 bytes`