Tezos Contract Script Language Specification

The language is stack based, with high level data types and primitives and scrict static type checking. Its design is insipired by Forth, Scheme, ML and Cat.

This specification gives the complete instruction set, type system and semantics of the language. It is meant as a precise reference manual, not an easy introduction. Even though, some examples are provided at the end of the document and can be read first or at the same time as the specification.

Table of contents

I - Type system

The types T of values in the stack are written using notations

* bool, string, void, u?int{8|16|32|64}, float, the core primitive types,
* identifier for a primitive data-type,
* T identifier for a parametric data-type with one parameter type T,
* identifier T_0 ... T_n for a parametric data-type with several parameters,
* 'a for a type variable,
* _ for an anonymous type variable,
* [P] for a code quotation whose program type is P,
* lambda T_arg T_ret is a shortcut for [T_arg :: [] -> T_ret :: []].
* other specific notations for compound types, described later.

Instructions, programs and primitives of the language are also typed, their types P are written using notation

```
    S before   ->   S after
```

where a stack type S can be written

* [] for the empty stack,
* T_top : S_rest for the stack whose first value has type Ttop and queue Srest,
* 'A for a stack type variable,
* _ for an anonymous stack type variable.

II - Semantics

The instructions are specified as follows, giving their mnemonic, type in the previously defined syntax, and small step semantics as a list of rewriting rules of the form

```
   > pre state => result state
```

where the preconditions of all rules are to be read in order, the first match selecting the behaviour of the instruction, so that the choice is deterministic. Only the valid pre states are described, any other cannot happen thanks to static typing.

The pre and post result states are described as

  * pairs code / stack for stack manipulation primitives,
  * triples code / stack / memory for primitives that also manipulate memory,
  * [FAIL] for a fatal failure state.

The notations used are

  * ; to represent the concatenation of instructions or sequences,
  * [] for the empty code sequence,
  * top : tail for stack consing, as in types,
  * constants for fixed stack elements,
  * identifiers for variable stack or code elements,
  * _ for elements whose value does not affect the semantics.

The memory is described as a relation between locations and constants of the form

```
  l = const, ..., l = const
```

where constants can be

  * integers with their type e.g. (Uint8 3),
  * floats in libc-style notation e.g. (Float 4.5e2) ,
  * Void the unique value of type void
  * booleans True and False,
  * string literals String "contents",
  * memory locations <l>,
  * structured constants of compound types described later.

III - Core instructions

Sequence evaluation

```
   * (I :: [ 'A -> 'B ]) ; (C :: [ 'B -> 'C ])
     :: 'A   ->   'C
     > I ; C / SA   =>   C / SB   iff   I / SA   =>   [] / SB
```

Stack operations

```
   * DROP
     :: _ : 'A   ->   'A
     > DROP ; C / _ : S   =>   C / S
     Drop the top element of the stack.

   * DUP
     :: 'a : 'A   ->   'a : 'a : 'A
     > DUP ; C / x : S   =>   C / x : x : S
     Duplicate the top of the stack.

   * SWAP
```

```
  :: 'a : 'b : 'A   ->   'b : 'a : 'A
  > SWAP ; C / x : y : S   =>   C / y : x : S
  Exchange the top two elements of the stack.

* PUSH x
  :: 'A   ->   'a : 'A
     iff   x :: 'a
  > PUSH x ; C / S   =>   C / x : S
  Push a value onto the stack.

* DROP
  :: _ : 'A   ->   'A
  > DROP ; C / _ : S   =>   C / S
  Drop the top element of the stack.

* VOID
  :: 'A   ->   void : 'A
  > VOID ; C / S   =>   C / () : S
  Push a void value onto the stack.

Control operations
----------------

* IF bt bf
  :: bool : 'A   ->   'B
     iff   bt :: [ 'A -> 'B ]
           bf :: [ 'A -> 'B ]
  > IF ; C / True : S    =>    bt ; C / S
  > IF ; C / False : S   =>    bf ; C / S
  Conditional branching.

* LOOP body
  :: bool : 'A   ->   'A
     iff   body :: [ 'A -> bool : 'A ]
  > LOOP body ; C / True : S    =>    body ; LOOP body ; C / S
  > LOOP body ; C / False : S   =>    C / S
  A generic loop.

* DIP code
  :: 'b : 'A   ->   'b : 'C
     iff   code :: [ 'A -> 'C ]
  > DIP code ; C / x : S   =>   code ; PUSH x ; C / S
  Runs code protecting the top of the stack.

* DII+P code
  > DII(\rest)P code ; C / S   =>   DIP (DI(\rest)P code) ; C / S
  A sugar syntax for working deeper in the stack.

* LAMBDA 'a 'b code
  :: 'C   ->   lambda 'a 'b : 'C
     iff   code :: lambda 'a 'b
  > LAMBDA 'a 'b code ; C / S   =>   C / code : S
  Push a function onto the stack.

* EXEC
  :: 'a : lambda 'a 'b : 'C   ->   'b : 'C
  > EXEC ; C / a : f : S   =>   f ; C / a : S
  Execute a function from the stack.

Generic comparison
------------------

 Comparison only works on a class of types that we call comparable.
 A COMPARE operation is defined in an ad hoc way for each comparable
```

type, but the result of compare is always an int64, which can in turn be checked in a generic manner using the following combinators. The result of COMPARE is 0 if the compared values are equal, negative if the first is less than the second, and positive otherwise.

* EQ
  :: int64 : 'S -> bool : 'S
  > EQ ; C / Int64 (0) : S => C / True : S
  > EQ ; C / _ : S => C / False : S
  Checks that the top if the stack EQuals zero.

* NEQ
  :: int64 : 'S -> bool : 'S
  > NEQ ; C / Int64 (0) : S => C / False : S
  > NEQ ; C / _ : S => C / True : S
  Checks that the top if the stack does Not EQual zero.

* LT
  :: int64 : 'S -> bool : 'S
  > LT ; C / Int64 (v) : S => C / True : S iff v < 0
  > LT ; C / _ : S => C / False : S
  Checks that the top if the stack is Less Than zero.

* GT
  :: int64 : 'S -> bool : 'S
  > GT ; C / Int64 (v) : S => C / True : S iff v > 0
  > GT ; C / _ : S => C / False : S
  Checks that the top if the stack is Greater Than zero.

* LE
  :: int64 : 'S -> bool : 'S
  > LE ; C / Int64 (v) : S => C / True : S iff v <= 0
  > LE ; C / _ : S => C / False : S
  Checks that the top if the stack is Less Than of Equal to zero.

* GE
  :: int64 : 'S -> bool : 'S
  > GE ; C / Int64 (v) : S => C / True : S iff v >= 0
  > GE ; C / _ : S => C / False : S
  Checks that the top if the stack is Greater Than of Equal to zero.

Syntactic sugar is added to the frontend language for merging COMPARE and comparison combinators, and also for branching.

* CMP{EQ|NEQ|LT|GT|LE|GE}
  > CMP(\op) ; C / S => COMPARE ; (\op) ; C / S

* IF{EQ|NEQ|LT|GT|LE|GE} bt bf
  > IFCMP(\op) ; C / S => (\op) ; IF bt bf ; C / S

* IFCMP{EQ|NEQ|LT|GT|LE|GE} bt bf
  > IFCMP(\op) ; C / S => COMPARE ; IF(\op) bt bf ; C / S

IV - Data types

* bool, string, void, u?int{8|16|32|64}, float
  The core primitive types.

* list 'a
  A single, immutable, homogeneous linked list, whose elements are of type 'a, and that we note Nil for the empty list or

```
    (Cons head tail).
```

* pair 'a 'b
    A pair of values a and b of types 'a and 'b, that we write (Pair a b).

* option 'a
    Optional value that we note (None) or (Some v).

* or 'a 'b
    A union of two types, a value holding either a value a of type 'a
    or a value b of type 'b, that we write (Left a) or (Right b).

* ref 'a
    Classical imperative stores, that we note (Ref const).

* set 'a, map 'a 'b
    Imperative map and sets, optimized in the db.

V - Operations

Operations on booleans

* OR
```
  :: bool : bool : 'S   ->   bool : 'S
  > OR ; C / x : y : S   =>   C / (x | y) : S
```

* AND
```
  :: bool : bool : 'S   ->   bool : 'S
  > AND ; C / x : y : S   =>   C / (x & y) : S
```

* XOR
```
  :: bool : bool : 'S   ->   bool : 'S
  > XOR ; C / x : y : S   =>   C / (x ^ y) : S
```

* NOT
```
  :: bool : 'S   ->   bool : 'S
  > NOT ; C / x : S   =>   C / ~x : S
```

Operations on integers

Integers can be of size 1, 2, 4 or 8 bytes, signed or unsigned.
Integer Operations are homogeneous, so that performing computations
between values of different int types must be done via explicit
casts.

For specifying arithmetics, we consider that integers are all
stored on 64 bits (the largest integer size) so that we can express
the operations, in particular casts, using usual bitwise masks. In
this context, the type indicator functions are defined as follows
(which can be read both as a constraint on the bitpatttern and as a
conversion operation).

```
  Uint64 (x) = Int64 (x) = x
  Uint32 (x) = x & 0x00000000FFFFFFFF
  Int32 (x) = x & 0x00000000FFFFFFFF
            | (x & 0x80000000 ? 0xFFFFFFFF00000000 : 0)
  Uint16 (x) = x & 0x000000000000FFFF
  Int16 (x) = x & 0x000000000000FFFF
            | (x & 0x8000 ? 0xFFFFFFFFFFFF0000 : 0)
  Uint8 (x) = x & 0x00000000000000FF
```

```
  Int8 (x) = x & 0x00000000000000FF
           | (x & 0x80 ? 0xFFFFFFFFFFFFFF00 : 0)
```

We also use the function bits (t) that retrieve the meaningful number of bits for a given integer type (e.g. bits (int8) = 8).

* NEG
```
  :: t : 'S   ->   t : 'S   where   t in int{8|16|32|64}
  > NEG ; C / t (x) : S   =>   C / t (-x) : S
```
  With cycling semantics for overflows (min (t) = -min (t)).

* ABS
```
  :: t : 'S   ->   t : 'S   where   t in int{8|16|32|64}
  > ABS ; C / t (x) : S   =>   C / t (abs (x)) : S
```
  With cycling semantics for overflows (abs (min (t)) = min (t)).

* ADD
```
  :: t : t : 'S   ->   t : 'S   where   t in u?int{8|16|32|64}
  > ADD ; C / t (x) : t (y) : S   =>   C / t (x + y) : S
```
  With cycling semantics for overflows.

* SUB
```
  :: t : t : 'S   ->   t : 'S   where   t in u?int{8|16|32|64}
  > SUB ; C / t (x) : t (y) : S   =>   C / t (x + y) : S
```
  With cycling semantics for overflows.

* MUL
```
  :: t : t : 'S   ->   t : 'S   where   t in u?int{8|16|32|64}
  > MUL ; C / t (x) : t (y) : S   =>   C / t (x + y) : S
```
  Unckeched for overflows.

* DIV
```
  :: t : t : 'S   ->   t : 'S   where   t in u?int{8|16|32|64}
  > DIV ; C / t (x) : t (0) : S   =>   C / [FAIL]
  > DIV ; C / t (x) : t (y) : S   =>   C / t (x / y) : S
```

* MOD
```
  :: t : t : 'S   ->   t : 'S   where   t in u?int{8|16|32|64}
  > MOD ; C / t (x) : t (0) : S   =>   C / [FAIL]
  > MOD ; C / t (x) : t (y) : S   =>   C / t (x % y) : S
```

* CAST t_to   where   t_to in u?int{8|16|32|64}
```
  :: t_from : 'S   ->   t_to : 'S   where   t_from in u?int{8|16|32|64}
  > CAST t_to ; C / t_from (x) : S   =>   C / t_to (x) : S
```

Alternative operators are defined that check for overflows.

* CHECKED_NEG
```
  :: t : 'S   ->   t : 'S   where   t in int{8|16|32|64}
  > CHECKED_NEG ; C / t (x) : S   =>   [FAIL] on overflow
  > CHECKED_NEG ; C / t (x) : S   =>   C / t (-x) : S
```

* CHECKED_ABS
```
  :: t : 'S   ->   t : 'S   where   t in int{8|16|32|64}
  > CHECKED_ABS ; C / t (x) : S   =>   [FAIL] on overflow
  > CHECKED_ABS ; C / t (x) : S   =>   C / t (abs (x)) : S
```

* CHECKED_ADD
```
  :: t : t : 'S   ->   t : 'S   where   t in u?int{8|16|32|64}
  > CHECKED_ADD ; C / t (x) : t (y) : S   =>   [FAIL] on overflow
  > CHECKED_ADD ; C / t (x) : t (y) : S   =>   C / t (x + y) : S
```

* CHECKED_SUB
```
  :: t : t : 'S   ->   t : 'S   where   t in u?int{8|16|32|64}
```

```
  > CHECKED_SUB ; C / t (x) : t (y) : S   =>   [FAIL] on overflow
  > CHECKED_SUB ; C / t (x) : t (y) : S   =>   C / t (x - y) : S

* CHECKED_MUL
  :: t : t : 'S   ->   t : 'S   where   t in u?int{8|16|32|64}
  > CHECKED_MUL ; C / t (x) : t (y) : S   =>   [FAIL] on overflow
  > CHECKED_MUL ; C / t (x) : t (y) : S   =>   C / t (x * y) : S

* CHECKED_CAST t_to   where   t_to in u?int{8|16|32|64}
  :: t_from : 'S   ->   t_to : 'S   where   t_from in u?int{8|16|32|64}
  > CHECKED_CAST t_to ; C / t_from (x) : S   =>   C / t_to (x) : S
    iff   t_from (x) = t_to (x)
  > CHECKED_CAST t_to ; C / t_from (x) : S   =>   [FAIL]

Bitwise logical operators are also available on unsigned integers.

* OR
  :: t : t : 'S   ->   t : 'S   where   t in uint{8|16|32|64}
  > OR ; C / t (x) : t (y) : S   =>   C / t (x | y) : S

* AND
  :: t : t : 'S   ->   t : 'S   where   t in uint{8|16|32|64}
  > AND ; C / t (x) : t (y) : S   =>   C / t (x & y) : S

* XOR
  :: t : t : 'S   ->   t : 'S   where   t in uint{8|16|32|64}
  > XOR ; C / t (x) : t (y) : S   =>   C / t (x ^ y) : S

* NOT
  :: t : 'S   ->   t : 'S   where   t in uint{8|16|32|64}
  > NOT ; C / t (x) : S   =>   C / t (~x) : S

* LSL
  :: t : uint8 (s) : 'S   ->   t : 'S   where   t in uint{8|16|32|64}
  > LSL ; C / t (x) : uint8 (s) : S   =>   C / t (x << s) : S
    iff   s <= bits (t)
  > LSL ; C / t (x) : uint8 (s) : S   =>   [FAIL]

* LSR
  :: t : uint8 (s) : 'S   ->   t : 'S   where   t in uint{8|16|32|64}
  > LSR ; C / t (x) : uint8 (s) : S   =>   C / t (x >>> s) : S
    iff   s <= bits (t)
  > LSR ; C / t (x) : uint8 (s) : S   =>   [FAIL]

Integer comparison (signed or unsigned according to the type)

* COMPARE :: t : t : 'S   ->   int64 : 'S   where   t in uint{8|16|32|64}

Operations on Floats
--------------------

The float type uses double precision IEEE754 semantics, including
NaN and infinite values.

* ADD
  :: float : float : 'S   ->   float : 'S
  > ADD ; C / x : y : S   =>   C / (x + y) : S

* SUB
  :: float : float : 'S   ->   float : 'S
  > SUB ; C / x : y : S   =>   C / (x - y) : S

* MUL
  :: float : float : 'S   ->   float : 'S
```

```
  > MUL ; C / x : y : S   =>   C / (x * y) : S

* DIV
  :: float : float : 'S   ->   float : 'S
  > DIV ; C / x : y : S   =>   C / (x / y) : S

* MOD
  :: float : float : 'S   ->   float : 'S
  > MOD ; C / x : y : S   =>   C / (fmod (x, y)) : S

* ABS
  :: float : 'S   ->   float : 'S
  > ABS ; C / x : S   =>   C / (abs (x)) : S

* NEG
  :: float : 'S   ->   float : 'S
  > NEG ; C / x : S   =>   C / (-x) : S

* FLOOR
  :: float : 'S   ->   float : 'S
  > FLOOR ; C / x : S   =>   C / (floor (x)) : S

* CEIL
  :: float : 'S   ->   float : 'S
  > CEIL ; C / x : S   =>   C / (ceil (x)) : S

* INF
  :: 'S   ->   float : 'S
  > INF ; C / S   =>   C / +Inf : S

* NAN
  :: 'S   ->   float : 'S
  > NAN ; C / S   =>   C / NaN : S

* ISNAN
  :: float : 'S   ->   bool : 'S
  > ISNAN ; C / NaN : S   =>   C / true : S
  > ISNAN ; C / _ : S     =>   C / false : S

* NANAN
  :: float : 'S   ->   'S
  > NANAN ; C / NaN : S   =>   [FAIL]
  > NANAN ; C / _ : S     =>   C / S
```

Basic trigonometry with usual (libc) semantics.

```
* SIN :: float : 'S   ->  float :: 'S
* COS :: float : 'S   ->  float :: 'S
* TAN :: float : 'S   ->  float :: 'S
* ASIN :: float : 'S   ->  float :: 'S
* ACOS :: float : 'S   ->  float :: 'S
* ATAN :: float : 'S   ->  float :: 'S
* ATAN2 :: float : float : 'S   ->  float :: 'S
* SINH :: float : 'S   ->  float :: 'S
* COSH :: float : 'S   ->  float :: 'S
* TANH :: float : 'S   ->  float :: 'S
* ASINH :: float : 'S   ->  float :: 'S
* ACOSH :: float : 'S   ->  float :: 'S
* ATANH :: float : 'S   ->  float :: 'S
* POW :: float : float : 'S   ->  float :: 'S
* EXP :: float : 'S   ->  float :: 'S
* EXPM1 :: float : 'S   ->  float :: 'S
* LOG :: float : 'S   ->  float :: 'S
* LOG1P :: float : 'S   ->  float :: 'S
```

```
 * LOG10 :: float : 'S   ->  float :: 'S

 Conversion to and from integers.

 * CAST float
   :: t_from : 'S   ->   float : 'S   where   t_from in u?int{8|16|32|64}
   > CAST float ; C / x : S   =>   C / float (x) : S

 * CAST t_to   where   t_to in u?int{8|16|32|64}
   :: float : 'S   ->   t_to : 'S
   > CAST t_to ; C / NaN : S      =>   C / t_to (0) : S
   > CAST t_to ; C / +/-Inf : S   =>   C / t_to (0) : S
   > CAST t_to ; C / x : S        =>   C / t_to (floor (x)) : S

 * CHECKED_CAST float
   :: t_from : 'S   ->   float : 'S   where   t_from in u?int{8|16|32|64}
   > CHECKED_CAST float ; C / x : S   =>   [FAIL]   on overflow
   > CHECKED_CAST float ; C / x : S   =>   C / float (x) : S

 * CHECKED_CAST t_to   where   t_to in u?int{8|16|32|64}
   :: float : 'S   ->   t_to : 'S
   > CHECKED_CAST t_to ; C / x : S   =>   [FAIL]   on overflow or NaN
   > CHECKED_CAST t_to ; C / x : S   =>   C / t_to (floor (x)) : S

 IEEE754 comparison

 * COMPARE :: float : float : 'S   ->   int64 : 'S

Operations on strings
---------------------

 Strings are mostly used for naming things without having to rely on
 external ID databases. So what can be done is basically use string
 constants as is, concatenate them and use them as keys.

 * CONCAT :: string : string : 'S   -> string : 'S

 Lexicographic comparison.

 * COMPARE :: string : string : 'S   ->   int64 : 'S

Operations on timestamps
------------------------

Timestamp immediates can be obtained by the NOW operation, or
retrieved from script parameters or globals. The only valid
operations are the addition of a (positive) number of seconds and
the comparison.

 * ADD
   :: timestamp : float : 'S -> timestamp : 'S
   > ADD ; C / t : period : S   =>   [FAIL]   iff   period < 0
   > ADD ; C / t : period : S   =>   C / (t + period seconds) : S

 * ADD
   :: timestamp : uint{8|16|32|64} : 'S -> timestamp : 'S
   > ADD ; C / t : seconds : S   =>   [FAIL]   on overflow
   > ADD ; C / t : seconds : S   =>   C / (t + seconds) : S

 Comparison.

 * COMPARE :: timestamp : timestamp : 'S   ->   int64 : 'S

Operations on pairs
```

```
  ------------------

   * PAIR
     :: 'a : 'b : 'S   ->   pair 'a 'b : 'S
     > PAIR ; C / a : b : S   =>   C / (Pair a b) : S
     Build a pair from the stacks top two elements.

   * P(A*AI)+R
     > PA{N}AI(\rest)R ; C / S   =>   DIP (PA{n-1}AIR) ; P(\rest)R ; C / S
     > PAIR ; C / S             =>   PAIR ; C / S
     > PR ; C / S               =>   C / S
     A syntactic sugar for building nested pairs.

   * CAR
     :: pair 'a _ : 'S   ->   'a : 'S
     > Car ; C / (Pair a _) : S   =>   C / a : S
     Access the left part of a pair.

   * CDR
     :: pair _ 'b : 'S   ->   'b : 'S
     > Car ; C / (Pair _ b) : S   =>   C / b : S
     Access the left part of a pair.

   * C[AD]+R
     > CA(\rest)R ; C / S   =>   CAR ; C(\rest)R ; C / S
     > CD(\rest)R ; C / S   =>   CDR ; C(\rest)R ; C / S
     > CR ; C / S          =>   C / S
     A sugar syntax for accessing fields in nested pairs.

  Operations on refs
  ------------------

   * REF
     :: 'a : 'S   ->   ref 'a : 'S
     > REF ; C / a : S / M   =>   C / l : S / l = (Ref a), M
     Build a ref from its initial contents.

   * DEREF
     :: ref 'a : 'S   ->   'a : 'S
     > DEREF ; C / l : S / l = (Ref a), M   =>   C / a : S / l = (Ref a), M
     Access the contents of a ref.

   * SET
     :: 'a : ref 'a : 'S   ->   'S
     > SET ; C / v :: l : S / l = (Ref _), M   =>   C / S / l = (Ref v), M
     Update the contents of a ref.

   * INCR step
     :: ref 'a : 'S   ->   'S
        iff   step :: 'a, operator ADD defined on 'a
     > INCR step ; C / l : S / M   =>   DUP ; DEREF ; PUSH step ; ADD ; Set ;
C / S / M
     Increments a counter.

   * DECR step
     :: ref 'a : 'S   ->   'S
        iff   step :: 'a, operator SUB defined on 'a
     > DECR step ; C / l : S / M   =>   DUP ; DEREF ; PUSH step ; SUB ; Set ;
C / S / M
     Decrements a counter.

  Operations on sets
  ------------------
```

* EMPTY_SET 'elt
  :: 'S -> set 'elt : 'S
  Build a new, empty imperative set, whose types must match its
  usage in the rest of the program. The 'elt type must be
  comparable (the COMPARE primitive must be defined over it).

* MEM
  :: 'key : set 'elt : 'S -> bool : 'S
  Check for the presence of an element in a set.

* UPDATE
  :: 'elt : bool : set 'elt : 'S -> 'S
  Inserts or removes an element in a set, replacing a previous value.

* ITER
  :: lambda 'elt void : set 'elt : 'S -> 'S
  Apply a function on a set.

* REDUCE
  :: lambda (pair 'elt * 'b) 'b : set 'elt : 'b : 'S -> 'b : 'S
  Apply a function on a set passing the result of each
  application to the next one and return the last.

Operations on maps
----

* EMPTY_MAP 'key 'val
  :: 'S -> map 'key 'val : 'S
  Build a new, empty imperative map, whose types must match its
  usage in the rest of the program. The 'key type must be
  comparable (the COMPARE primitive must be defined over it).

* GET
  :: 'key : map 'key 'val : 'S -> option 'val : 'S
  Access an element in a map, returns an optional value to be
  checked with [Opt_cases].

* MEM
  :: 'key : map 'key 'val : 'S -> bool : 'S
  Check for the presence of an element in a map.

* UPDATE
  :: 'key : option 'val : map 'key 'val : 'S -> 'S
  Assign or remove an element in a map.

* ITER
  :: lambda (pair 'key 'val) void : map 'key 'val : 'S -> 'S
  Apply a function on a map.

* MAP
  :: lambda (pair 'key 'val) 'b : map 'key 'val : 'S -> map 'key 'b :
'S
  Applys a function on a map and return the map of results under
  the same bindings.

* REDUCE
  :: lambda (pair (pair 'key 'val) 'b) 'b : map 'key 'val : 'b : 'S ->
'b : 'S
  Apply a function on a map passing the result of each
  application to the next one and return the last.

Operations on optional values

```
 * SOME
   :: 'a : 'S   ->   'a? : 'S
   > SOME ; C / v :: S   =>   C / (Some v) :: S
   Pack a present optional value.

 * NONE 'a
   :: 'S   ->   'a? : 'S
   > NONE ; C / v :: S   =>   C / None :: S
   The absent optional value.

 * IF_SOME bt bf
   :: 'a? : 'S   ->   'b : 'S
      iff   bt :: [ 'a : 'S -> 'b : 'S]
            bf :: [ 'S -> 'b : 'S]
   > IF_SOME ; C / (Some a) : S    =>    bt ; C / a : S
   > IF_SOME ; C / (None) : S   =>    bf ; C / S
   Inspect an optional value.

Operations on unions
--------------------

 * LEFT 'b
   :: 'a : 'S   ->   or 'a 'b : 'S
   > LEFT ; C / v :: S   =>   C / (Left v) :: S
   Pack a value in a union (left case).

 * RIGHT 'a
   :: 'b : 'S   ->   or 'a 'b : 'S
   > RIGHT ; C / v :: S   =>   C / (Right v) :: S
   Pack a value in a union (right case).

 * IF_LEFT bt bf
   :: or 'a 'b : 'S   ->   'c : 'S
      iff   bt :: [ 'a : 'S -> 'c : 'S]
            bf :: [ 'b : 'S -> 'c : 'S]
   > IF_LEFT ; C / (Left a) : S    =>    bt ; C / a : S
   > IF_LEFT ; C / (Right b) : S   =>    bf ; C / b : S
   Inspect an optional value.

Operations on lists
-------------------

 * CONS
   :: 'a : list 'a : 'S   ->   list 'a : 'S
   > CONS ; C / a : l : S   =>   C / (Cons a l) : S
   Prepend an element to a list.

 * NIL 'a
   :: 'S   ->   list 'a : 'S
   > NIL ; C / S   =>   C / Nil : S
   The empty list.

 * IF_CONS bt bf
   :: list 'a : 'S   ->   'b : 'S
      iff   bt :: [ 'a : list 'a : 'S -> 'b : 'S]
            bf :: [ 'S -> 'b : 'S]
   > IF_CONS ; C / (Cons a rest) : S   =>    bt ; C / a : rest : S
   > IF_CONS ; C / Nil : S   =>    bf ; C / S
   Inspect an optional value.

 * ITER
   :: lambda 'a void : list 'a : 'S -> 'S
   Apply a function on a list from left to right.
```

* MAP
  :: lambda 'a 'b : list 'a : 'S -> list 'b : 'S
  Apply a function on a list from left to right and
  return the list of results in the same order.

* REDUCE
  :: lambda (pair 'a 'b) 'b : list 'a : 'b : 'S -> 'b : 'S
  Apply a function on a list from left to right
  passing the result of each application to the next one
  and return the last.

VI - Domain specific data types

* tez
  A special numeric type for manipulating tokens.

* contract 'param 'result
  A contract, with the type of its code.

* key
  A public key.

* signature
  A cryptographic signature.

VII - Domain specific operations

Operations on Tez

Operations on tez are limited to prevent overflow and mixing them
with other numerical types by mistake. They are also mandatorily
checked for under/overflows.

* ADD
  :: tez : tez : 'S -> tez : 'S
  > Add ; C / x : y : S => [FAIL] on overflow
  > Add ; C / x : y : S => C / (x + y) : S

* SUB
  :: tez : tez : 'S -> tez : 'S
  > Sub ; C / x : y : S => [FAIL] iff x < y
  > Sub ; C / x : y : S => C / (x - y) : S

* MUL
  :: tez : u?int{8|16|32|64} : 'S -> tez : 'S
  > Mul ; C / x : y : S => [FAIL] on overflow
  > Mul ; C / x : y : S => C / (x * y) : S

* COMPARE :: tez : tez : 'S -> int64 : 'S

Operations on contracts

* MANAGER
  :: contract 'p 'r : 'S -> key : 'S
  Access the manager of a contract.

* CREATE_CONTRACT
  :: key : key? : bool : bool : tez :

lambda (pair (pair tez 'p) 'g) (pair 'r 'g) : 'g : 'S -> contract 'p 'r : 'S
Forge a new contract. As with non code-emitted originations the contract code takes as argument the transfered amount plus an ad-hoc argument and returns an ad-hoc value. The code also takes the global data and returns it to be stored and retrieved on the next transaction. These data are initialized by another parameter. The calling convention for the code is as follows: (Pair (Pair amount arg) globals)) -> (Pair ret globals), as extrapolable from the instruction type. The first parameters are the manager, optional delegate, then spendable and delegatable flags and finally the initial amount taken from the currently executed contract. The contract is returned as a first class value to be called immediately or stored.

* CREATE_ACCOUNT
:: key : key? : bool : tez : 'S -> contract void void : 'S
Forge an account (a contract without code). Take as argument the manager, optional delegate, the delegatable flag and finally the initial amount taken from the currently executed contract.

* TRANSFER_TOKENS
:: 'p : tez : contract 'p 'r : 'S -> 'r : 'S
Forge and evaluate a transaction. The parameter and return value must be consistent with the ones expected by the contract, void for an account.

* BALANCE
:: 'S -> tez :: 'S
Push the current amount of tez of the current contract.

* SOURCE 'p 'r
:: 'S -> contract 'p 'r :: 'S
Push the source contract of the current transaction.

* SELF
:: 'S -> contract 'p 'r :: 'S
where contract 'p 'r is the type of the current contract
Push the current contract.

* AMOUNT
:: 'S -> tez :: 'S
Push the amount of the current transaction.

Special operations
---

* STEPS_TO_QUOTA
:: 'S -> uint32 :: 'S
Push the remaining steps before the contract execution must terminate.

* NOW
:: 'S -> timestamp :: 'S
Push the timestamp of the block whose validation triggered this execution (does not change during the execution of the contract).

* FAIL
:: _ -> _
> FAIL ; _ / _ => [FAIL]
Explicitly abort the current transaction (and all of its parents).

Cryptographic primitives
---

* H
  :: 'a : 'S -> signature : 'S
  Compute a sha256c hash of the value contents

* CHECK_SIGNATURE
  :: key : pair signature string : 'S -> bool : 'S
  Check that a sequence of bytes has been signed with a given key using curve Ed25619

* COMPARE :: key : key : 'S -> int64 : 'S

VIII - Concrete syntax

The concrete language is based on a single syntactic construction: a primitive applied to a set of arguments. This is used to describe instructions, type annotations as well as constant declarations. The simplest form use indentation to distinguish the arguments:

```
PRIM
  arg1
  arg2
  ...
```

It allows copound arguments, for instance:

```
PRIM1
  PRIM2
    arg1_prim2
    arg2_prim2
  arg2_prim1
```

It is possible to put successive arguments on a single line using a semicolon as a separator:

```
PRIM
  arg1; arg2
  arg3; arg4
```

It is also possible to add arguments on the same line as the primitive as a lighter way to write simple expressions. An other representation of the first example is:

```
PRIM arg1 arg2 ...
```

It is possible to mix both notations as in:

```
PRIM arg1 arg2
  arg3
  arg4
```

Or even:

```
PRIM arg1 arg2
  arg3; arg4
```

Equivalent to:

```
PRIM
  arg1
  arg2
  arg3
```

```
  arg4
```

A trayling semicolon is ignored:

```
PRIM
  arg1;
  arg2
```

Calling a primitive with a compound argument on a single line is allowed by wrapping with parentheses. Another notation for the second example is:

```
PRIM1 (PRIM2 arg1_prim2 arg2_prim2) arg2_prim1
```

Sequences

Higher order keywords such as IF or LAMBDA take single instructions by default. Sequences of operations can be used instead by grouping them inside braces:

```
IF instr_true instr_false

IF { instr1_true ; instr2_true ; ... } { instr1_false ; instr2_false ; ...
}

IF
  { instr1_true ; instr2_true ; ... }

  { instr1_false ; instr2_false ; ... }
```

A sequence block can be split on several lines. In this situation, the whole block, including the closint brace, must be indented with respect to the first instruction.

```
LAMBDA t_arg t_ret
  { instr1 ; instr2
    instr3 ; instr4 }
```

Type annotations

Some contructions need type annotations, for instance NIL (the type of elements in the list since it cannot be infered from an empty one) and LAMBDA (to typecheck the body against the signature). Those are simply passed as normal arguments would.

```
NIL int8

LAMBDA int8 void { DROP ; VOID }
```

Lexical convention

Instructions are represented by uppercase identifiers, type constructor are lowercase identifiers and constant constructors are Capitalised.

* Types, in lowercase, in prefixed notation as in this specification:

```
  string

  pair string (pair int8 tez)

  lambda int8 int16
```

Of course, types can be split over multiple lines using the common indented notation.

```
map
  string
  uint32
```

* Constants are built using constructors (starting with a capital) followed by the actual value.

```
Int8 1
```

```
Float 3.5e12
```

Compound constants such as lists, in order not to repeat the same constant constructor for each element, take the type(s) of inner values as first argument(s), and then the values without their constructors.

```
List int8 1 2 3 4 5
```

```
Pair int8 int16 1 2
```

For constructors whose type cannot be completely deduced fron a single value, the free type variables must be specified. For this, some constant constructors take extra types arguments as follows.

```
List int8
```

```
None tez
```

```
Left (Int8 3) int16
```

```
Right int16 (Int8 3)
```

When the type is already completely specified, by a parent constructor or as in the instruction PUSH, these annotations must be omitted.

```
Pair int8 (list int16) 1 (List 2 3)
```

```
Pair (option (pair void int8)) void
  None
  Void
```

```
Pair (or int8 string) (or int8 string)
  Left 3
  Right "text"
```

The PUSH instruction takes a single constant, its notation should thus be such as { PUSH (List int8 1 2 3) }, but the useless parentheses are dropped so we write { PUSH List int8 1 2 3 }.

* Instructions, in uppercase:

```
ADD
```

Comments

A hash sign (#) anywhere outside of a string literal will make the rest of the line (and itself) completely ignored, as in the

following example.

```
PUSH Int8 1 # pushes 1
PUSH Int8 2 # pushes 2
ADD         # computes 2 + 1
```

IX - Reference implementation

The language is implemented in OCaml as follows:

* The lower internal representation is written as a GADT whose type parameters encode exactly the typing rules given in this specification. In other words, if a program written in this representation is accepted by OCaml's typechecker, it is mandatorily type-safe. This of course also valid for programs not handwritten but generated by OCaml code, so we are sure that any manipulated code is type-safe.

  In the end, what remains to be checked is the encoding of the typing rules as OCaml types, which boils down to half a line of code for each instruction. Everything else is left to the venerable and well trusted OCaml.

* The interpreter is basically the direct transcription of the rewriting rules presented above. It takes an instruction, a stack and transforms it. OCaml's typechecker ensures that the transformation respects the pre and post stack types declared by the GADT case for each instruction.

  The only things that remain to we reviewed are value dependent choices, such as that we did not swap true and false when interpreting the If instruction.

* The input, untyped internal representation is an OCaml ADT which has the exact same shape as the GADT, except for the stack invariants. It is the target language for the parser, since not all parsable programs are well typed, and thus could simply not be constructed using the GADT.

* The typechecker is a simple function that transform each untyped instruction to its typed counterpart. It is just a checker, not an inferer, and thus takes some annotations (basically the inpout and output of the program, of lambdas and of uninitialized imperative structures). It works by performing a symbolic evaluation of the program, transforming a symbolic stack. It only needs one pass over the whole program.

  Here again, OCaml does most of the checking, the structure of the function is very simple, what we have to check is that we transform an If into a typed If, a Dup into a typed Dup, etc.