Lamtez Cheat-sheet

1 Contract

```
<contract> ::=
  <type_declaration*>
  <storage_declaration*>
  <expression>
```

The expression must denote a function, taking a parameter and returning a result. If types can't be innferred from the code, they must be annotated.

2 Type declarations

2.1 Labelled sum types

```
type <name>=<Label_0>:<type_0>+...+<Label_n>:<type_n>
```

Union of named type cases, in arbitrary number, compiling into nested `Left/Right` cases. Two products or sum types cannot share a common label. The value associated with a label can be omitted when it is of type `unit`.
The following sym types are built-in:

```
type option t =  None + Some t
type bool     = False + True
type list a   =   Nil + Cons (a * list a)
```

2.2 Labelled product types

```
type <name>=<Label_0>:<type_0>*...*<Label_n>:<type_n>
```

Record structures with named fields, in arbitrary number, compiling into nested pairs. Two product or sum types cannot share a common label.

2.3 Type aliases

```
type <name> <params>* = <type(params)>
```

`type account = contract unit unit` is built-in.

2.4 Unlabelled tuple types

```
(type_0 * ... * type_n)
```

Unlabelled tuples can be used as annotations without type declarations, as a parameter for sum cases and product fields, or named with a type alias.
`type pair a b = (a*b)` is built-in.

3 Storage declaration

```
@<name> :: <type> (= <expr>)?
```

Storage fields are persisted as the contract's storage. If every storage field has an initialization value, the compiler can optionally produce an initial store value litteral (option `--store-output`). Names can be capitalized or not.

4 Expressions

4.1 Litterals

Natural numbers: `[0-9]+`
e.g. `1`, `123456`

Signed integers: `(+|-)[0-9]+`
e.g. `+1`, `-2`, `+0`.
Plus sign is mandatory for ints ≥ 0.
Beware that `f+1` (without space) is parsed as `f(+1)`, not `f + 1`.

Tez: `tz<d>*.<dd> | tz<d>+`
where `<d>` is a decimal digit, `<dd>` a pair thereof.
e.g. `tz1`, `tz2.00`, `tz.02`.

Key hashes: `tz1<b58_char>+`
e.g. `tz1MGxkasF5ABVVrxzxGbWo8wPg7EaLKn4RS`

Signatures: `sig:[0-9a-f]+`
e.g. `sig:91b334be19d66d30205563b426c2f9b3`...

Date:

```
dddd-dd-ddTdd:dd:ddZ |
dddd-dd-ddTdd:dd:dd(+|-)dd(:dd?)
```

where `d` are decimal digits.
e.g. `1970-01-01T00:00:00Z`,
`2014-09-02T12:34:56+02:00`

String: `"[^"]*"`:
e.g. `"foo"`, `"hello \"world\""`.
Must fit on a single line, double quotes escaped with backslashes; same escaped characters as Michelson.

4.2 Composite types and collections

```
{ <Label_0>:<expr_0>,...,<Label_n>:<expr_n>} # Product
(<expr_0>, ..., <expr_n)                     # Tuple
(list <expr_0> ... <expr_n>)                 # List
(set <expr_0> ... <expr_n>)                  # Set
(map <expr_k0> <expr_v0>...<expr_kn> <expr_vn>) # Map
```

4.3 Identifiers

Identifiers start with a lowercase letter, can contain alphanumeric characters, underscores and dashes.

4.4 Functions

```
fun (<pattern>(::<type_param>)?)+ (::<type_result>)?:
   <expr>
```

As in ML, arguments are syntactically separated by spaces without mandatory parentheses around them. Unlike ML, multi-parameter functions are encoded with tuples rather than in Curry style, i.e. `f x y z` is syntax sugar for `f (x, y, z)` not `(((f x) y) z)`. Function application binds tighter than binary and unary operators (e.g. `a b + c d` parses as `a(b) + c(d)`), but looser than product field accessors.

```
f arg_0 arg_1 ... arg_n
f (g arg_g0 arg_g1) arg_f1
```

Functions are created with "`fun`" standing for the λ operator (backslash also accepted for Haskellers); parameters are comma-separated between them, separated from the function body by a colon; when parameter types can't be inferred, they must be annotated with a double-colon `::` sign. Optionally, the function result can be annotated with a double colon after annotated parameters.

```
fun p: p - 1
fun p0, p1: p0 * p1
fun p0 :: tez, p1 :: nat :: tez: p0 * p1
fun p :: unit: ()
fun p :: unit :: time: self-now
```

4.5 Local variables

```
let <pattern> = <expr_0>; <expr_1>; ...; <expr_n>
```

Local variables are created with the `let` keyword, either as identifiers, or as binding patterns (nested product types).

4.6 Binding patterns

```
  <id>
| _
| (<pattern_0>,...,<pattern_n>)
| {<Label_0>:<pattern_0>,...,<Label_n>:<pattern_n>}
```

When binding an identifier and a value (in function parameters, let declarations and sum cases), patterns for product types and tuples can be used, thus binding several variables simultaneously. Patterns can be nested. They may not use sum types, as it would introduce the possibility of runtime failures.

```
let (a, b) = (1, 2);
case p | Some {Lon: x, Lat: y}: ... end;
let norm = fun (x, y) :: int*int: x*x + y*y
```

4.7 Flow control

4.7.1 Sum case deconstruction

```
case <expr>
| <Label_0> <pattern_0>: <expr_0>
| ...
| <Label_n> <pattern_n>: <expr_n>
end
```

Labels can appear in any order, but all cases of one sum type must be present. Patterns `p_n` which are not used in `e_n` can be omitted. Booleans, options and lists are sum cases and can be deconstructed with a sum type.

4.7.2 If then else / switch

```
if
| <expr_cond_0>: <expr_if_true_0>
| ...
| <expr_cond_n>: <expr_if_true_n>
| else: <expr_if_everything_false>
end
```

The code `<expr_if_true_i>` corresponding to the first true `<expr_cond_i>` is evaluated. `else` clause may be ommitted if the result type is `unit`. The first bar after the `if` is optional.

4.8 Field access

```
<expr>.<d>                  # Tuple field access
<expr>.<Label>              # Product type field access
<expr> <- <Label>: <expr>   # Product type field update
@<name>                     # Persistent field access
@<name> <- <expr>;          # Persistent field update
<Label> <expr>              # Sum type constructor
```

Tuple fields are accessed with a dot followed by the field number, 0-indexed.
Labelled product fiels are accessed with a dot followed by the field label.
Persistent storage fields are accessed with "`@`" followed by the field name. They can be updated with `@<name> <- <expr>;`, but such operations cannot occur in a function, a litteral product or a function argument.
Sum types are constructed with the case label followed by the associated value; the value can be omitted when its type is `unit`.

```
let t = ("a","b","c"); t.1;    # Tuple access
let p = {X:45,Y: 1}; p.X;      # Product access
let s :: option nat = Some 42; # Sum constr.
let x = not @field;            # store access
@field <- x                    # store update
```

4.9 Type annotations

```
<expr> :: <type>
```

Michelson doesn't support polymorphic types. To prevent Lamtez from guessing such types, as well as to improve contract readbility, you might have to add type annotations.

4.10 Operations

4.10.1 Binary infix operators

in decreasing order of precedence:

- `a * b` and `a / b` (euclidian division);
- `a + b` and `a - b`;
- bit shifting `a << b` and `a >> b`;
- comparisons `a < b`, `a <= b`, `a = b`, `a != b`, `a > b`, `a >= b`;
- logical and bitwise conjunction `a && b`;
- logical disjunctions `a || b` and `a ^^ b`.

Beware of spaces around addition and substraction: "`-`" is a valid identifier character, and both characters can be interpreted as prefix sign for a signed literal integer.

4.10.2 Unary operators

`-<expr>`, `not <expr>`.
Precedence higher than binary ops, lower than field accessors and function applications.

4.10.3 Operands and result types

Operator types and semantics are lifted from Michelson:

- `nat` and `int` can be added/substracted/multiplied together, and result in a signed `int`, except for `nat+nat` and `nat*nat` which result in `nat`.
- As in Michelson, divisions are euclidian, return an (integer division $*$ natural reminder) option pair (`None` in case of division by 0).
- `nat` can be added/substracted with `time`, `tez` can be added/-substracted together.
- `tez` can be multiplied by `nat`, divided together or by a `nat`.
- Logical operators work on `bool` as well as bitwise on `nat`.
- comparison operators work on every type (both operands must have the same type, though).

5 Primitives

5.1 Current contract context

```
val fail          :: fail # FAIL
val self-amount   :: tez  # AMOUNT
val self-balance  :: tez  # BALANCE
val self-now      :: time # NOW
val self-contract :: $\forall$p,r: contract p r # SELF
val self-source   :: $\forall$p,r: contract p r # SOURCE
val self-steps-to-quota :: nat # STEPS_TO_QUOTA
```

5.2 Contract management

```
val contract-call :: ∀p,r:
  contract p r → p → tez →
  r # TRANSFER_TOKENS
val contract-create :: ∀p,s,r:
  key → option key → bool → bool →
  tez → (p*s→r*s) → storage →
  contract p r # CREATE_CONTRACT
val contract-create-account ::
  key → option key → bool → tez →
  account # CREATE_ACCOUNT
val contract-get ::
  key → account # DEFAULT_ACCOUNT
val contract-manager :: ∀p, s:
  contract p s → key # MANAGER
```

`contract-call` cannot be used in a function / tuple / product / collection.

5.3 Cryptography

```
val crypto-check :: key → sig → string → bool
val crypto-hash  :: ∀a: a → string
```

5.4 Collections

```
val list-map    :: ∀a,b: list a → (a→b) → list b
val list-reduce:: ∀a,acc:
    list a → acc → (a→acc→acc) → acc
val list-size   :: ∀a: list a → nat
val set-map     :: ∀a,b: set a → (a→b) → set b
val set-mem     :: ∀elt: set elt → elt → bool
val set-reduce  :: ∀elt acc:
    set elt → acc → (elt→acc→acc) → acc
val set-update  :: ∀elt: elt → bool → set elt
val set-size    :: ∀elt: set elt → nat
val map-get     :: ∀k,v: k → map k v → option v
val map-map     :: ∀k,v0,v1:
    map k v0 → (k→v0→v1) → map k v1
val map-mem     :: ∀k,v: k → map k v → bool
val map-reduce  :: ∀k,v,acc:
    map k v → acc (k→v→acc→acc) → → acc
val map-update  :: ∀k,v:
    k → option v → map k v → map k v
val map-size    :: ∀k,v: map k v → nat
```