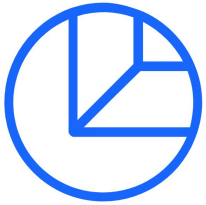


LIGOLANG

A MARIGOLD PROJECT

Writing a Simple Smart Contract

Suzanne Soy
ligo@suzanne.soy

ligolang.org

Why Tezos?

Proof of Stake

Consensus-driven protocol upgrades

Formal methods

Ecosystem redundancy

Why Tezos?

Proof of Stake

i.e. no palm trees at the North Pole

Consensus-driven protocol upgrades

i.e. no hard forks

Formal methods

i.e. fewer bugs

Ecosystem redundancy

i.e. don't put all your Tez in the same basket

Why LIGO?

Is Michelson suitable for application developers?

```
parameter unit ;
storage string ;
code {
        DROP ;
        PUSH string “Hello” ;
        NIL operation ;
        PAIR
}
```

Why LIGO?

Michelson is not very suitable for application developers!

Goal?

Reduce the barrier of entry and minimize risks of bugs

Why LIGO?

Michelson is not very suitable for application developers!

Goal?

Reduce the barrier of entry and minimize risks of bugs

Solution?

High-level language that compiles to Michelson with four syntaxes, to cater for different developers' taste.

LIGO Foundations

LIGO Foundation :: Syntaxes

Each syntax is as close as possible to the language it is inspired from

PascaLIGO a Pascal inspired syntax which provides an imperative developer experience.

Imperative paradigm describes the operations in sequences of instructions executed to change the program's state.

LIGO Foundation :: Syntaxes

Each syntax is as close as possible to the language it is inspired from

PascaLIGO a Pascal inspired syntax which provides an imperative developer experience.

CameLIGO an OCaml inspired syntax that allows you to write in a functional style.

Imperative paradigm describes the operations in sequences of instructions executed to change the program's state.
Functional paradigm considers computation as an evaluation of mathematical functions

LIGO Foundation :: Syntaxes

Each syntax is as close as possible to the language it is inspired from

PascaLIGO	a Pascal inspired syntax which provides an imperative developer experience.
CameLIGO	an OCaml inspired syntax that allows you to write in a functional style.
ReasonLIGO	a ReasonML inspired syntax that allows you to write in a functional style.

Imperative paradigm describes the operations in sequences of instructions executed to change the program's state.
Functional paradigm considers computation as an evaluation of mathematical functions

LIGO Foundation :: Syntaxes

Each syntax is as close as possible to the language it is inspired from

PascaLIGO a Pascal inspired syntax which provides an imperative developer experience.

CameLIGO an OCaml inspired syntax that allows you to write in a functional style.

Formal methods

ReasonLIGO a ReasonML inspired syntax that allows you to write in a functional style.

JsLIGO a Javascript inspired syntax which provides an imperative developer experience.

Imperative paradigm describes the operations in sequences of instructions executed to change the program's state.
Functional paradigm considers computation as an evaluation of mathematical functions

LIGO Foundation :: Syntaxes

Each syntax is as close as possible to the language it is inspired from

PascaLIGO a Pascal inspired syntax which provides an imperative developer experience.

CameLIGO an OCaml inspired syntax that allows you to write in a functional style.

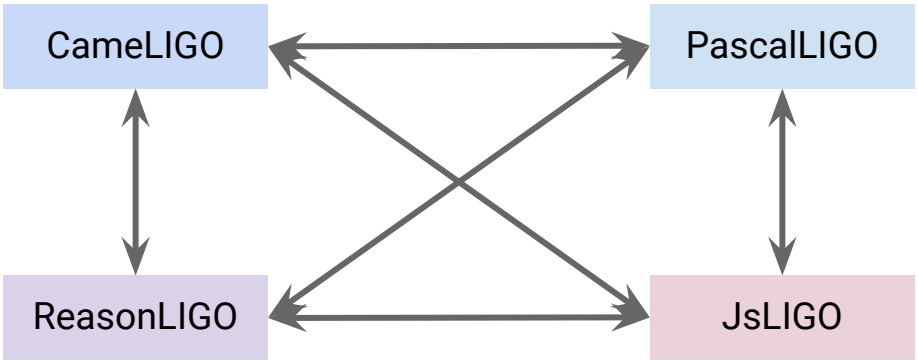
Formal methods

ReasonLIGO a ReasonML inspired syntax that allows you to write in a functional style.

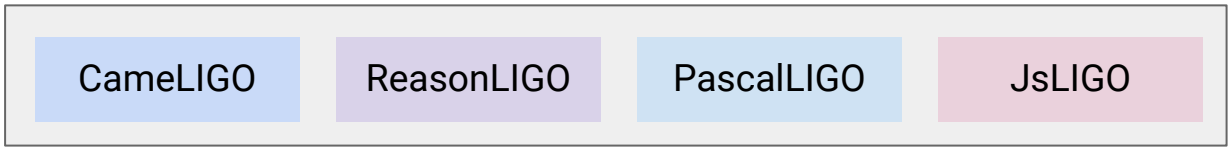
JsLIGO a Javascript inspired syntax which provides an imperative developer experience.

Imperative paradigm describes the operations in sequences of instructions executed to change the program's state.
Functional paradigm considers computation as an evaluation of mathematical functions

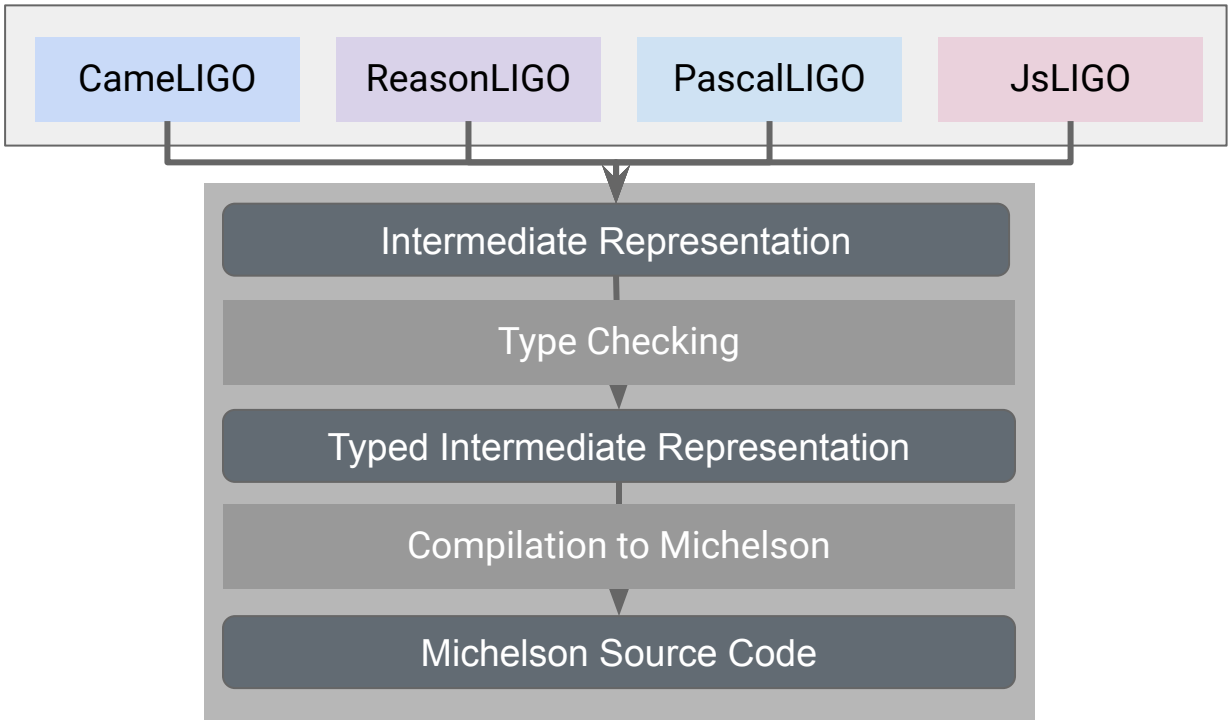
LIGO Foundation :: Interoperability



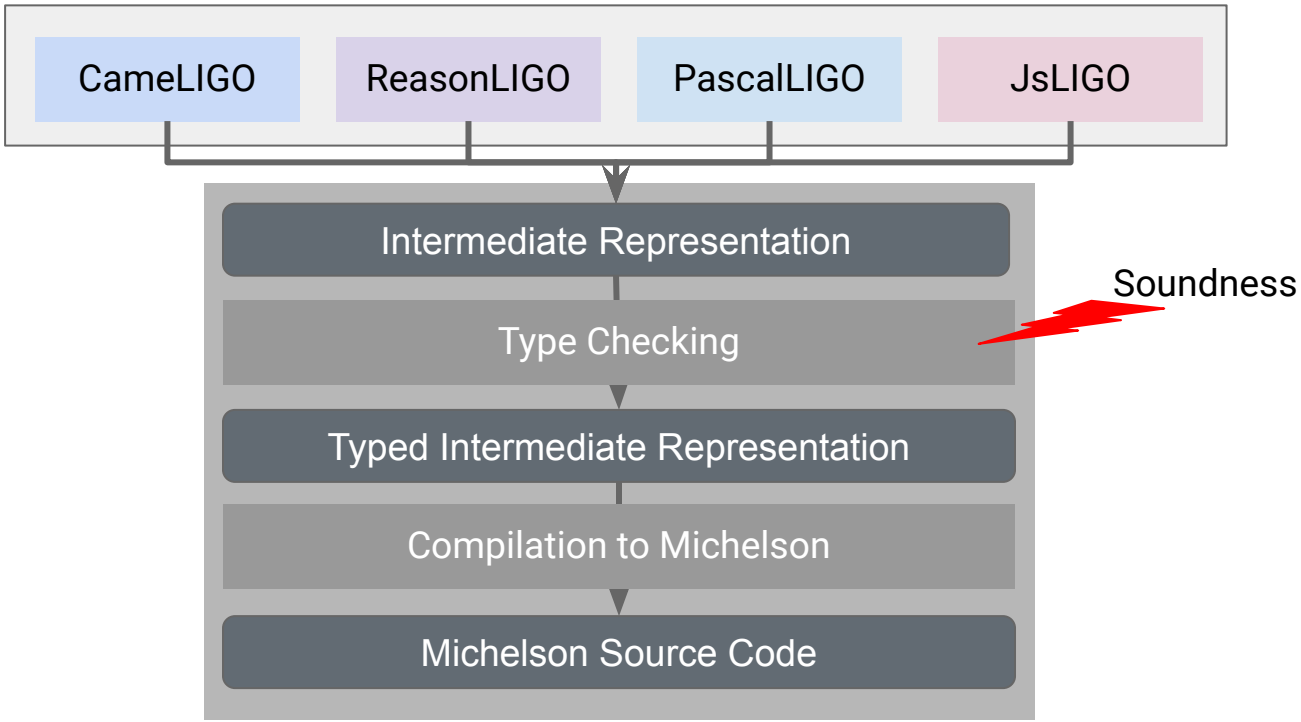
LIGO Foundation :: Compiler Overview



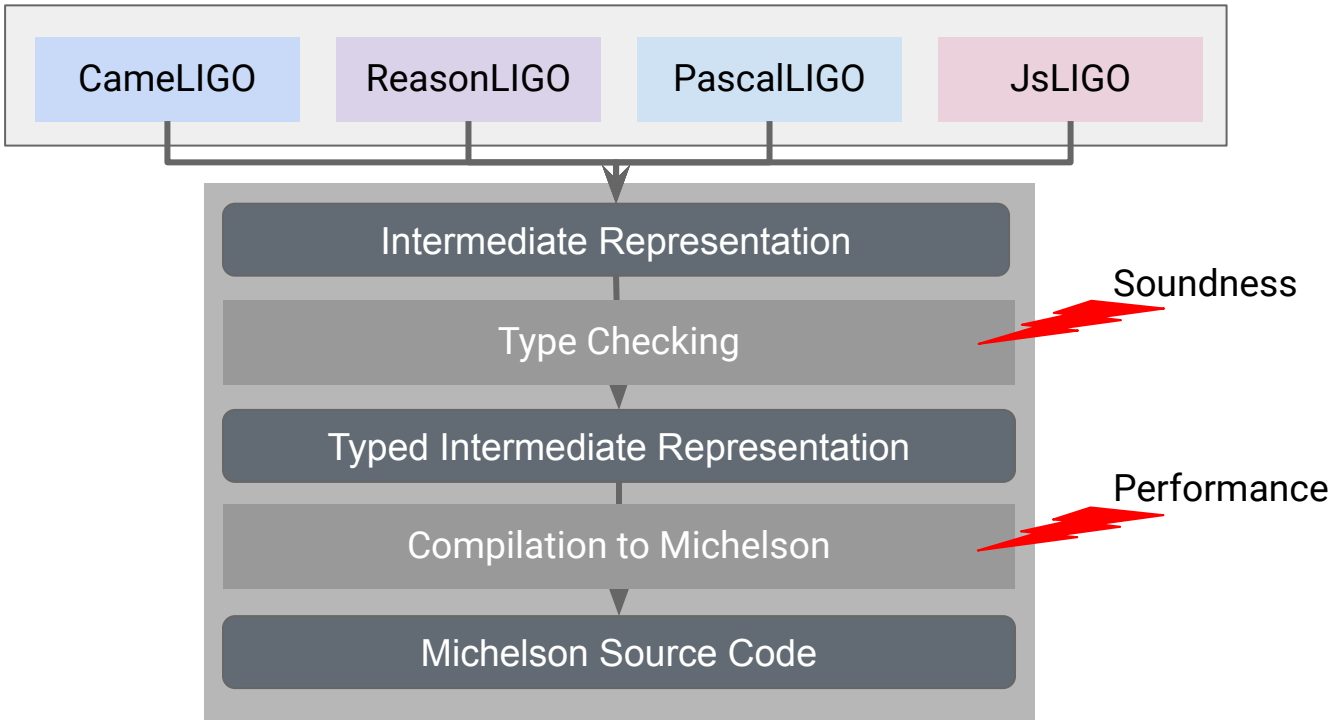
LIGO Foundation :: Compiler Overview



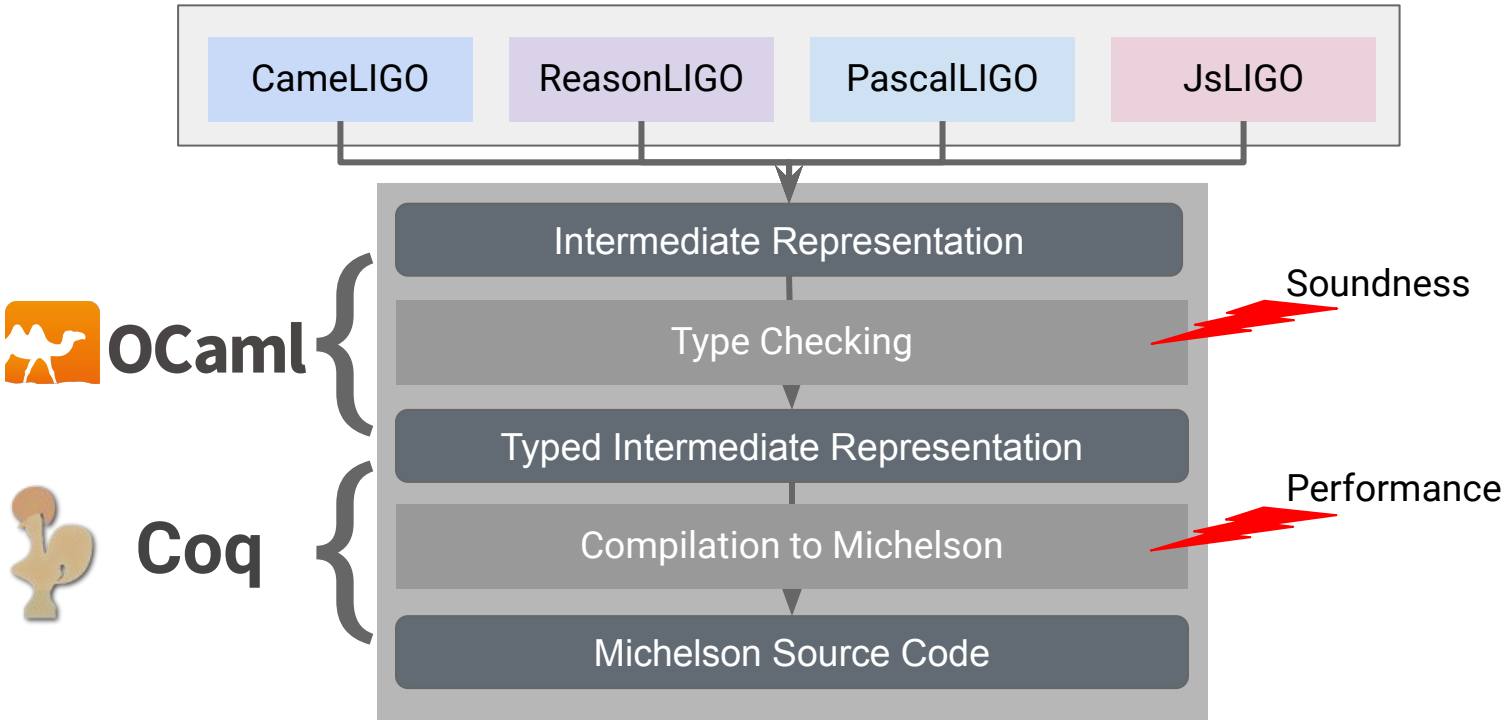
LIGO Foundation :: Compiler Overview



LIGO Foundation :: Compiler Overview



LIGO Foundation :: Compiler Overview



Contract Anatomy

Contract Anatomy :: Storage

```
type storage = int;
```

Contract Anatomy :: Parameter

```
type storage = int;
type parameter =
  ["Increment", int]
| ["Decrement", int]
| ["Reset"];
```

Parameter

Storage

Contract Anatomy :: Operations

```
type storage = int;
type parameter =
  ["Increment", int]
| ["Decrement", int]
| ["Reset"];
type result_ = [list<operation>, storage]
```

Parameter

Operations

Storage

Contract Anatomy :: Behavior

```
type storage = int;
type parameter =
  ["Increment", int]
| ["Decrement", int]
| ["Reset"];
type result_ = [list<operation>, storage]

let main = ([action, store]: [parameter, storage]) : result_ => {
  return [
    list([]) as list<operation>,
    match(action, {
      Increment: (n: int) => store + n,
      Decrement: (n: int) => store - n,
      Reset:     ()       => 0
    })
  ];
};
```

JsLIGO Overview

JsLIGO Overview :: Types

Native types	int, nat, **tez**, string, bytes…	// 1 “Hello” …

JsLIGO Overview :: Types

Native types	int, nat, **tez**, string, bytes…	// 1 “Hello” …
Tuples	[int, string, tez]	// (1, “Hello”, 42 as tez)

JsLIGO Overview :: Types

Native types	int, nat, **tez**, string, bytes…	// 1 “Hello” …
Tuples	[int, string, tez]	// (1, “Hello”, 42 as tez)
Records	{ name: string }	// { name: John }

JsLIGO Overview :: Types

Native types	int, nat, **tez**, string, bytes…	// 1 “Hello” …
Tuples	[int, string, tez]	// (1, “Hello”, 42 as tez)
Records	{ name: string }	// { name: John }
Variants	[“Foo”] | [“Bar”, nat]	// [“Bar”, 1 as nat]

JsLIGO :: Type Examples

```
type buy = {
     profile: bytes,
     initial_controller: option<address>
};

type update = {
     id: int,
     new_owner: address
};

type action = | ["Buy", buy]
              | ["Update", update];
```

JsLIGO Overview :: Constants & Variables

Constants

```
let x = (a : int) : int => {
  const age : int = 25;
  return age
};
```

Variables

```
let add = (a: int, b: int): int => {
  let c = a;
  c = c + b;
  return c
}
```

JsLIGO Overview :: Case Matching

```
type action = | ["Buy", buy] | ["Update", update];

// …

let main = ([act, storage]: [action, storage]):[list<operation>, storage] => {
  return match (act, {
    Buy: b => buy([b, storage]),
    Update: u => update_owner([u, storage]),
  });
};
```

JsLIGO Overview :: Iteration

Terminal Recursion (Functional style)

```
let sum_list = ([l, acc]: [list<int>, int]): int => {
  return match(l, list([
    ([] : list<int>)           => acc,
    ([hd, ...tl] : list<int>) => sum_list (tl, hd + acc)
  ]));
}
```

JsLIGO Overview :: Iteration

For Loop (Imperative style)

```
let sum_list = (l: list<int>): int => {
  let total : int = 0;
  for (const i of l) {
    total = total + i
  }
  return total
}
```

JsLIGO Overview :: Namespace (Module)

```
namespace EURO {
  export type t = nat;
  export let add = ([a, b]: [t, t]): t => a + b;
  export let zero: t = 0 as nat;
  export let one: t = 1 as nat
}
```

JsLIGO Overview :: Namespace (Module)

```
#import "euro.jsligo" "EURO"

type storage = EURO.t;
type return_ = [list<operation>, storage];

let main = ([action, store]: [unit, storage]): return_ =>
 [list([]) as list (operation),
  EURO.add(store, EURO.one)]
```

JsLIGO Overview :: Namespace (Module)

```
#import "euro.jsligo" "EURO"

type storage = EURO.t;
type return_ = [list<operation>, storage];

let main = ([action, store]: [unit, storage]): return_ =>
 [list([]) as list (operation),
  EURO.add(store, EURO.one)]
```

`#import` and syntaxes interoperability

Example Contract

Example Contract :: Behavior

```
type storage = int;
type parameter =
  ["Increment", int]
| ["Decrement", int]
| ["Reset"];
type result_ = [list<operation>, storage]

let main = ([action, store]: [parameter, storage]) : result_ => {
  return [
    list([]) as list<operation>,
    match(action, {
      Increment: (n: int) => store + n,
      Decrement: (n: int) => store - n,
      Reset:     ()       => 0
    })
  ];
};
```

Testing a Contract

Testing a Contract :: Framework

Tests are written in LIGO and work directly with LIGO code

Extra types and operations for manipulating Tezos context

Mutation testing primitives

Generation of random values for a type

Ability to catch errors in transactions

Testing a Contract :: Example

```
#import "contract.jsligo" "Contract"

let _test_increment = () : bool => {




}

let test_increment = _test_increment();
```

Testing a Contract :: Example

```
#import "contract.jsligo" " Contract"

let _test_increment = () : bool => {
   let initial_storage = 42 as int;
   let [address, _, _] = Test.originate(Contract.main, initial_storage, 0 as tez);



}

let test_increment = _test_increment();
```

Testing a Contract :: Example

```
#import "contract.jsligo" "Contract"

let _test_increment = () : bool => {
   let initial_storage = 42 as int;
   let [address, _, _] = Test.originate(Contract.main, initial_storage,  0 as tez);
   let contract = Test.to_contract(address);


}

let test_increment = _test_increment();
```

Testing a Contract :: Example

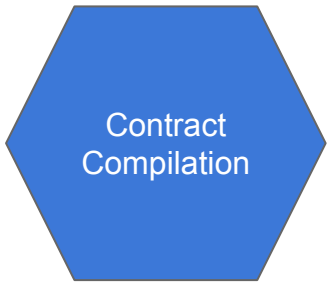
```
#import "contract.jsligo" "Contract"

let _test_increment = () : bool => {
   let initial_storage = 42 as int;
   let [address, _, _] = Test.originate(Contract.main, initial_storage,  0 as tez);
   let contract = Test.to_contract(address);
   let r = Test.transfer_to_contract_exn(contract, (Increment (1)), 1 as mutez);

}

let test_increment = _test_increment();
```

Testing a Contract :: Example

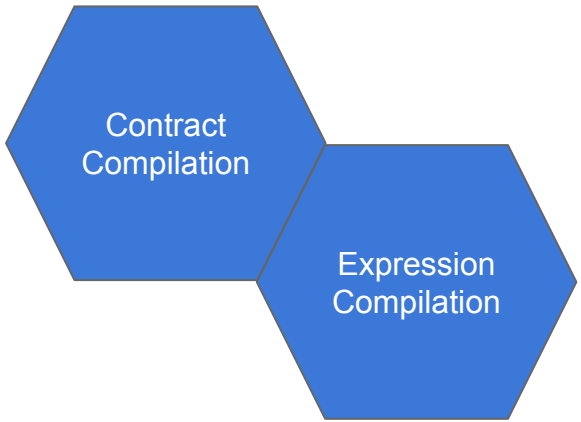
```
#import "contract.jsligo" "Contract"

let _test_increment = () : bool => {
   let initial_storage = 42 as int;
   let [address, _, _] = Test.originate(Contract.main, initial_storage,  0 as tez);
   let contract = Test.to_contract(address);
   let r = Test.transfer_to_contract_exn(contract, (Increment (1)), 1 as mutez);
   return (Test.get_storage(address) == initial_storage +  1);
}

let test_increment = _test_increment();
```

LIGO Tools

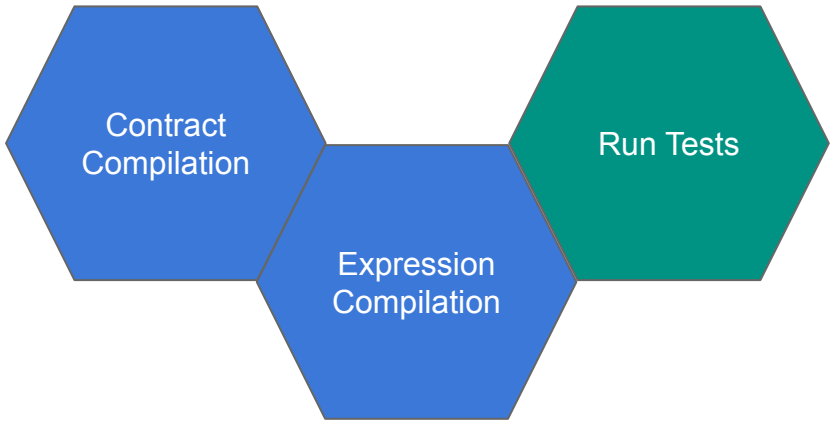
LIGO Tools :: CLI



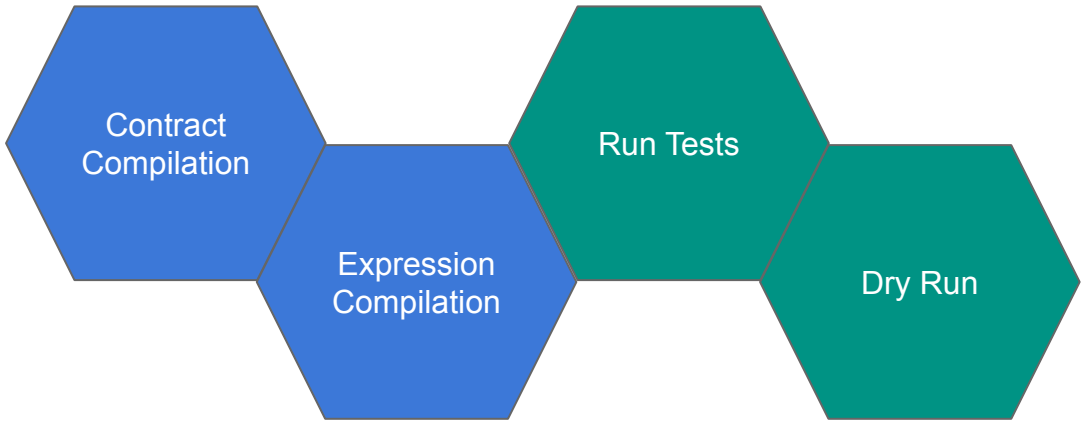
LIGO Tools :: CLI



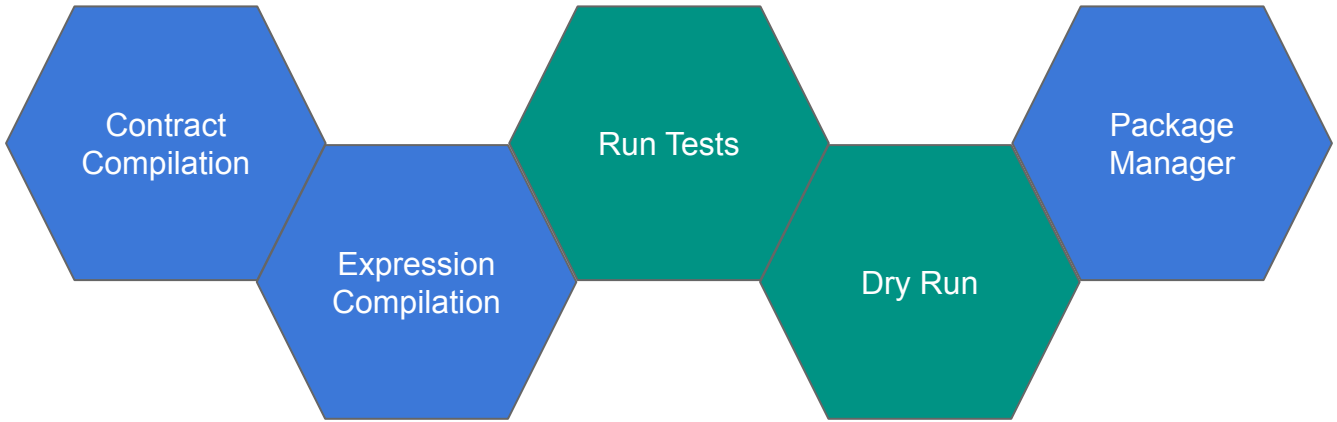
LIGO Tools :: CLI



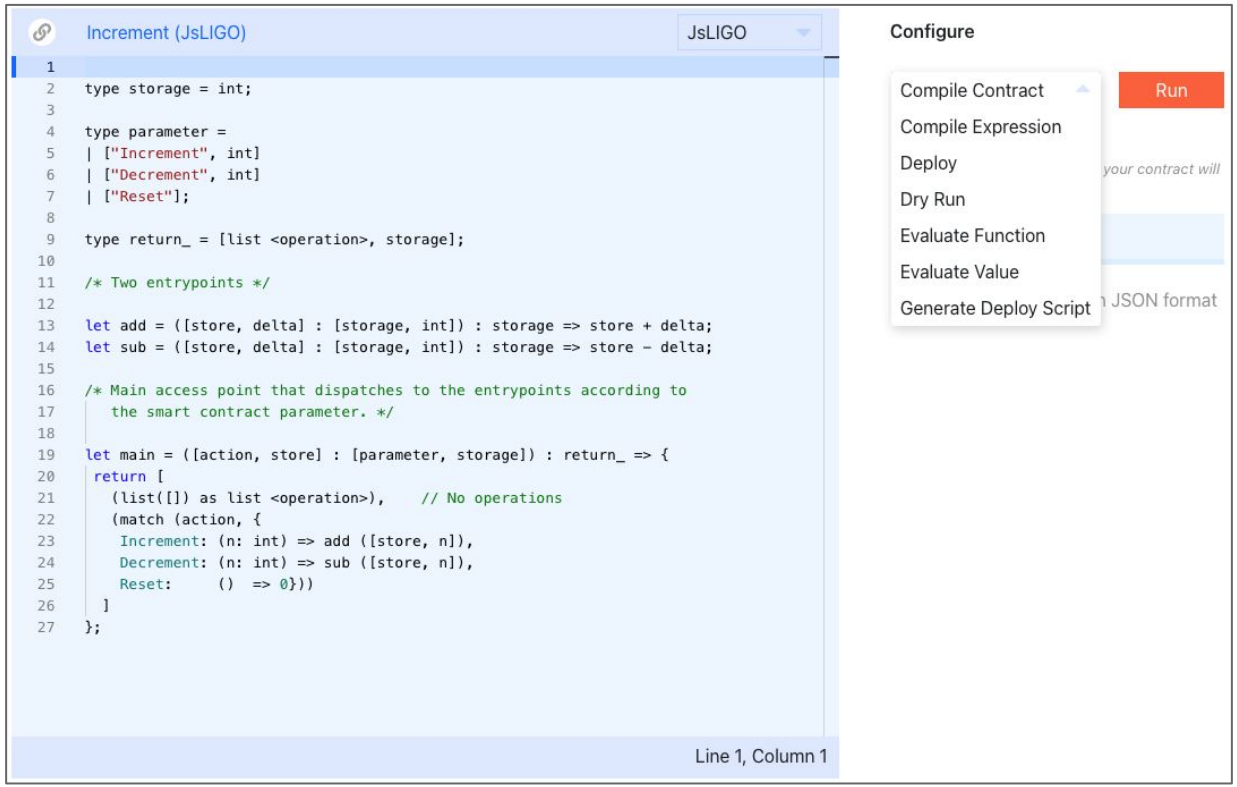
LIGO Tools :: CLI



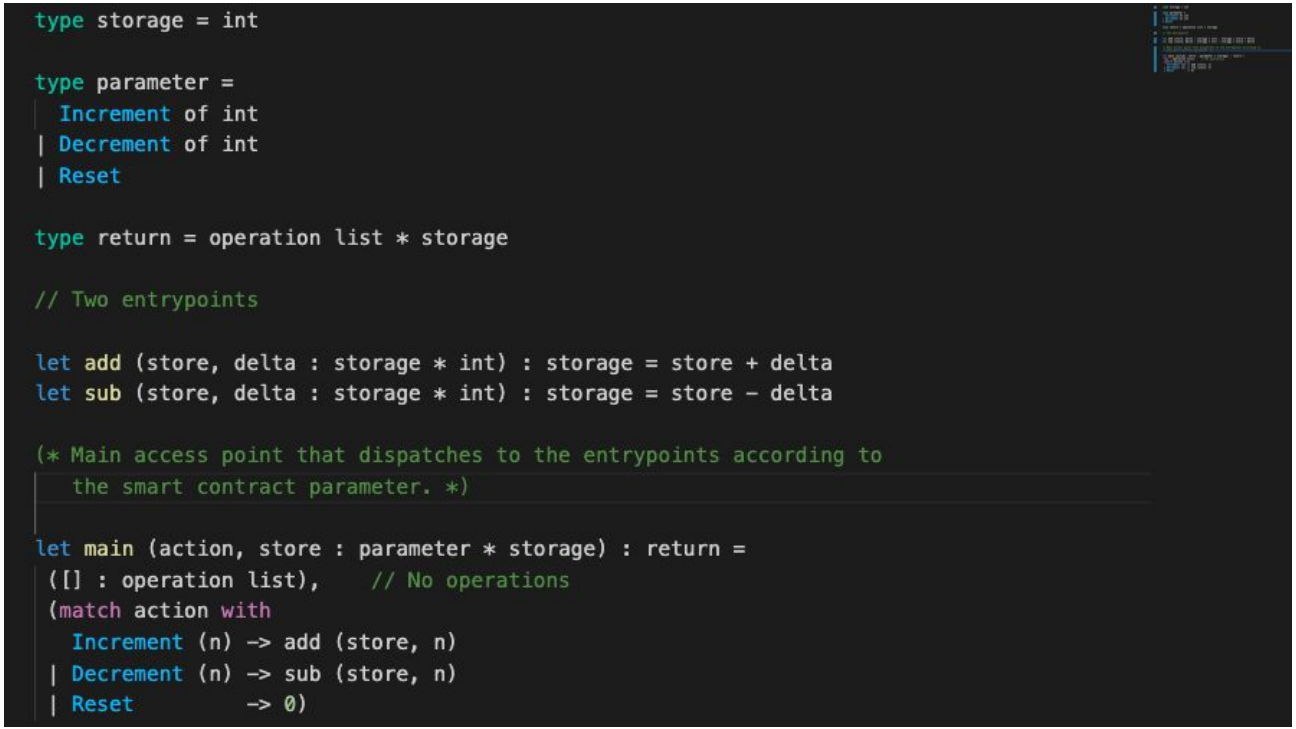
LIGO Tools :: CLI



LIGO Tools :: Web IDE (ide.ligolang.org)

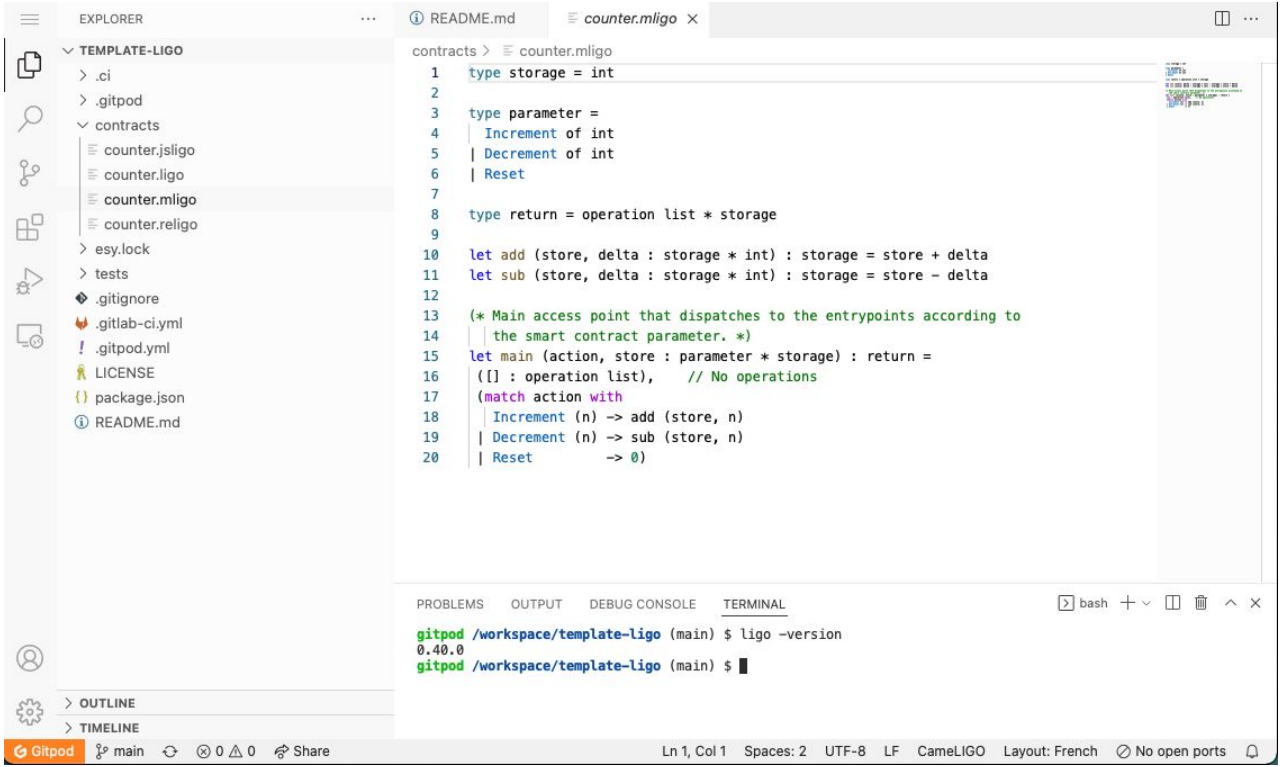


LIGO Tools :: VSCode



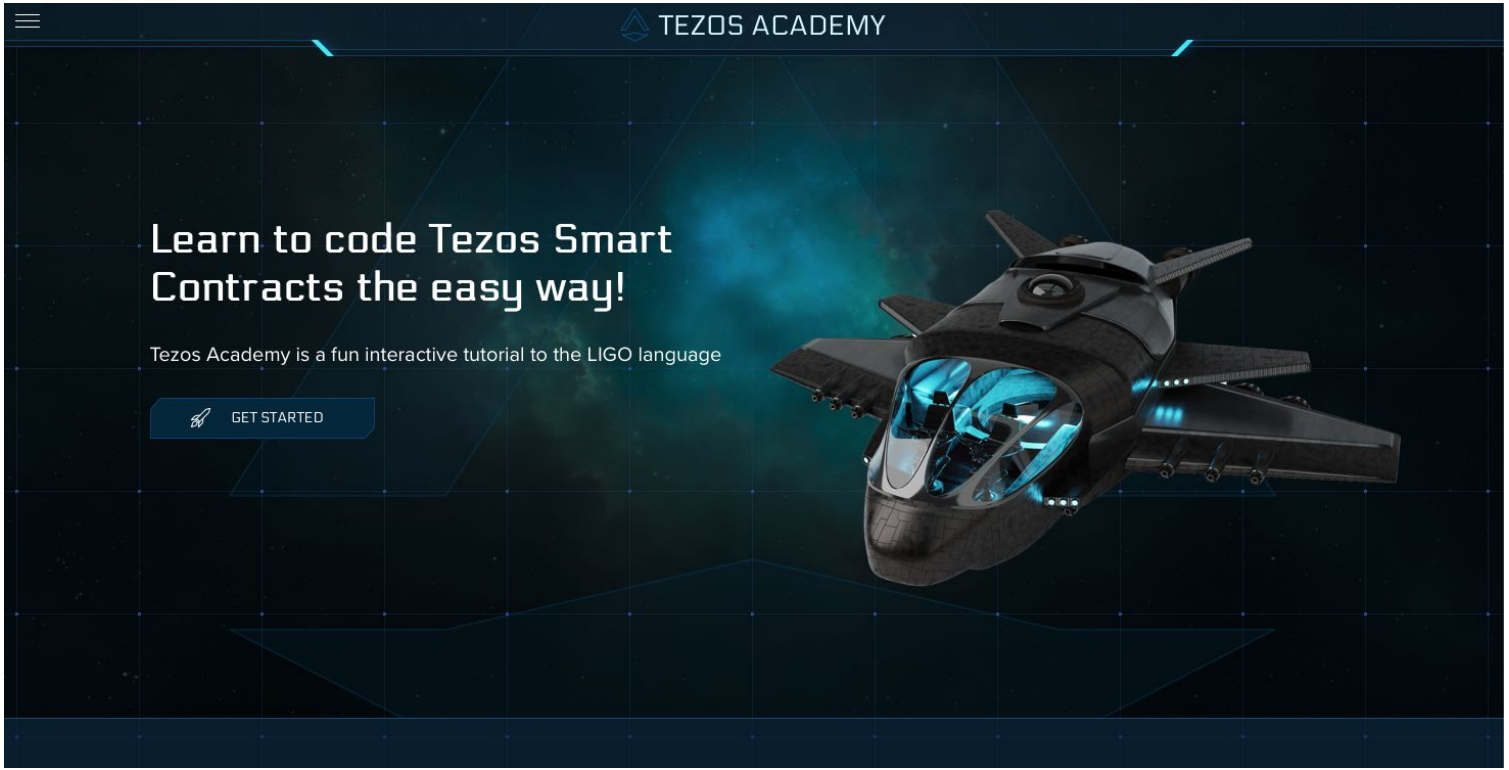
```
type storage = int

type parameter =
  Increment of int
| Decrement of int
| Reset

type return = operation list * storage

// Two entrypoints

let add (store, delta : storage * int) : storage = store + delta
let sub (store, delta : storage * int) : storage = store - delta

(* Main access point that dispatches to the entrypoints according to
   the smart contract parameter. *)

let main (action, store : parameter * storage) : return =
 ([] : operation list),    // No operations
 (match action with
   Increment (n) -> add (store, n)
 | Decrement (n) -> sub (store, n)
 | Reset         -> 0)
```

LIGO Tools :: Gitpod



LIGO Tools :: And more ...

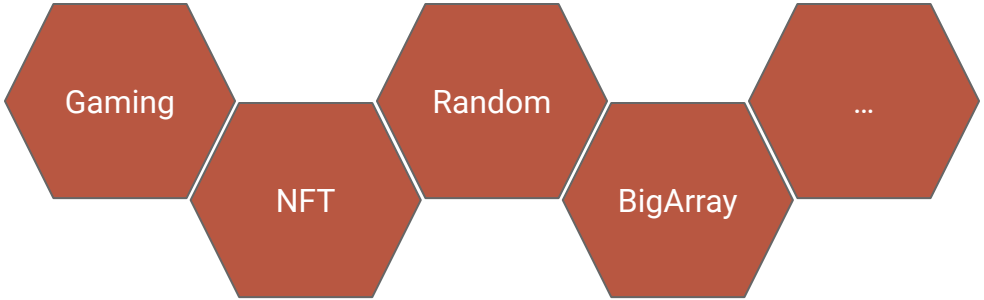
LIGO Tools :: Tutorial (academy.ligolang.org)



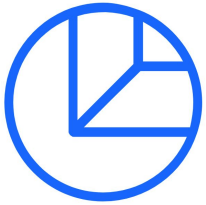
LIGO Catalog

LIGO Catalog :: Contract Library (Soon)

Smart contracts templates + Documentations



ligolib.org



LIGOLANG

A MARIGOLD PROJECT

“A programming language for writing Tezos smart contracts”

ligolang.org