

# Ligo Formal Description

## Syntax

The following describe the syntax of the simplify AST which is an internal of LIGO. The concrete syntax will be different depending of the choosen one but all the caterogies are present and the corresponding evaluation are the same

A LIGO program is a succession of declarations and expressions. Declarations add bindings to the environment while expressions are evaluated and yield values

*variables* ( $x, X$ )

Variable are the key to access elements in the environments. They are store with element they refers to at creation.

*label* ( $l$ )

Label identifies filed in records

*constructor* ( $C$ )

Constructor creates custom type's literals for built-in types literals

*declaration* ( $d$ ) =

<i>type</i> $X$ <i>is</i> $te$	(Type variable declaration)
<i>const</i> $x$ ( $:$ $te$ )? = $e$	(Term Constant variable declaration)
<i>var</i> $x$ ( $:$ $te$ )? = $e$	(Term Mutable variable declaration)

declaration adds binding to the environment i.e. a pair of the variable and the expression it corresponds to

*type expression* ( $te$ ) =

$te$ ( $*$ $te_i$ ) $+$	(type of tuple)
( $ $ $C_i$ <i>of</i> $te_i$ )	(type of sum)
{ $l_i$ : $te_i$ }	(type of record)
$te1 \rightarrow te2$	(type of function)
$X$	(type variable)
<i>Operator</i> ( $te_i$ )	(built in function on type)

The above describes all expression that are valid at the level of types

<i>term expression</i> ( <i>e</i> ) =	
<i>value</i>	(values)
<i>built-in</i> ( <i>e<sub>i</sub></i> )	(built-in function)
<i>x</i>	(variables)
<i>λx . expr</i>	(lambda abstraction)
<i>e<sub>1</sub> e<sub>2</sub></i>	(application)
<i>let x = e<sub>1</sub> in e<sub>2</sub></i>	(let in)
( <i>e<sub>i</sub></i> )	(tuple)
<i>C e</i>	(constructor)
{ <i>l<sub>i</sub> = e<sub>i</sub></i> }	(record)
[ <i>e<sub>1i</sub> = e<sub>2i</sub></i> ]	(map)
[[ <i>e<sub>1i</sub> = e<sub>2i</sub></i> ]]	(big map)
[ <i>e<sub>i</sub></i> ]	(list)
{ <i>e<sub>i</sub></i> }	(set)
<i>e</i> ( <i>a<sub>i</sub></i> )	(accessor)
<i>e<sub>1</sub>[e<sub>2</sub>]</i>	(look up)
<i>match e with matching</i>	(matching)
<i>e<sub>1</sub>; e<sub>2</sub></i>	(sequence)
<i>while e<sub>1</sub> do e<sub>2</sub></i>	(loop)
<i>x</i> ( <i>a<sub>i</sub></i> ) = <i>e</i>	(assign)
<i>SKIP</i>	(skip)
<i>e as T</i>	(ascription)

The above describes all expression that are valid at the level of terms

<i>value</i> ( <i>v</i> ) =	
<i>literal</i>	(values of built-in types)
<i>C v</i>	(values of construct types)
<i>λx . expr</i>	(lambda abstraction)

Values are valid termination of expression, it can be a built-in or construct literal or an abstraction

<i>literal</i> =	
<i>unit</i>	()
<i>bool</i>	()
<i>int</i>	()
<i>nat</i>	()
<i>mutez</i>	()
<i>string</i>	()
<i>bytes</i>	()
<i>address</i>	()
<i>timestamp</i>	()
<i>operation</i>	()

the above lists all predefined literals supported by ligos

<i>accessor</i> ( <i>a</i> ) =	
<i>i</i> ( <i>natural number</i> )	(for tuples)
<i>l</i>	(for record)
<i>e</i>	(for map)

Accessor are use to access fields of data structure. If S is a structure, S.a if field a of S.

$$\begin{aligned}
\text{matching } (m) = & \\
& | \{ \text{true} \Rightarrow e; \text{false} \Rightarrow e; \} & (\text{match bool}) \\
& | \{ \text{nil} \Rightarrow e; \text{cons}(hd :: tl) \Rightarrow e; \} & (\text{match list}) \\
& | \{ \text{none} \Rightarrow e; \text{some}(x) \Rightarrow e; \} & (\text{match option}) \\
& | (x_i) \Rightarrow e & (\text{match tuple}) \\
& | (\{ \text{const}_i(x_i) \Rightarrow e_i; \}) & (\text{match variant})
\end{aligned}$$

Matchings represent the different branch of the control flow that are taken depending on what the value it is matched to i.e in "match a :bool with {true =<sub>i</sub> print 'toto'; false =<sub>i</sub> print 'tata'}" the program will display 'toto' if a is true and 'tata' if a is false. notice that if clause are just a boolean matching. We will call matching values (mv), matchings were all expressions are values.

## Evaluation of expression

The following describes how expression are evaluated to yield expressions  $A \rightarrow A'$  reads as A evaluates to A',  $\frac{P}{Q}$  reads as P implies Q

### base

$$\begin{aligned}
x &\rightarrow v \text{ (corresponding value in the environment)} & (\text{E-VARIABLE}) \\
\text{built in } (e_i) &\rightarrow \text{built in result } (* \text{ evaluated depending on each case } *) & (\text{E-BUILTIN}) \\
\text{SKIP} &\rightarrow \text{unit} & (\text{E-SKIP}) \\
\frac{e \rightarrow e'}{C \ e \rightarrow C \ e'} & & (\text{E-CONST})
\end{aligned}$$

Constructor are avaluated by evaluating the inner expression

$$(\lambda x.e) \ v \rightarrow [x \rightarrow v] \ e \quad (\text{E-LAMBDA})$$

Lambda expression are evaluated by replacing the bound variable in the inner expression with the value it is applied to

$$\begin{aligned}
\frac{e1 \rightarrow e1'}{e1 \ e2 \rightarrow e1' \ e2} & & (\text{E-APP1}) \\
\frac{e2 \rightarrow e2'}{v1 \ e2 \rightarrow v1 \ e2'} & & (\text{E-APP2})
\end{aligned}$$

In application, expressions are evaluated from left to right

$$\begin{aligned}
\frac{e1 \rightarrow e1'}{\text{let } x = e1 \text{ in } e2 \rightarrow \text{let } x = e1' \text{ in } e2} & & (\text{E-LET}) \\
\text{let } x = v1 \text{ in } e2 \rightarrow [x \rightarrow v1] \ e2 & & (\text{E-LETIN})
\end{aligned}$$

In let in the first expression is evaluated to a value before using this value in place of the bound variable in the second expression

$$\begin{aligned}
\frac{e1 \rightarrow e1'}{e1; e2 \rightarrow e1'; e2} & & (\text{E-SEQ}) \\
\text{unit; } e2 \rightarrow e2 & & (\text{E-SEQNEXT})
\end{aligned}$$

In sequence the left expression are evaluated first and shall yield a unit

$$\begin{aligned}
\frac{e1 \rightarrow e1'}{\text{while } e1 \text{ then } e2 \rightarrow \text{while } e1' \text{ then } e2} & & (\text{E-LOOP}) \\
\text{while true(= } e1) \text{ then } e2 \rightarrow e2; \text{ while } e1 \text{ then } e2 & & (\text{E-LOOPTRUE}) \\
\text{while false then } e2 \rightarrow \text{unit} & & (\text{E-LOOPFALSE})
\end{aligned}$$

In loop, the condition expression is evaluated first, if it is true, the loop yield a sequence of inner expression follow by a replication of the loop, otherwise the loop yield a unit

$$\frac{e \rightarrow e'}{x(.a_i) = e \rightarrow x(.a_i) = e'} \quad (\text{E-ASSIGN})$$

$$x(.a_i) = v \rightarrow x'(.a_i) \text{ with } x' \text{ as } x \text{ with field } (.a_i) \text{ replace by } v \quad (\text{E-ASSIGN2})$$

In an assign expression, the expression to be assign is evaluated first and then the expression yield the data structure with the corresponding field replace by the value yield by right expression

$$\frac{e \rightarrow e'}{e \text{ as } T \rightarrow e' \text{ as } T} \quad (\text{E-ASCR1})$$

$$v \text{ as } T \rightarrow v \quad (\text{E-ASCR2})$$

Ascription are dropped while being evaluated

### data structure

$$\frac{e_j \rightarrow e'_j}{(v_i, e_j, e_k) \rightarrow (v_i, e'_j, e_k)} \quad (\text{E-TUPLES})$$

$$\frac{e_j \rightarrow e'_j}{\{l_i = v_i, l_j = e_j, l_k = e_k\} \rightarrow \{l_i = v_i, l_j = e'_j, l_k = e_k\}} \quad (\text{E-RECORDS})$$

$$\frac{e1_j \rightarrow e1'_j}{[v1_i = v_i, e1_j = e2_j, e1_k = e2_k] \rightarrow [v1_i = v_i, e1'_j = e2_j, e1_k = e2_k]} \quad (\text{E-MAP})$$

$$\frac{e2_j \rightarrow e2'_j}{[v1_i = v_i, v1_j = e2_j, e1_k = e2_k] \rightarrow [v1_i = v_i, v1_j = e2'_j, e1_k = e2_k]} \quad (\text{E-MAP})$$

$$\frac{e1_j \rightarrow e1'_j}{[[v1_i = v_i, e1_j = e2_j, e1_k = e2_k]] \rightarrow [[v1_i = v_i, e1'_j = e2_j, e1_k = e2_k]]} \quad (\text{E-BIGMAP})$$

$$\frac{e2_j \rightarrow e2'_j}{[[v1_i = v_i, v1_j = e2_j, e1_k = e2_k]] \rightarrow [[v1_i = v_i, v1_j = e2'_j, e1_k = e2_k]]} \quad (\text{E-BIGMAP})$$

$$\frac{e_j \rightarrow e'_j}{[v_i, e_j, e_k] \rightarrow [v_i, e'_j, e_k]} \quad (\text{E-LIST})$$

$$\frac{e_j \rightarrow e'_j}{\{v_i, e_j, e_k\} \rightarrow \{v_i, e'_j, e_k\}} \quad (\text{E-SET})$$

$$\frac{e \rightarrow e'}{e(.a_i) \rightarrow e'(.a_i)} \quad (\text{E-ACCESS})$$

These rules Means in a data structure, expression are fully evaluated from left to right until it reach a value.

### look up

$$(v_i)[j] \rightarrow v_j \quad (\text{E-LUPTUPLE})$$

$$\{l_i = v_i\}[l_j] \rightarrow v_j \quad (\text{E-LUPRECORD})$$

$$[v1_i = v2_i][v1_j] \rightarrow v2_j \quad (\text{E-LUPMAP})$$

$$[[v2_i = v2_i]][v1_j] \rightarrow v2_j \quad (\text{E-LUPBIGMAP})$$

$$[v_i][j] \rightarrow v_j \quad (\text{E-LUPLIST})$$

$$\{v_i\}[j] \rightarrow v_j \quad (\text{E-LUPSET})$$

These rules indicated that when a loop up is evaluate to the element in the data-structure whose key/label/indice correspond to the argument of the lookup (no evaluation if there is no correspondence)

## matching

$$\frac{e \rightarrow e'}{\text{match } e \text{ with } m \rightarrow \text{match } e' \text{ with } m} \quad (\text{E-MATCH1})$$

$$\text{match } v \text{ with } m \rightarrow e \text{ with } v \Rightarrow e \text{ in } m \quad (\text{E-MATCH2})$$

To sum up, in matching expression, the element to be match is first evaluated, then the expression is evaluated by look in the matching value, the element to be match against and extracting the expression given.

## Derive form

The following describe equivalent notation. Meaning one could be drop for the AST without change the CSTs

$$e1; e2 \iff (\lambda x : \text{Unit}.e1) e2 \text{ with } x \text{ not a free variable in } e1$$

$$\text{let } x = e1 \text{ in } e2 \iff (\lambda x : T1.e2) e1$$