Building Example Code and Learning Plutus Playground

Contributed By:
Joe Totes

Table of Contents

Preparation for Lecture 1

Before we can get started in lecture 1, we first must get our development environment configured. This guide will be using a fresh install of Ubuntu linux.

If you want to use linux but only have a computer with Windows, you can run a virtual environment inside of Windows. A great step by step guide for how to get started can be found here:
How to install an Ubuntu VM in Windows

You can copy and paste any of the code in this guide directly into your terminal or IDE. If you are new to linux and are unfamiliar with terminal shell commands, this cheat sheet gives a quick overview:
Linux Command Master List

The haddock documentation is also a great source of information for all the public plutus libraries. This can be found here:
Documentation for all public Plutus Libraries

First, Open up the terminal to get started. We will first install the necessary dependencies first for a fresh copy of linux.

We need to install Nix and get it configured properly to use IOG's caches. In this guide we will be doing a single user install.

Before we can install Nix, we need to make sure the version of linux you are using has curl installed. First run:

```
totinj@penguin:~$ sudo sh -c 'apt update && apt install curl'
```

Now that curl is installed, we can now install Nix. Run:

```
totinj@penguin:~$ sh <(curl -L https://nixos.org/nix/install) --no-daemon
```

```
Output:

Installation finished!  To ensure that the necessary environment
variables are set, either log in again, or type

  . /home/totinj/.nix-profile/etc/profile.d/nix.sh
```

Now to finish, we need to set the environment with the following command notice from above.

<u>Very important here to replace “totinj” with your current linux user!!</u>

```
totinj@penguin:~$ . /home/totinj/.nix-profile/etc/profile.d/nix.sh
```

We now need to add Input Outputs caches to greatly speed up the building process. Without this step, you might be running nix-shell for days rather than minutes! Let's create a new config file that has the associated IOG links. Run:

```
totinj@penguin:~$ mkdir ~/.config/nix
echo 'substituters = https://hydra.iohk.io https://iohk.cachix.org
https://cache.nixos.org/' >> ~/.config/nix/nix.conf
echo 'trusted-public-keys =
hydra.iohk.io:f/Ea+s+dFdN+3Y/G+FDgSq+a5NEWhJGzdjvKNGv0/EQ=
iohk.cachix.org-1:DpRUyj7h7V830dp/i6Nti+NEO2/nhblbov/8MW7Rqoo=
cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY=' >>
~/.config/nix/nix.conf
```

With Nix now installed and configured, we will clone the appropriate repositories from github. We will be cloning the plutus-apps and the plutus-pioneer program.

First, let's clone plutus-apps:

```
totinj@penguin:~$ git clone
https://github.com/input-output-hk/plutus-apps.git
```

Next, let's clone the plutus-pioneer-program repo:

```
totinj@penguin:~$ git clone
https://github.com/input-output-hk/plutus-pioneer-program.git
```

You can now navigate to the current week01 directory in the plutus-pioneer-program folder and open the cabal.project file:

```
totinj@penguin:~/plutus-pioneer-program/code/week01$ cat cabal.project
```

Grab the plutus-apps tag inside the cabal.project file:

```
location: https://github.com/input-output-hk/plutus-apps.git
  tag:41149926c108c71831cfe8d244c83b0ee4bf5c8a
```

Head back to to the plutus-apps directory and update it to the current git tag:

```
totinj@penguin:~/plutus-apps$ git checkout main
```

```
totinj@penguin:~/plutus-apps$ git pull
```

```
totinj@penguin:~/plutus-apps$ git checkout
41149926c108c71831cfe8d244c83b0ee4bf5c8a
```

You should now be up to date and can run nix-shell in this directory. Run nix-shell:

```
totinj@penguin:~/plutus-apps$ nix-shell
```

Nix-shell will take a good amount of time to build the first time you are running it, so be patient. If you have setup your caches correctly, you will notice it building from https://hydra.iohk.io.

If successful, you should see the nix-shell:

```
[nix-shell:~/plutus-apps]$
```

Head back to the week01 folder to start running the cabal commands:

```
[nix-shell:~/plutus-pioneer-program/code/week01]$ cabal update
```

```
[nix-shell:~/plutus-pioneer-program/code/week01]$ cabal build
```

```
[nix-shell:~/plutus-pioneer-program/code/week01]$ cabal repl
```

These will also take a long time to run the first time. If successful, you should now be ready to start the lecture:

```
Ok, one module loaded.
Prelude Week01.EnglishAuction>
```

The EUTxO Model

This is the transcript for the lecture video on EUTxOs by Lars Brünjes. Further information about EUTxO models can be found here: [Accounting Systems for Blockchains]

One of the most important things you need to understand in order to write Plutus smart contracts is the accounting model that Cardano uses; and that is the so-called (E)UTxO model, which is an abbreviation for extended unspent transaction output model. The UTxO model without being extended is the one that has been introduced by Bitcoin. But there are other models. Ethereum, for example, uses a so-called account-based model, which is what you're used to from a normal bank, where everybody has an account and each account has a balance. And if you transfer money from one account to another, then the balance gets updated accordingly, but that is not how the UTxO model works. Unspent transaction outputs are exactly what the name says. They are transaction outputs that are outputs from previous transactions that happened on the blockchain that have not yet been spent.

So let's look at an example where we have two such UTxOs, one belonging to Alice, 100 ADA and another one belonging to Bob, 50 ADA. And as an example, let's assume that Alice wants to send 10 ADA to Bob. So she creates a transaction and the transaction is something that has inputs and outputs, can be an arbitrary number of inputs and an arbitrary number of outputs. And an important thing is that you can always only use complete UTxOs as input. So, if she wants to send 10 ADA to Bob, she can't simply split herexisting 100 ADA into a 90 to 10 piece. She has to use the full 100 ADA as input. So by using the UTxO 100 ADA as input to a transaction. Alice has not spent that UTxO, so it's no longer an UTxO. It's no longer unspent, it's been spent. And now she can create outputs for a transaction. So she wants to pay 10 ADA to Bob. So one output will be 10 ADA to Bob, and then she wants her change back. So she creates a second output of 90 ADA to herself.

And so this is how, even though you always have to consume complete UTxOs, you can get your change back. So you consume the complete UTxO, but then you create an output for the change and note that in a transaction, the sum of the input values must equal the sum of the output values. So in this case, 100 ADA go in and 10 plus 90 ADA go out. This is strictly speaking, not true. There are two exceptions, the first exception is transaction fees. So in the real blockchain for each transaction, you have to pay fees. So that means that the sum of input values has to be slightly higher than the sum of output values to accommodate for the fees. And the second exception is the native tokens that we have on Cardano. So it's possible for transactions to create new tokens. In which case the outputs will be higher than the inputs or to burn tokens, in which case the inputs will be higher than the outputs. But that is a somewhat advanced topic, how to handle minting and burning of native tokens in Plutus. And we'll come back to that later in the course. So for now we only look at transactions where the sum of the input value equals the sum of the output values.

So this is a first example of a simple transaction, and we see that the effect of a transaction is to consume and spend transaction output and to produce new ones. So in this example, one UTxO has been consumed, Alice original 100 ADA UTxO, and two new ones have been created. One 90 ADA UTxO belongs to Alice and another 10 ADA UTxO belongs to Bob. It's important to note that this is the only thing that happens on an UTxO blockchain. The only thing that happens when a new transaction is added to the blockchain is that someform a UTxOs becomes spent and UTxOs appear. So in particular, nothing is ever changed, no value or any other data associated with the transaction output is ever changed. The only thing that changes by a new transaction is that some of the formerly unspent transaction outputs disappear and others are created, but the outputs themselves never change. The only thing that changes is whether they are unspent or not.

Let's do one other example, a slightly more complicated one where Alice and Bob together want to pay 55 ADA each to Charlie. So they create a transaction together. And as inputs, Alice has no choice, she only has one UTxO, so she uses that one. And Bob also doesn't have a choice because neither of his two UTxOs is large enough to cover 55 ADA. So Bob has to use both his UTxOs as input. This time we need three outputs, one the 55 plus 55 equals 110 ADA for Charlie, and the two change outputs, one for Alice's change and one for Bob's change. So Alice paid 90, so she should get 35 change and Bob paid 60. So he should get five change. One thing I haven't yet explained is under which conditions a transaction can spend a given UTxO. Obviously it wouldn't be a good idea if any transaction could spend arbitrary UTxOs, if that was the case, then Bob could spend Alice's money without her consent. So the way it works is by adding signatures to transactions, so for our first example, our transaction one, because that consumes an UTxO belonging to Alice as input. Alice's signature has to be added to the transaction. And in the second example, because there are inputs belonging to both Alice and Bob, both Alice and Bob have to sign that transaction, which incidentally is something you can't do in Daedalus. So you would have to use the Cardano CLI for complex transactions like that.

Everything I've explained so far is just about the UTxO model, not the extended UTxO model. So this is all just a simple UTxO model. And the extended part comes in when we talk about smart contracts. So in order to understand that, let's just concentrate on one consumption on a UTxO by an input. And as I just explained, the validation that decides whether the transaction this input belongs to is allowed to consume that you take so in the simple UTxO model relies on digital signatures. So in this case, Alice has to sign the transaction for this consumption of the UTxO to be valid. And now the idea of the extended UTxO model is to make this more general. So instead of just having one condition, namely that the appropriate signature is present in the transaction. We replace this by arbitrary logic, and this is where Plutus comes in. So instead of just having an address that corresponds to a public key, and that can be verified by a signature that is added to the transaction, instead we have more general addresses that are not based on public keys or the hashes of public keys, but instead contain arbitrary logic that can decide under which condition this specific UTxO can be spent by a transaction. So instead of an address going to a public key, like Alice's public key in this example, there will be an arbitrary script, a script containing arbitrary logic. And instead of the

signature in the transaction, the input will justify that it is allowed to consume this output with some arbitrary piece of data that we call the redeemer. So we replace the public key address.

Alice in our example by a script, we place a digital signature by a redeemer which is an arbitrary piece of data. Now, the next question is, what exactly does that mean? What do we mean by arbitrary logic? And in particular it's important to consider what information? What context does this script have? So there are several options. And the one indicated in this diagram is that all the script sees is the redeemer. So all the information the script has in order to decide whether it's okay for the transaction to consume this UTxO or not is looking at the redeemer. And that is the thing that Bitcoin incidentally does.So, in Bitcoin, there are smart contracts, they are just not very smart.They are called Bitcoin script and Bitcoin script works exactly like this. So there's a script on the UTxO site and a redeemer on the input side and the script gets the redeemer and can use the redeemer to decide whether it's okay to consume the UTxO or not. But that's not the only option, we can decide to give more information to the script. So, Ethereum uses a different concept. In Ethereum the script basically can see everything, the whole blockchain,the whole state of the blockchain. So that's like the opposite extreme of Bitcoin. Bitcoin the script has very little context, all it can see is the redeemer. In Ethereum the solidity scripts in Ethereum can see the complete state of the blockchain. So that enables Ethereum's scripts to be much more powerful so they can do basically everything, but it also comes with problems because the scripts are so powerful, it's also very difficult to predict what a given script will do and that opens the door to all sorts of security issues and dangerous, because it's very hard to predict for the developers of an Ethereum smart contract what can possibly happen because there are so many possibilities.

So what Cardano does is something in the middle, so it doesn't offer such a restricted view as Bitcoin, but also does not have a global view as Ethereum, but instead chooses a middle ground. So the script can see the whole blockchain, can see the state of the word blockchain, but it can't see the whole transaction that is being validated. So, in contrast to Bitcoin it can just see this one input, the redeem of this one input, but it can see that and all the other inputs of the transaction and also all the outputs of the transaction and the transaction itself, and the Plutus script can use that information to decide whether it's okay to consume this output.

Now, in this example, there's only one input, but if this transaction had more than one input, then the script would be able to see those as well. There's one last ingredient that Plutus scripts need in order to be as powerful and expressive as Ethereum scripts. And that is a so-called datum which is a piece of data that can be associated with a UTxO in addition to the value. So at a script address, like in this example, in addition to this 100 ADA value, that can be an arbitrary piece of data attached, which we call datum. And with this, we can actually mathematically prove that Plutus is at least as powerful as Ethereum, so everything, every logic you can express in Ethereum you can also express in this extended UTxO model that Cardano uses, but it has a lot of important advantages in comparison to the Ethereum model. So for example, in Plutus, it is possible to check whether a transaction will validate in your wallet before you ever sent it to the chain. So something can still go wrong, so for example, your transaction can consume an output and then when it gets to the chain, somebody else has already consumed that output.This output has already been consumed by another

transaction.You can't prevent that, but in that case, your transaction will simply fail without you having to pay any fees. But if all the inputs are still there, that your transaction expects, then you can be sure that the transaction will validate and that it will have the effect that you predicted when you ran it in your wallet .

This is definitely not the case in Ethereum, in Ethereum in the time between you constructing the transaction and it being incorporated into the blockchain, a lot of stuff can happen concurrently and that's unpredictable, and that can have unpredictable effects on what will happen when your script eventually executes. So that means in Ethereum it's always possible that you have to pay gas fee for a transaction, although the transaction eventually fails with an error, and that is guaranteed not to happen in Cardano. In addition to that, it's also easier to analyze a Plutus script and to check or even proof that it is secure because you don't have to consider the whole state of the blockchain, which is unknowable. You can concentrate on this context that just consists of the spending transaction. So you have a much more limited scope and that makes it much easier to understand what a script is actually doing and what can possibly happen or what could possibly go wrong.So this is it, that's the extended UTxO model that Plutus uses.

So to recapitulate in extending the normal UTxO model, we replace public key addresses from the normal UTxO model with scripts, Plutus scripts, and instead of legitimizing the consumption of new UTxO by digital signatures, as in the simple UTxO model, arbitrary data called redeemer is used on the input side. And we also add arbitrary custom data on the output side. And the script as context when it runs, sees the spending transaction, the transaction one, in this example. So given the redeemer and the datum and the transaction with its other inputs and outputs, the script can run arbitrary logic to decide whether it's okay for this transaction to consume the output or not. And that is how Plutus works.

One thing I haven't mentioned yet is who is responsible for providing datum, redeemer and the validator, the script that validates whether a transaction can consume an input. And the rule in Plutus is that the spending transaction has to do that whereas the producing transaction only has to provide hashes.So that means if I produce an output that sits at a script address, then this producing transaction only has to include the hash of the script and the hash of the datum that belongs to this output. But optionally, it can include the datum and the script as well, fully, but that's only optional. And if a transaction wants to consume such a script output, then that transaction,the spending transaction has to include the datum and the redeemer and the script. So that's the rule, how it works in Plutus, which of course means that in order to be able to spend a given input, you need to know the datum because only the hash is publicly visible on the blockchain. Which is sometimes a problem and not what you want and that's where this possibility comes into to also include it in the producing transaction. Otherwise only people that know the datum by some other means not by looking at the blockchain would be able to ever spend such an output.

So this is the UTxO model, the extended unspent transaction output model. And that is of course not tied to a specific programming language.I mean, what we have is Plutus, which is based on Haskell, but in principle, you could use the same concept, the same UTxO model with a completely different programming language. And we also plan to write compilers from other programming languages to Plutus script which issort of the assembly language and aligned

Plutus.So there's an extended UTxO model is different from the specific programming language we use.In this course, we will use Plutus obviously, but the understanding the UTxO model is independently valid from understanding Plutus or learning Plutus syntax.

The Auction Contract in the EUTxO Model

The code in Plutus is broken down into on-chain and off-chain code. On-chain code just checks and validates, it just says yes or no. The off-chain code actively creates that translation that will then pass validation. Both of the on-chain and off-chain parts are uniformly written in haskell. This is largely advantageous as code can be shared, and you only need to concern yourself with one programming language.

Looking at the auction contract EnglishAuction.hs, we see the various data types are listed first in the contract:

```
minLovelace :: Integer
minLovelace = 2000000

data Auction = Auction
   { aSeller   :: !PaymentPubKeyHash
   , aDeadline :: !POSIXTime
   , aMinBid   :: !Integer
   , aCurrency :: !CurrencySymbol
   , aToken    :: !TokenName
   } deriving (P.Show, Generic, ToJSON, FromJSON, ToSchema)

instance Eq Auction where
   {-# INLINABLE (==) #-}
   a == b = (aSeller   a == aSeller   b) &&
            (aDeadline a == aDeadline b) &&
            (aMinBid   a == aMinBid   b) &&
            (aCurrency a == aCurrency b) &&
            (aToken    a == aToken    b)

PlutusTx.unstableMakeIsData ''Auction
PlutusTx.makeLift ''Auction

data Bid = Bid
   { bBidder :: !PaymentPubKeyHash
   , bBid    :: !Integer
   } deriving P.Show
```

```
instance Eq Bid where
   {-# INLINABLE (==) #-}
   b == c = (bBidder b == bBidder c) &&
            (bBid    b == bBid    c)

PlutusTx.unstableMakeIsData ''Bid
PlutusTx.makeLift ''Bid

data AuctionAction = MkBid Bid | Close
   deriving P.Show

PlutusTx.unstableMakeIsData ''AuctionAction
PlutusTx.makeLift ''AuctionAction

data AuctionDatum = AuctionDatum
   { adAuction    :: !Auction
   , adHighestBid :: !(Maybe Bid)
   } deriving P.Show

PlutusTx.unstableMakeIsData ''AuctionDatum
PlutusTx.makeLift ''AuctionDatum

data Auctioning
instance Scripts.ValidatorTypes Auctioning where
   type instance RedeemerType Auctioning = AuctionAction
   type instance DatumType Auctioning = AuctionDatum
```

Followed by the main on-chain code (validation):

```
{-# INLINABLE mkAuctionValidator #-}
mkAuctionValidator :: AuctionDatum -> AuctionAction -> ScriptContext -> Bool
mkAuctionValidator ad redeemer ctx =
   traceIfFalse "wrong input value" correctInputValue &&
   case redeemer of
       MkBid b@Bid{..} ->
           traceIfFalse "bid too low"        (sufficientBid bBid)         &&
           traceIfFalse "wrong output datum" (correctBidOutputDatum b)    &&
           traceIfFalse "wrong output value" (correctBidOutputValue bBid) &&
           traceIfFalse "wrong refund"       correctBidRefund             &&
           traceIfFalse "too late"           correctBidSlotRange
       Close           ->
           traceIfFalse "too early" correctCloseSlotRange &&
           case adHighestBid ad of
               Nothing      ->
                   traceIfFalse "expected seller to get token" (getsValue
(aSeller auction) $ tokenValue <> Ada.lovelaceValueOf minLovelace)
               Just Bid{..} ->
                   traceIfFalse "expected highest bidder to get token"
(getsValue bBidder $ tokenValue <> Ada.lovelaceValueOf minLovelace) &&
                   traceIfFalse "expected seller to get highest bid" (getsValue
(aSeller auction) $ Ada.lovelaceValueOf bBid)

 where
   info :: TxInfo
   info = scriptContextTxInfo ctx

   input :: TxInInfo
   input =
     let
       isScriptInput i = case (txOutDatumHash . txInInfoResolved) i of
           Nothing -> False
           Just _  -> True
       xs = [i | i <- txInfoInputs info, isScriptInput i]
     in
       case xs of
           [i] -> i
           _   -> traceError "expected exactly one script input"
```

```
    inVal :: Value
    inVal = txOutValue . txInInfoResolved $ input

    auction :: Auction
    auction = adAuction ad

    tokenValue :: Value
    tokenValue = Value.singleton (aCurrency auction) (aToken auction) 1

    correctInputValue :: Bool
    correctInputValue = inVal == case adHighestBid ad of
        Nothing      -> tokenValue <> Ada.lovelaceValueOf minLovelace
        Just Bid{..} -> tokenValue <> Ada.lovelaceValueOf (minLovelace + bBid)

    sufficientBid :: Integer -> Bool
    sufficientBid amount = amount >= minBid ad

    ownOutput   :: TxOut
    outputDatum :: AuctionDatum
    (ownOutput, outputDatum) = case getContinuingOutputs ctx of
        [o] -> case txOutDatumHash o of
            Nothing   -> traceError "wrong output type"
            Just h -> case findDatum h info of
                Nothing        -> traceError "datum not found"
                Just (Datum d) ->  case PlutusTx.fromBuiltinData d of
                    Just ad' -> (o, ad')
                    Nothing  -> traceError "error decoding data"
        _   -> traceError "expected exactly one continuing output"

    correctBidOutputDatum :: Bid -> Bool
    correctBidOutputDatum b = (adAuction outputDatum == auction)   &&
                              (adHighestBid outputDatum == Just b)

    correctBidOutputValue :: Integer -> Bool
    correctBidOutputValue amount =
        txOutValue ownOutput == tokenValue <> Ada.lovelaceValueOf (minLovelace +
amount)

    correctBidRefund :: Bool
    correctBidRefund = case adHighestBid ad of
        Nothing      -> True
        Just Bid{..} ->
          let
            os = [ o
```

```
                | o <- txInfoOutputs info
                , txOutAddress o == pubKeyHashAddress bBidder Nothing
                ]
        in
          case os of
              [o] -> txOutValue o == Ada.lovelaceValueOf bBid
              _   -> traceError "expected exactly one refund output"

   correctBidSlotRange :: Bool
   correctBidSlotRange = to (aDeadline auction) `contains` txInfoValidRange
info

   correctCloseSlotRange :: Bool
   correctCloseSlotRange = from (aDeadline auction) `contains` txInfoValidRange
info

   getsValue :: PaymentPubKeyHash -> Value -> Bool
   getsValue h v =
     let
       [o] = [ o'
             | o' <- txInfoOutputs info
             , txOutValue o' == v
             ]
     in
       txOutAddress o == pubKeyHashAddress h Nothing
```

Next is where the compilation happens:

```
typedAuctionValidator :: Scripts.TypedValidator Auctioning
typedAuctionValidator = Scripts.mkTypedValidator @Auctioning
   $$(PlutusTx.compile [|| mkAuctionValidator ||])
   $$(PlutusTx.compile [|| wrap ||])
 where
   wrap = Scripts.wrapValidator @AuctionDatum @AuctionAction
```

After that follows with the off-chain code starting with the three endpoints start, bid, and close.

```
data StartParams = StartParams
   { spDeadline :: !POSIXTime
   , spMinBid   :: !Integer
   , spCurrency :: !CurrencySymbol
   , spToken    :: !TokenName
   } deriving (Generic, ToJSON, FromJSON, ToSchema)

data BidParams = BidParams
   { bpCurrency :: !CurrencySymbol
   , bpToken    :: !TokenName
   , bpBid      :: !Integer
   } deriving (Generic, ToJSON, FromJSON, ToSchema)

data CloseParams = CloseParams
   { cpCurrency :: !CurrencySymbol
   , cpToken    :: !TokenName
   } deriving (Generic, ToJSON, FromJSON, ToSchema)

type AuctionSchema =
       Endpoint "start" StartParams
   .\/ Endpoint "bid"   BidParams
   .\/ Endpoint "close" CloseParams

start :: AsContractError e => StartParams -> Contract w s e ()
start StartParams{..} = do
   pkh <- ownPaymentPubKeyHash
   let a = Auction
               { aSeller   = pkh
```

```
                , aDeadline = spDeadline
                , aMinBid   = spMinBid
                , aCurrency = spCurrency
                , aToken    = spToken
                }
        d = AuctionDatum
                { adAuction    = a
                , adHighestBid = Nothing
                }
        v = Value.singleton spCurrency spToken 1 <> Ada.lovelaceValueOf
minLovelace
        tx = Constraints.mustPayToTheScript d v
    ledgerTx <- submitTxConstraints typedAuctionValidator tx
    void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
    logInfo @P.String $ printf "started auction %s for token %s" (P.show a)
(P.show v)

bid :: forall w s. BidParams -> Contract w s Text ()
bid BidParams{..} = do
    (oref, o, d@AuctionDatum{..}) <- findAuction bpCurrency bpToken
    logInfo @P.String $ printf "found auction utxo with datum %s" (P.show d)

    when (bpBid < minBid d) $
        throwError $ pack $ printf "bid lower than minimal bid %d" (minBid d)
    pkh <- ownPaymentPubKeyHash
    let b  = Bid {bBidder = pkh, bBid = bpBid}
        d' = d {adHighestBid = Just b}
        v  = Value.singleton bpCurrency bpToken 1 <> Ada.lovelaceValueOf
(minLovelace + bpBid)
        r  = Redeemer $ PlutusTx.toBuiltinData $ MkBid b

        lookups = Constraints.typedValidatorLookups typedAuctionValidator P.<>
                  Constraints.otherScript auctionValidator                P.<>
                  Constraints.unspentOutputs (Map.singleton oref o)
        tx      = case adHighestBid of
                    Nothing      -> Constraints.mustPayToTheScript d' v
<>
                                    Constraints.mustValidateIn (to $ aDeadline
adAuction)          <>
                                    Constraints.mustSpendScriptOutput oref r
                    Just Bid{..} -> Constraints.mustPayToTheScript d' v
<>
                                    Constraints.mustPayToPubKey bBidder
(Ada.lovelaceValueOf bBid) <>
```

```
                                    Constraints.mustValidateIn (to $ aDeadline
adAuction)          <>
                                    Constraints.mustSpendScriptOutput oref r
   ledgerTx <- submitTxConstraintsWith lookups tx
   void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
   logInfo @P.String $ printf "made bid of %d lovelace in auction %s for token
(%s, %s)"
       bpBid
       (P.show adAuction)
       (P.show bpCurrency)
       (P.show bpToken)

close :: forall w s. CloseParams -> Contract w s Text ()
close CloseParams{..} = do
   (oref, o, d@AuctionDatum{..}) <- findAuction cpCurrency cpToken
   logInfo @P.String $ printf "found auction utxo with datum %s" (P.show d)

   let t      = Value.singleton cpCurrency cpToken 1
       r      = Redeemer $ PlutusTx.toBuiltinData Close
       seller = aSeller adAuction

       lookups = Constraints.typedValidatorLookups typedAuctionValidator P.<>
                 Constraints.otherScript auctionValidator                 P.<>
                 Constraints.unspentOutputs (Map.singleton oref o)
       tx      = case adHighestBid of
                   Nothing      -> Constraints.mustPayToPubKey seller (t <>
Ada.lovelaceValueOf minLovelace)  <>
                                   Constraints.mustValidateIn (from $ aDeadline
adAuction)                    <>
                                   Constraints.mustSpendScriptOutput oref r
                   Just Bid{..} -> Constraints.mustPayToPubKey bBidder (t <>
Ada.lovelaceValueOf minLovelace) <>
                                   Constraints.mustPayToPubKey seller
(Ada.lovelaceValueOf bBid)             <>
                                   Constraints.mustValidateIn (from $ aDeadline
adAuction)                    <>
                                   Constraints.mustSpendScriptOutput oref r
   ledgerTx <- submitTxConstraintsWith lookups tx
   void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
   logInfo @P.String $ printf "closed auction %s for token (%s, %s)"
       (P.show adAuction)
       (P.show cpCurrency)
       (P.show cpToken)
```

```
findAuction :: CurrencySymbol
           -> TokenName
           -> Contract w s Text (TxOutRef, ChainIndexTxOut, AuctionDatum)
findAuction cs tn = do
   utxos <- utxosAt $ scriptHashAddress auctionHash
   let xs = [ (oref, o)
            | (oref, o) <- Map.toList utxos
            , Value.valueOf (_ciTxOutValue o) cs tn == 1
            ]
   case xs of
       [(oref, o)] -> case _ciTxOutDatum o of
           Left _          -> throwError "datum missing"
           Right (Datum e) -> case PlutusTx.fromBuiltinData e of
               Nothing -> throwError "datum has wrong type"
               Just d@AuctionDatum{..}
                   | aCurrency adAuction == cs && aToken adAuction == tn ->
return (oref, o, d)
                   | otherwise                                           ->
throwError "auction token missmatch"
       _           -> throwError "auction utxo not found"

endpoints :: Contract () AuctionSchema Text ()
endpoints = awaitPromise (start' `select` bid' `select` close') >> endpoints
 where
   start' = endpoint @"start" start
   bid'   = endpoint @"bid"   bid
   close' = endpoint @"close" close

mkSchemaDefinitions ''AuctionSchema

myToken :: KnownCurrency
myToken = KnownCurrency (ValidatorHash "f") "Token" (TokenName "T" :| [])
```

The Auction Contract in Plutus Playground

In order to get started with Plutus Playground, we need to have two terminals running, both of which are in the nix-shell.

Let's get started with terminal 1. Head to the plutus-apps directory and first run nix-shell:

Terminal 1

```
totinj@penguin:~/plutus-apps$ nix-shell
```

Next we head to plutus-playground-server directory and run:

Terminal 1

```
[nix-shell:~/plutus-apps/plutus-playground-server]$
plutus-playground-server
```

If Successful, you will see the output:

Terminal 1

```
Interpreter Ready
```

Let's get started with terminal 2. Head to the plutus-apps directory and first run nix-shell:

Terminal 2

```
totinj@penguin:~/plutus-apps$ nix-shell
```

Next we head to plutus-playground-client directory and run:

Terminal 2

```
[nix-shell:~/plutus-apps/plutus-playground-client]$ npm run start
```

If Successful, you will see the output:

Terminal 2

```
[wdm]: Compiled successfully.

or

[wdm]: Compiled with warnings.
```

Keep both terminals open, and we should now be able to access Plutus Playground from the browser.

Open a browser and head to the address:

```
https://localhost:8009
```

You will get a warning complaining about it being a risky website, ignore the message to click through anyway.

You should now be able to successfully compile and run the auctions contract by using the two buttons in the top right corner: "Compile" and "Simulate".

The next part is taken from reddit (u/RikAlexander) which walks through plutus playground. Credit to him for this submission:

Press "simulate" (blue button in the top right).

This opens the Simulate window, where we can try out our newly compiled contract.

This defaults to 2 wallets, to make things interesting though, add another one. (the big "add wallet" button)

The whole idea of this contract is to auction off an NFT (Non-Fungible Token).

Each wallet has 10 lovelaces, and 10 T (T is the Token here).

Change the total of T's for Wallet1 to 1, and for Wallet2 and 3, to 0. (if there were more than 1 it wouldn't be an NFT ofcourse)

Simply put: **Wallet1 is going to put up for auction 1T, and Wallet2-3 will be bidding.**

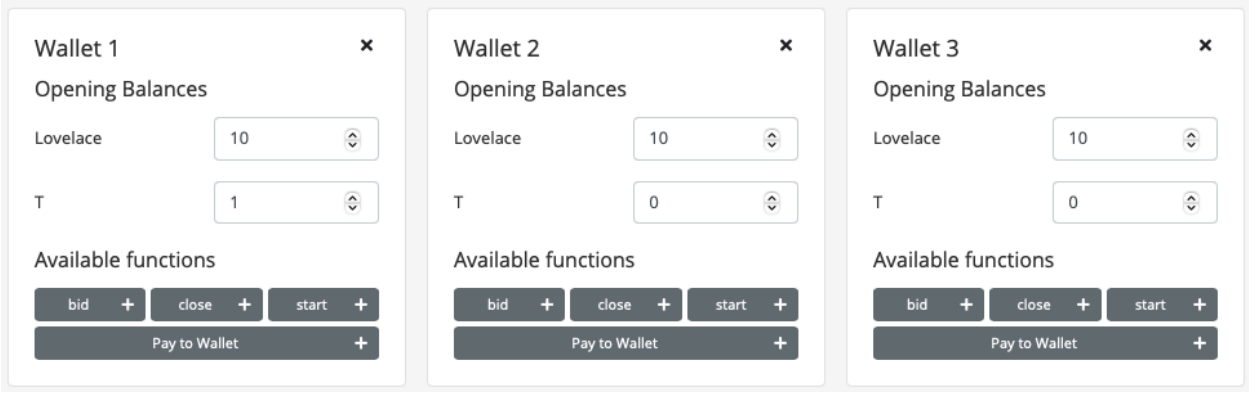
As you can see, each wallet has the functions "bid", "close" and "start".

Bid -> places a bid of x lovelaces

Start -> starts the bidding procedure with getSlot (how long will the bidding last for), spMinBid (minimal lovelaces required)

Close -> closes the bidding; gives the highest bidder its NFT/Token

Note: "pay to wallet" is always there, don't worry about that now :)



6 - Wallet1 is going to put the Token up for auction, we'll do this by pressing the "start" button at Wallet1.

This will add an Action to the Action Sequence.

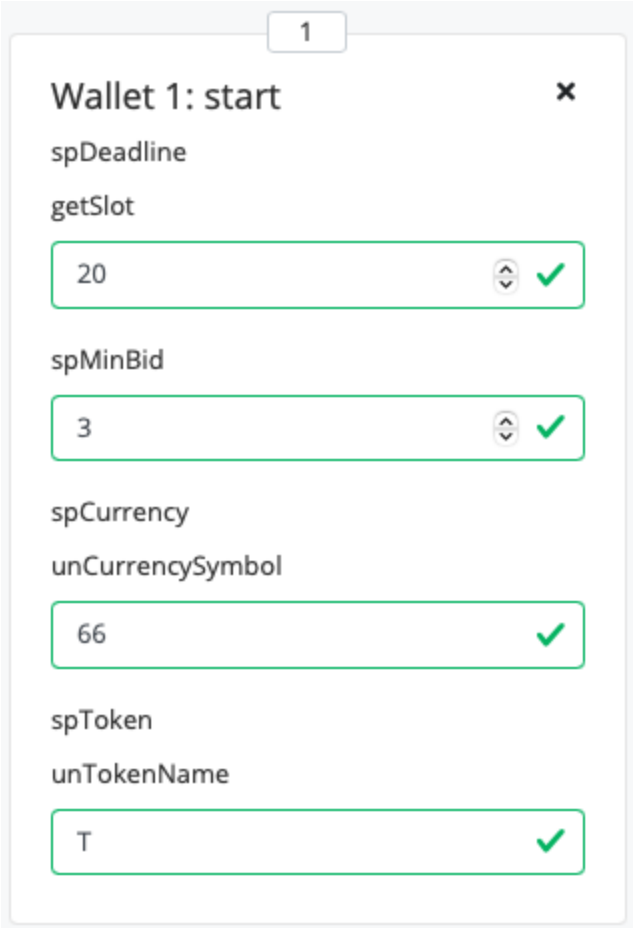
Here we need to set the parameters:

getSlot: 20 (the bidding will close on slot 20)

spMinBid: 3 (atleast 3 lovelaces are required)

spCurrency: 66 (the currencysymbol for the T token; will be explained in future lectures)

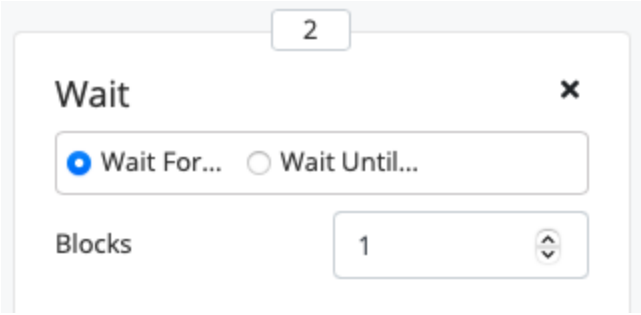
spToken: T (the Token)



Wallet 1

7 - Next we need to add a wait action (1 slot).

This will give all the actions time to be executed.



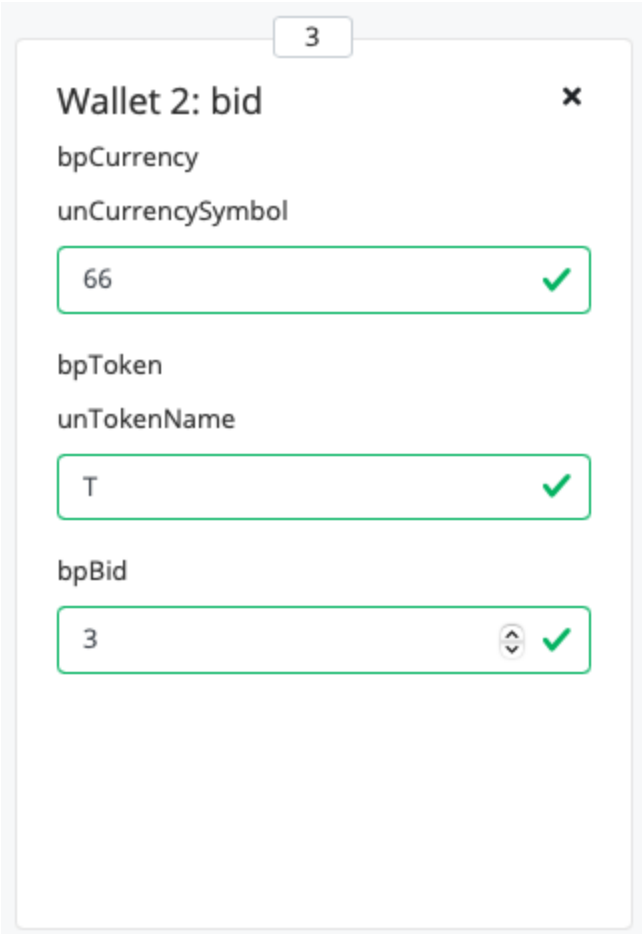
8 - Now for this example Wallet2 will start the bidding with a Bid of 3 lovelaces.

Press the "bid" button at Wallet2, and update the Action with the parameters:

spCurrency: 66 (Same as above)

spToken: T (the Token)

bpBid: 3 (how much lovelaces)



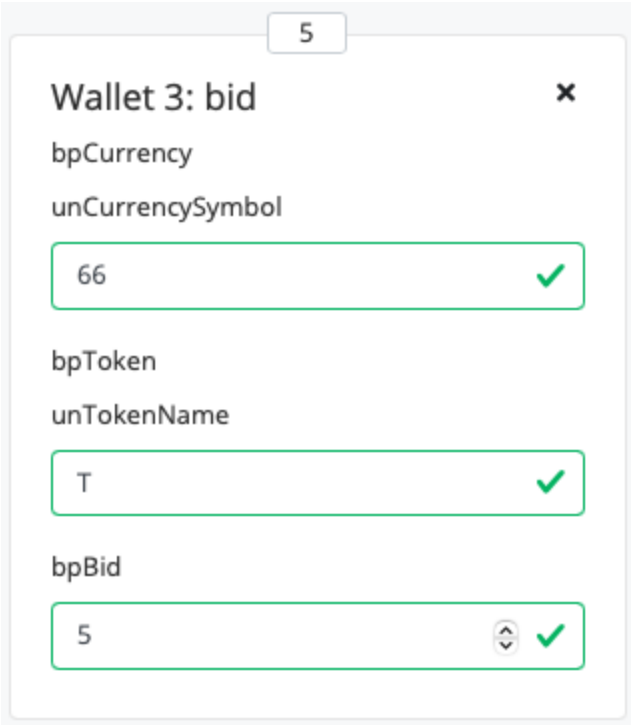
Wallet 2

9 - Insert another wait action here (1 slot)

10 - Now Wallet3 also wants to place a bid.

Same as Wallet2, add a "bid" action, with all the same parameters as above; except for the bpBid parameter.

This could be set to anything (min. 3), but for this example we'll set it to 5.



Wallet 3

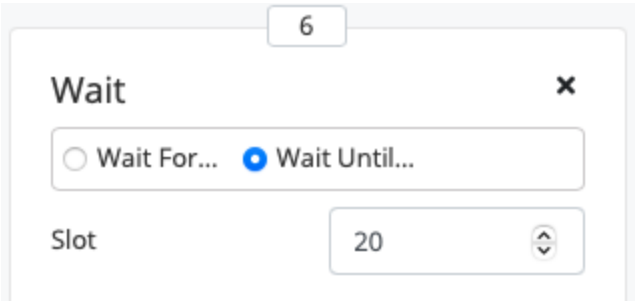
Great. The whole bidding sequence is *DONE.*

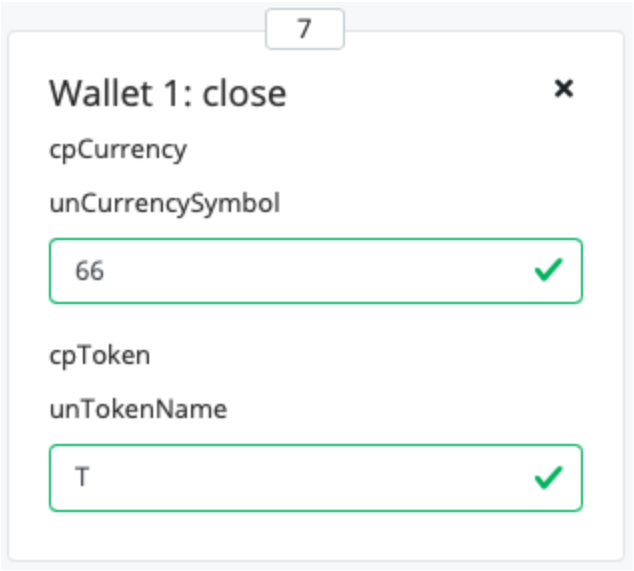
11 - To finish the bidding, we'll add yet another wait action; only this time we'll "wait until" slot 20.

(remember the first action? At slot 20 the bidding will be closed!)

After this the last function (close) still needs to be added, to finalize the bidding sequence.

We will call this from Wallet1, so add the "close" action from Wallet1, with the correct parameters (you know what to do).





12 - Last but not least, we'll add another wait action here (1 slot)

Evaluation

Great we're done with the whole setup!

To execute everything on the simulated blockchain, press the green "Evaluate" button on the bottom of your screen.

On the next screen you'll see the individual slots.

Slot 0, Tx 0 **-> the Genesis slot. This is there to setup everything.**

Wallet1 -> 1T and 10 lovelaces, Wallet2 -> 10 lovelaces, Wallet3 -> 10 lovelaces

Slot 1, Tx 0 **-> The start action, this is where Wallet1 transfers it's 1T (the Token) to the Contract.**

Slot 2, Tx 0 **-> The bid of Wallet2 (3 lovelaces)**

Note: The contract now has 1T and 3 lovelaces

Slot 3, Tx 0 **-> The bid of Wallet3 (5 lovelaces)**

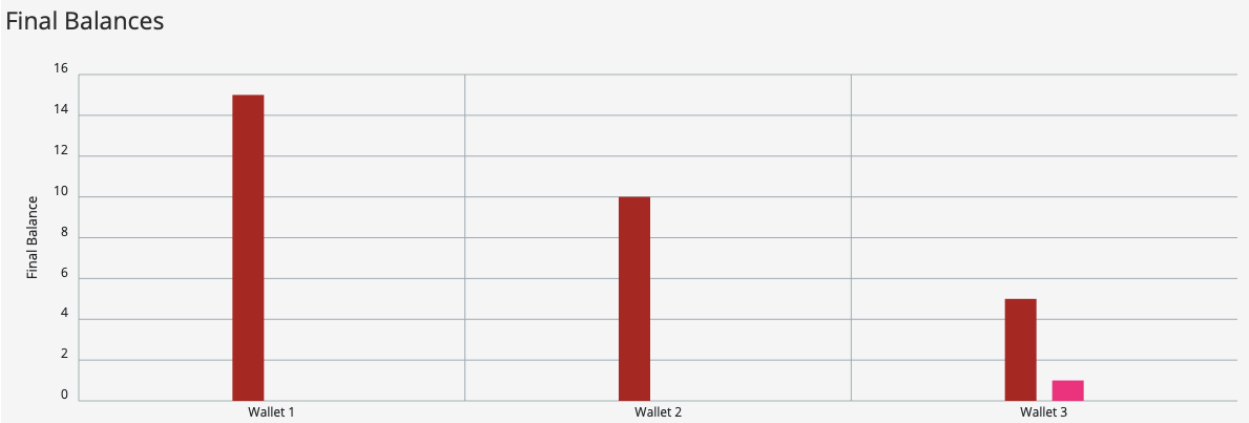
Note: The contract now has 1T and 5 lovelaces; Wallet2 gets it's 3 lovelaces back

Slot 20, Tx 0 **-> Here Wallet3 has won the bidding "war", and is granted its 1T! Also Wallet1 gets its 5 lovelaces :)**

Note: The contract now does not have anything at all :) everything is nicely given to it's rightful owners.

13 - To check the final output of all wallets, scroll down to the "Final balances" section.

As you can see, Wallet3 now has the 1T.



Homework

The objective of the homework this week is to get familiar with running the environment and playing around inside of Plutus Playground. If you have been following the guide up to this point, we should now have the essentials both knowledge and dev environment wise to be ready to move on to lecture 2.