Archetype Cheat sheet

This document doesn't describe all of the syntax of the language, but contains everything that is covered during this training, and needed to solve the exercises.

Important: in all the examples below, when some text is between square brackets and italics, *[like this]*, all of it should be replaced by the value you need. In particular, you shouldn't type these square brackets.

Links

The links below should not be needed during the training, as everything you need should be provided in the sections below. Please use this document rather than documentation, as much as possible, during the training.

Documentation	https://archetype-lang.org/
Completium	https://completium.com/

Completium command line

To test Archetype contracts, please use `completium-cli`, and the mockup mode.

Initialization of the mockup mode	`completium-cli mockup init`
Using the mockup endpoint	`completium-cli set endpoint mockup`
Selection of a default account	`completium-cli switch account`
Running a test	`node [path to .js file]` `# Warning: make sure you are in the same folder as the .js file the path to the contract in the .js file is relative to your current folder`
Sélection of another endpoint (for example a testnet)	`completium-cli switch endpoint`
Creation of a new account based on a faucet	`completium-cli import faucet carlfaucet.json as carl`
Transfer of tez	`completium-cli transfer 10tz from alice to carl`

Construction, storage, entry points

Contract	`archetype` *`[name of the contract]`* `...`
Contract with a parameter	`archetype` *`[name of the contract]`* `(`*`[parameter name]`* `:` *`[type]`*`)`
Storage variable	`variable` *`[name of the variable]`* `:` *`[type]`* `=` *`[value]`*
Read and write access to the storage	*`[name of the variable]`* `=` *`[name of the variable]`* `+ 1`
Entry point	`entry` *`[name of entry point]`*`(`*`[name param 1]`* `:` *`[type param 1]`*`, [...])` `{` *`[code]`* `}`
Entry point with verifications	`entry` *`[name of entry point]`*`(`*`[name param 1]`* `:` *`[type param 1]`*`, [...])` `{` `called by` *`[expression]`* `require {` *`[block of conditions]`* `}` `effect {` *`[code]`* `}` `}`

Basic types literals

Integer	`int`	`42i, 0i, -34, -1_000_000`
Natural	`nat`	`12, 0, 1_000_000`
Tez Token	`tez`	`12tz, 12mtz, 12utz, 1.2tz`
Boolean	`bool`	`true, false`
String	`string`	`"Hello World!"`
Address	`address`	`tz1YtuZ4vhzzn7ssCt93Put8U9UJDdvCXci4`

Date/time	`date`	`2019-01-01` `2019-01-01T01:02:03` `2019-01-01T01:02:03Z` `2019-01-01T00:00:00+01:00` `2019-01-01T00:00:00-05:30`
Duration	`duration`	`24h // 24 hours` `1s // 1 second` `3w8d4h34m18s3 // 3 weeks 8 days 4 hours 34 minutes and 18 seconds`

Completium TS Test Library

Create project	`$ completium-cli create project tezos_training` `$ cd tezos_training` `$ npm i`
Generate TS bindings	`$ completium-cli run binder-ts`
Run tests	`$ npm run test`
Structure of a test.	`import { Nat } from "@completium/archetype-ts-types";` `const assert = require('assert');` `const { my_contract } = from './binding/my_contract.ts';` `describe('[MY_CONTRACT] Contract deployment', async () => {` `  it('Deploy my_contract', async () => {` `  await my_contract.deploy(new Nat(2), { as: alice })` `  });` `})`
Deploy a contract	**# Without parameters:** `await my_contract.deploy({})` **# With parameters :** `await my_contract.deploy(new Nat(2), { as: alice })`
Call an entry point	`await my_contract.increment({});`
Access to a variable of the storage	`const counter = await my_contract.get_counter()` `const msg = await my_contract.get_msg()`
Test a variable value	`assert(counter.equals(new Nat(7)), "counter should be 7");` `assert(msg == "Hello", "msg should be Hello");`
Call an entry point, as a given user. (that user will be identified as the `caller`).	`import { get_account } from "@completium/experiment-ts";` `const alice = get_account("alice")` … `await my_contract.increment({ as : alice });`

An account is generated/imported with completium-cli.	
Call an entry point and transfer some tez to the contract	`await my_contract.deposit({ amount: new Tez(100) });`
Fetch the address of a contract or an account.	`const alice_addr = alice.get_address()`
Get the balance of an account	`const alice_balance = await alice.get_balance()`
Testing a contract by setting a specific value for `now`	**# when using Mockup mode:** `import { set_mockup_now } from "@completium/experiment-ts";` `const now = new Date()` `set_mockup_now(now)` **# Shift now by 5 minutes** `const nowplus5min = now.setMinutes(now.getMinutes()+5)` `set_mockup_now(new Date(nowplus5min))`
Check that a call to an entry point fails	`await expect_to_fail(async () => {` `  await my_contract.replace(new Int(200))` `}, my_contract.errors.r1)`
Create an option value	`import { Option, Nat } from "@completium/archetype-ts-types";` **# None of nat:** `const o = Option.None<Nat>()` **# Some nat value:** `const o = Option.Some<Nat>(new Nat("99999999999999999"))`
Access map (archetype asset)	`const visitors : Array<[Address, visitor_value]> =` `  await my_contract.get_visitor()`

Local variables and assignments

Declaration of a local variable	*`var [var name] = [value];`*

	var [var name] : [type var] = [value];
Declaration of a local constant	*const [var name] = [value];* *const [var name] : [type var] = [value];*
Access to the value of the variable	*[var name]*
Assignment of a new value	*[var name] := [value];*

Operators

Addition	`1 + 2`
Subtraction	`2 - 1`
Multiplication	`2 * 3`
Integer division	`4 / 2`
Modulo	`4 % 3`
Integer division and modulo	`4 %/ 3`
And	`a and b`
Or	`a or b`
Not	`not a`
Exclusive or	`a xor b`
Comparison	`0 = 1` `0 <> 1` `0 < 1` `0 <= 1` `0 > 1` `0 >= 1`
Concatenation	`"Hello " + "World!"`
Shorthand operators	*[var name] += [value];* *[var name] -= [value];* *[var name] *= [value];* *[var name] /= [value];*

Verifications

Errour	`fail(`*`[value]`*`)`
Verification with message	`do_require(`*`[condition], [message]`*`)`
Condition with only one instruction	`if` *`[expression de condition]`* `then` `  `*`[single instruction]`* `else` `  `*`[single instruction]`* *`#`* *Warning: don't put a semicolon here after the first single instruction, it would mean the end of the if instruction, and the compiler wouldn't like a new instruction that starts with else*
Condition with multiple instructions	`if` *`[expression de condition]`* `then begin` `  `*`[instructions separated with ;]`* *`end`* `else begin` `  `*`[instructions separated with ;]`* *`end;`*

Options

Type	`option<nat>`
Litteral	With a value: `some(1)` No value: `none`
Extract the value	`// if an_option is typed option<nat>:` `const v = an_option ? the : 0` **Or** `const v ?= an_option : "ERROR"  // fails if an_option is none`
Pattern matching	`match myopt with` `| some(v)  -> [code]` `| none     -> [code]` `end`

Records

Record type	`record myrecord {` `  id : nat;` `  val : string;` `}`
Record litteral	With the names of the properties: `{id = 0; val = "mystr1"}` Without the names: `{0; "mystr1"}`
Access	`my_rec.field`
Assigning a field	`my_rec.field := [value] // instruction`
Modification	`{my_rec with field = 0[; ...]} // expression of type record_id`

Asset

Asset	`asset myasset {` `  id : nat;` `  str : string;` `  v : int;` `} initialized by {` `  {0, "empty", 42i},` `  {1, "ok", 33i}` `}`
Using a big map	`asset my_asset to big_map {...}`
Adding	`myasset.add({id = 0; str = "mystr"; v = 123})`
Removing	`myasset.remove(0)`
Updating	`myasset.update(0, {str = "otherstring"; v = 321})`
Adding or updating	`myasset.add_update(0, {str = "otherstring"; v = 321})`
Existence	`myasset.contains(0)`
Testing	`const myasset = await mycontract.get_myasset()` `// array of (key,value) pairs`

Constants

Balance of the contract	`tez`	`balance`
Address of the direct caller of the contract	`address`	`caller`
Date of the the block that executes the call	`date`	`now`
Address of the original caller of the chain of contracts	`address`	`source`
Amount of the transaction.	`tez`	`transferred`

Transfers, contract, operations

Transfer of tezzies	`transfer 1tz to tz1YtuZ4vhzzn7ssCt93Put8U9UJDdvCXci4`
Calling a contract	`transfer 0tz to KT1UZ512pdDcD36GjXLJEkVifeQz3TduZ9uw call set_value<nat>(42);`
With an operation	`var e : contract<unit> = tz1YtuZ4vhzzn7ssCt93Put8U9UJDdvCXci4;` `var op : operation = make_operation(0tz, e, Unit);` `transfer op;`
List of operations	`operations`

Serialization, Hashing

Serialization of a value (returns bytes)	`pack(v)`
Deserialization of a value	`unpack<string>(v)`
Hashing of a value of type bytes	`blake2b(`*`[value as bytes]`*`)`
Hashing of a value of another type	`hashedValue = blake2b(pack(`*`[vaule]`*`)`

In the tests:

Compute the hash of a value in bytes	`blake2b(`*`bytes`*`)`
Serialize a value	`pack(`*`value`*`)`