CARDANO DEVELOPMENT COURSE

Module 4: Aiken Intro (Cont'd).

4.0. Aiken Functions

Functions in Aiken are defined using the *fn* keyword. Functions can have (typed) arguments, and always have a return type. Because in Aiken, pretty much everything is an expression, functions do not have an explicit return keyword. Instead, they implicitly return whatever they evaluate to.

```
fn add(x: Int, y: Int) -> Int {
  x + y
}

fn multiply(x: Int, y: Int) -> Int {
  x * y
}
```

Functions are first class values and so can be assigned to variables, passed to other functions, or anything else you might do with any other data type with the exception of: being part of a data-type definition.

Anonymous function

Anonymous functions can be defined with a similar syntax and assigned to an identifier using a let-binding. The identifier then serves as a name to call and pass the function around.

```
fn run() {
  let add = fn(x, y) { x + y }

  add(1, 2)
}
```

Further reading: *https://aiken-lang.org/language-tour/functions*

4.1. Aiken Constants

Constants is a way to use certain fixed values in multiple places of a project.

Examples:

```
const the_ultimate_answer_of_everything: Int = 42
const pkh =
#"00000000000000000000000000000000000000000000000000000056"
pub const key_token = ("PolicyId", "NFT", 1)
```

Constants are immutable. They can hold *almost* any Aiken expression and are fully **evaluated at compile-time**.

Constants can refer to other identifiers such as functions and other constants with one restriction: constants cannot refer to other constants that are defined **after** them in the file. As such, there's no recursive or mutually recursive constants.

4.2. Control Flow

4.2.1. Blocks

Every block in Aiken is an expression. All expressions in the block are executed, and the result of the last expression is returned.

```
let value: Bool = {
    "Hello"
    42 + 12
    False
}

value == False
```

4.2.2. If-Else

Pattern matching on a Bool value is discouraged and if *condition* else expressions should be used instead.

```
let some_bool = True

if some_bool {
  "It's true!"
} else {
  "It's not true."
}
```

Note that, while it may look like an imperative instruction: if this then do that or else do that, it is in fact one single expression. This means, in particular, that the return types of both branches have to match.

Further reading: *https://aiken-lang.org/language-tour/control-flow*

4.3. Intro to Aiken Standard Library

Before diving into the standard library, let's review some Aiken commands.

- **aiken new:** Create a new Aiken project
- **aiken c:** Type-check an Aiken project and run any tests found
- **aiken b:** Build an Aiken project
- **aiken add:** Add a new project package as dependency
- **aiken docs:** Build the documentation for an Aiken project

The official standard library (*stdlib*) for the Aiken Cardano smart-contract language. It extends the language builtins with useful data-types, functions, constants and aliases that make using Aiken a bliss. To add to your project, you run:

```
aiken add aiken-lang/stdlib --version v2
```

Example Usage:

```
use aiken/collection/list
use aiken/crypto.{VerificationKeyHash}
use cardano/transaction.{OutputReference, Transaction}

pub type Datum {
  owner: VerificationKeyHash,
}

pub type Redeemer {
  msg: ByteArray,
}

/// A simple validator which replicates a basic public/private
signature lock.
///
/// - The key (hash) is set as datum when the funds are sent to the
```

```
script address.
/// - The spender is expected to provide a signature, and the string
'Hello, World!' as message
/// - The signature is implicitly verified by the ledger, and
included as 'extra_signatories'
///
validator hello_world {
  spend(datum: Option<Datum>, redeemer: Redeemer, _, self:
Transaction) {
    expect Some(Datum { owner }) = datum

    let must_say_hello = redeemer.msg == "Hello, World!"

    let must_be_signed = list.has(self.extra_signatories, owner)

    and {
      must_say_hello,
      must_be_signed,
    }
  }
}
```

In the line below for example:

```
let must_be_signed = list.has(self.extra_signatories, owner)
```

You can see that *list* from the stdlib is used as a utility function to check if an item is a member of a list.

Checkout The Standard Library: *https://aiken-lang.github.io/stdlib/*

4.4. Intro Aiken Validators

A validator is a named block that contains one or more handlers. The handler name must match Cardano's well-known purposes:

- mint, spend, withdraw, publish, vote or propose.

Every handler is a predicate function: they must return True or False. When True, they authorize the action they are validating. Alternatively, to return false, they can instead halt using the *fail* keyword or via an invalid *expect* assignment.

With the exception of the spend handler, each handler is a function with exactly three arguments:

- A **redeemer**, which is a user-defined type and value.
- A **target**, whose type depends on the purpose.
- A **transaction**, which represents the script execution context.

The spend handler takes an additional first argument which is an optional datum, which is also a user-defined type.

Handler	What for
mint	Minting / burning of assets
spend	Spending of transaction outputs
withdraw	Withdrawing staking rewards
publish	Publishing of delegation certificates
vote	Voting on governance proposals
propose	Constitution guardrails, executed when submitting governance proposals

```
use cardano/address.{Credential}
use cardano/assets.{PolicyId}
use cardano/certificate.{Certificate}
use cardano/governance.{ProposalProcedure, Voter}
use cardano/transaction.{Transaction, OutputReference}

validator my_script {
  mint(redeemer: MyMintRedeemer, policy_id: PolicyId, self: Transaction) {
    todo @"mint logic goes here"
  }

  spend(datum: Option<MyDatum>, redeemer: MySpendRedeemer, utxo:
OutputReference, self: Transaction) {
    todo @"spend logic goes here"
  }

  withdraw(redeemer: MyWithdrawRedeemer, account: Credential, self:
Transaction) {
    todo @"withdraw logic goes here"
  }

  publish(redeemer: MyPublishRedeemer, certificate: Certificate, self:
Transaction) {
    todo @"publish logic goes here"
  }

  vote(redeemer: MyVoteRedeemer, voter: Voter, self: Transaction) {
    todo @"vote logic goes here"
  }

  propose(redeemer: MyProposeRedeemer, proposal: ProposalProcedure, self:
Transaction) {
    todo @"propose logic goes here"
  }
}
```

Further reading: *https://aiken-lang.org/language-tour/validators*

4.5. Hello world Aiken Program; Always True Validator

```
use aiken/collection/list
use aiken/crypto.{VerificationKeyHash}
use cardano/transaction.{OutputReference, Transaction}

pub type Datum {
  owner: VerificationKeyHash,
}

pub type Redeemer {
  msg: ByteArray,
}

validator hello_world {
  spend(
    datum: Option<Datum>,
    redeemer: Redeemer,
    _own_ref: OutputReference,
    self: Transaction,
  ) {
    expect Some(Datum { owner }) = datum
    let must_say_hello = redeemer.msg == "Hello, World!"
    let must_be_signed = list.has(self.extra_signatories, owner)
    must_say_hello && must_be_signed
  }
}
```

Our first validator is rudimentary, yet there's already a lot to say about it.

It looks for a verification key hash (owner) in the datum and a message (msg) in the redeemer. Remember that, in the eUTxO model, the datum is set when locking funds in the contract and can be therefore seen as configuration. Here, we'll indicate the owner of the contract and require a signature from them to unlock funds—very much like it already works on a typical non-script address.

Moreover, because there's no "Hello, World!" without a proper "Hello, World!" our little contract also demands this very message, as a UTF-8-encoded byte array, to be passed as redeemer (i.e. when spending from the contract).

It's now time to build our first contract!

Further reading: *https://aiken-lang.org/example--hello-world/basics*