Programming Paradigms

Lecture 7. Input and output in Haskell

Test N°6

See Moodle

Outline

- Recap
- Development environment
- I/O primitives
- Interactive loop
- Managing state
- Lists and programs
- Parsing and handling commands
- Pure vs IO

Development tools

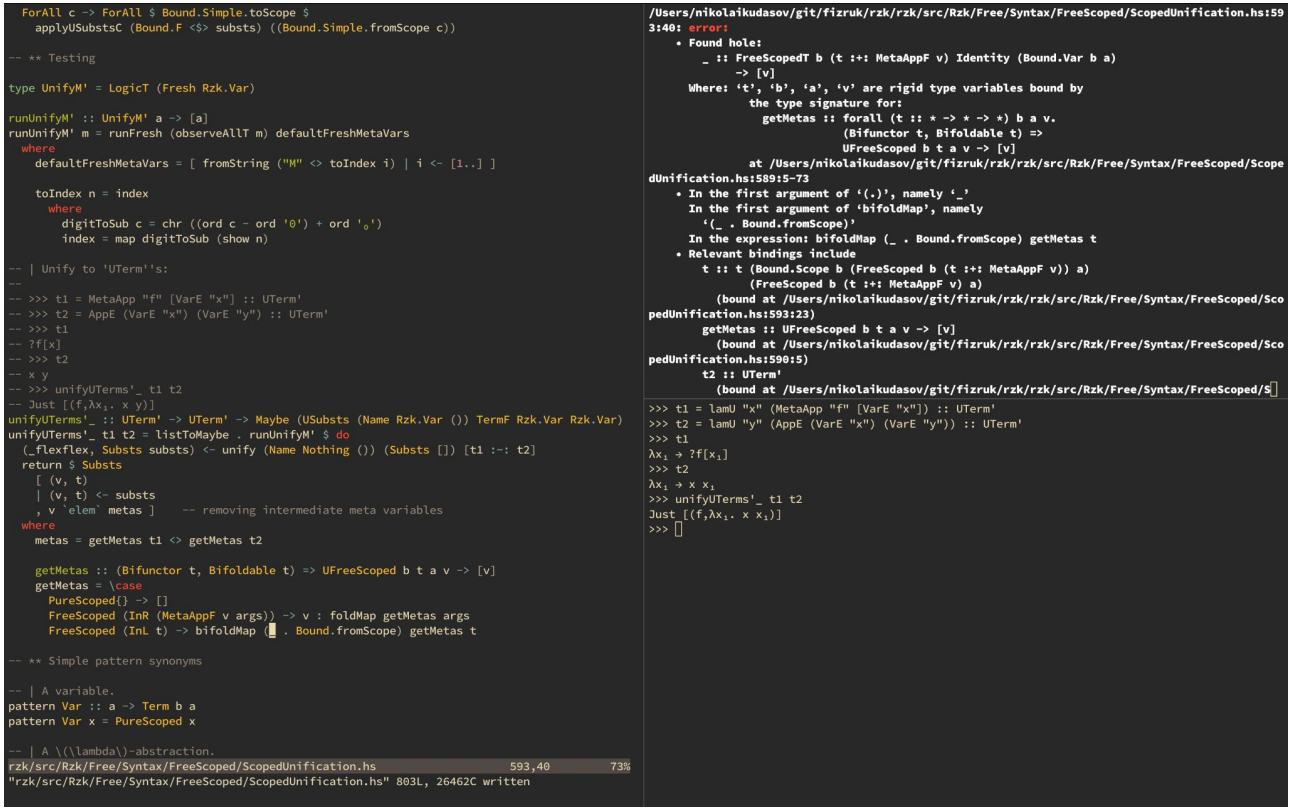
Downloads

This page describes the installation of the Haskell toolchain, which consists of the following tools:

- GHC: the Glasgow Haskell Compiler
- cabal-install: the Cabal installation tool for managing Haskell software
- stack: a cross-platform program for developing Haskell projects
- haskell-language-server (optional): A language server for developers to integrate with their editor/IDE

https://www.haskell.org/downloads/

Development environment (example)



```
  ForAll c -> ForAll $ Bound.Simple.toScope $
    applyUSubstsC (Bound.F <$> substs) ((Bound.Simple.fromScope c))

-- ** Testing

type UnifyM' = LogicT (Fresh Rzk.Var)

runUnifyM' :: UnifyM' a -> [a]
runUnifyM' m = runFresh (observeAllT m) defaultFreshMetaVars
  where
    defaultFreshMetaVars = [ fromString ("M" <> toIndex i) | i <- [1..] ]

    toIndex n = index
      where
        digitToSub c = chr ((ord c - ord '0') + ord '₀')
        index = map digitToSub (show n)

-- | Unify to 'UTerm''s:
--
-- >>> t1 = MetaApp "f" [VarE "x"] :: UTerm'
-- >>> t2 = AppE (VarE "x") (VarE "y") :: UTerm'
-- >>> t1
-- ?f[x]
-- >>> t2
-- x y
-- >>> unifyUTerms'_ t1 t2
-- Just [(f,λx₁. x y)]
unifyUTerms'_ :: UTerm' -> UTerm' -> Maybe (USubsts (Name Rzk.Var ()) TermF Rzk.Var Rzk.Var)
unifyUTerms'_ t1 t2 = listToMaybe . runUnifyM' $ do
  (_flexflex, Substs substs) <- unify (Name Nothing ()) (Substs []) [t1 :~: t2]
  return $ Substs
    [ (v, t)
    | (v, t) <- substs
    , v `elem` metas ]    -- removing intermediate meta variables
  where
    metas = getMetas t1 <> getMetas t2

    getMetas :: (Bifunctor t, Bifoldable t) => UFreeScoped b t a v -> [v]
    getMetas = \case
      PureScoped{} -> []
      FreeScoped (InR (MetaAppF v args)) -> v : foldMap getMetas args
      FreeScoped (InL t) -> bifoldMap (_ . Bound.fromScope) getMetas t

-- ** Simple pattern synonyms

-- | A variable.
pattern Var :: a -> Term b a
pattern Var x = PureScoped x

-- | A \(\lambda\)-abstraction.
rzk/src/Rzk/Free/Syntax/FreeScoped/ScopedUnification.hs                593,40        73%
"rzk/src/Rzk/Free/Syntax/FreeScoped/ScopedUnification.hs" 803L, 26462C written
```

```
/Users/nikolaikudasov/git/fizruk/rzk/rzk/src/Rzk/Free/Syntax/FreeScoped/ScopedUnification.hs:593:40: error:
    • Found hole:
        _ :: FreeScopedT b (t :+: MetaAppF v) Identity (Bound.Var b a)
          -> [v]
      Where: ‘t’, ‘b’, ‘a’, ‘v’ are rigid type variables bound by
               the type signature for:
                 getMetas :: forall (t :: * -> * -> * -> *) b a v.
                             (Bifunctor t, Bifoldable t) =>
                             UFreeScoped b t a v -> [v]
               at /Users/nikolaikudasov/git/fizruk/rzk/rzk/src/Rzk/Free/Syntax/FreeScoped/ScopedUnification.hs:589:5-73
    • In the first argument of ‘(.)’, namely ‘_’
      In the first argument of ‘bifoldMap’, namely
        ‘(_ . Bound.fromScope)’
      In the expression: bifoldMap (_ . Bound.fromScope) getMetas t
    • Relevant bindings include
        t :: t (Bound.Scope b (FreeScoped b (t :+: MetaAppF v)) a)
          (FreeScoped b (t :+: MetaAppF v) a)
          (bound at /Users/nikolaikudasov/git/fizruk/rzk/rzk/src/Rzk/Free/Syntax/FreeScoped/ScopedUnification.hs:593:23)
        getMetas :: UFreeScoped b t a v -> [v]
          (bound at /Users/nikolaikudasov/git/fizruk/rzk/rzk/src/Rzk/Free/Syntax/FreeScoped/ScopedUnification.hs:590:5)
        t2 :: UTerm'
          (bound at /Users/nikolaikudasov/git/fizruk/rzk/rzk/src/Rzk/Free/Syntax/FreeScoped/S
```

```
>>> t1 = lamU "x" (MetaApp "f" [VarE "x"]) :: UTerm'
>>> t2 = lamU "y" (AppE (VarE "x") (VarE "y")) :: UTerm'
>>> t1
λx₁ → ?f[x₁]
>>> t2
λx₁ → x x₁
>>> unifyUTerms'_ t1 t2
Just [(f,λx₁. x x₁)]
>>>
```

Development environment (example)

I/O primitives: Hello, world!

```haskell
module Main where

main :: IO ()
main = putStrLn "Hello, world!"
```

I/O primitives: `IO` type constructor

```
module Main where

main :: IO ()
main = putStrLn "Hello, world!"
```

`data IO a`

`IO a` is the type of *executable programs*
that can perform input/output
and return a value of type `a` as a result

I/O primitives: `IO` type constructor

```haskell
module Main where

main :: IO ()
main = putStrLn "Hello, world!"
```

`data IO a`

`IO ()` is the type of *executable programs*
that can perform input/output
and return a value of type `()` as a result

I/O primitives: `IO` type constructor

```
module Main where

main :: IO ()
main = putStrLn "Hello, world!"
```

Empty tuple
(no interesting values)

`data IO a`

`IO ()` is the type of *executable programs*
that can perform input/output
and return a value of type `()` as a result

I/O primitives: `IO` type constructor

```
module Main where

main :: IO ()
main = putStrLn "Hello, world!"
```

What is the type of `putStrLn`?

I/O primitives: `IO` type constructor

```
module Main where

main :: IO ()
main = putStrLn "Hello, world!"
```

```
putStrLn :: String -> IO ()
```

`putStrLn` is a **pure function**!

I/O primitives: `IO` type constructor

```haskell
module Main where

main :: IO ()
main = putStrLn "Hello, world!"
```

```haskell
putStrLn :: String -> IO ()
```

`putStrLn` is a **pure function**!
It takes a `String` and returns an executable program of type `IO ()`.

I/O primitives: `IO` type constructor

```
module Main where

main :: IO ()
main = putStrLn "Hello, world!"
```

```
putStrLn :: String -> IO ()
```

`putStrLn` is a **pure function**!

It takes a `String` and returns an executable program of type `IO ()`.
This program, when executed (through `main`), will print out the string.

Combining I/O: **do**-notation

```
module Main where

main :: IO ()
main = do
  putStrLn "Hello, world!"
  putStrLn "Goodbye!"
```

do-notation is syntactic sugar,
that allows us to conveniently compose
executable programs into larger programs.

Combining I/O: **do**-notation

```
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

Combining I/O: **do**-notation

```
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

```
getLine :: IO String
```

Combining I/O: **do**-notation

```
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

```
getLine :: IO String
```

`getLine` is a **constant**!

It is an executable program that returns a value of type `String`.

Combining I/O: **do**-notation

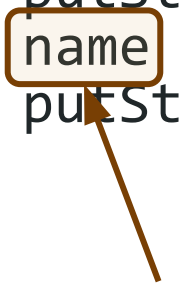
```
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

has type `IO String`

has type `String`

Combining I/O: **do**-notation

```
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```



An **immutable** variable,
available anywhere in the program expressions below

Combining I/O: **do**-notation

```
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

An **immutable** variable,
available anywhere in the program expressions below

```
do
  <expr1>
  x <- <expr2>
  ...
  <exprN>
```

Understanding **do**-notation and `IO` through recipes

Cake layer recipe

1. Mix the dough
2. Split the dough between the pans
3. Bake until golden
4. Cool off for 30 minutes

Frosting recipe

1. Mix butter and cream cheese
2. Add sugar
3. Mix again

Understanding **do**-notation and `IO` through recipes

Cake recipe

1. Prepare **cake layers**
2. Prepare **frosting**
3. Apply `frosting` between `layers`
4. Wait for one hour
5. Cake is ready!

Cake layer recipe

1. Mix the dough
2. Split the dough between the pans
3. Bake until golden
4. Cool off for 30 minutes

Frosting recipe

1. Mix butter and cream cheese
2. Add sugar
3. Mix again

Understanding **do**-notation and `IO` through recipes

Cake recipe

1. `layers <-` Prepare **cake layers**
2. `frosting <-` Prepare **frosting**
3. Apply `frosting` between `layers`
4. Wait for one hour
5. Cake is ready!

Cake layer recipe

1. Mix the dough
2. Split the dough between the pans
3. Bake until golden
4. Cool off for 30 minutes

Frosting recipe

1. Mix butter and cream cheese
2. Add sugar
3. Mix again

Interactive loop

```
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

How do we make an interactive loop?

Interactive loop

```haskell
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
  main
```

Locally defined programs

```
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  input <- getLine
  handle input
  where
    handle "EXIT" = putStrLn "Goodbye!"
    handle name = do
      putStrLn ("Hello, " ++ name ++ "!")
      main
```

Pattern matching with **`case`**-expression

```haskell
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  input <- getLine
  case input of
    "EXIT" -> putStrLn "Goodbye!"
    name -> do
      putStrLn ("Hello, " ++ name ++ "!")
      main
```

Pattern matching with **case**-expression

```
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  input <- getLine
  case input of
    "EXIT" -> putStrLn "Goodbye!"
    name -> do
      putStrLn ("Hello, " ++ name ++ "!")
      main
```

```
case <expr> of
  <pattern1> -> <expr1>
  ...
  <patternN> -> <exprN>
```

Managing state

```haskell
module Main where

main :: IO ()
main = do
  putStrLn "What is your name?"
  input <- getLine
  case input of
    "EXIT" -> putStrLn "Goodbye!"
    name -> do
      putStrLn ("Hello, " ++ name ++ "!")
      main
```

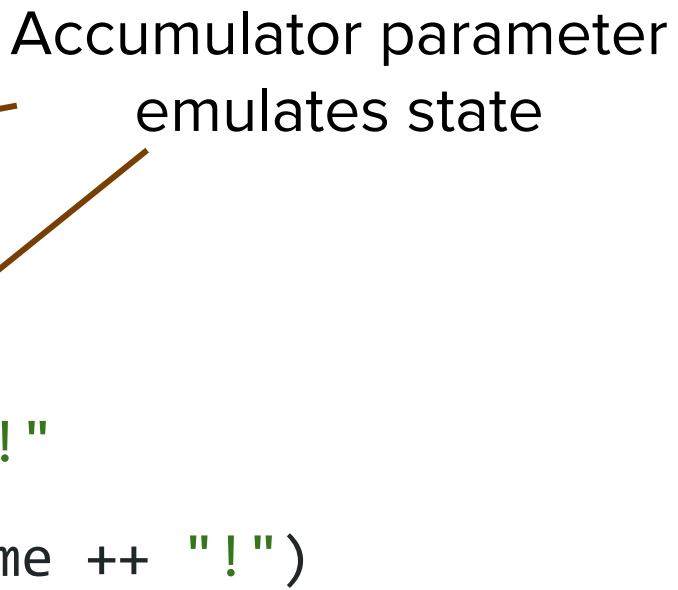
How do we add state to our loop?

Managing state

```
module Main where

runWith :: State -> IO ()
runWith state = do
  putStrLn "What is your name?"
  input <- getLine
  case input of
    "EXIT" -> putStrLn "Goodbye!"
    name -> do
      putStrLn ("Hello, " ++ name ++ "!")
      runWith state

main = runWith initialState
```

Managing state

```
module Main where

runWith :: State -> IO ()
runWith state = do
  putStrLn "What is your name?"
  input <- getLine
  case input of
    "EXIT" -> putStrLn "Goodbye!"
    name -> do
      putStrLn ("Hello, " ++ name ++ "!")
      runWith state

main = runWith initialState
```



Managing state

```haskell
module Main where

type Task = String
type State = [Task]

runWith :: State -> IO ()
runWith state = do
  putStrLn "What is your name?"
  input <- getLine
  case input of
    "EXIT" -> putStrLn "Goodbye!"
    name -> do
      putStrLn ("Hello, " ++ name ++ "!")
      runWith state

main = runWith initialState
```

Managing state

```
module Main where

type Task = String
type State = [Task]

runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case input of
    "EXIT" -> putStrLn "Goodbye!"
    newTask -> do
      putStrLn ("New task: " ++ newTask ++ "!")
      runWith (newTask : state)

main = runWith []
```

Lists and programs

```
type Task = String
type State = [Task]
```

```
printTasks :: [Task] -> IO ()
```

Lists and programs

```
type Task = String
type State = [Task]
```

```
printTasks :: [Task] -> IO ()
printTasks [] =

printTasks (task : tasks) =
```

Lists and programs

```
type Task = String
type State = [Task]
```

```
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."

printTasks (task : tasks) =
```

Lists and programs

```
type Task = String
type State = [Task]
```

```
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."

printTasks (task : tasks) = do
  putStrLn task
```

Lists and programs

```
type Task = String
type State = [Task]
```

```
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."

printTasks (task : tasks) = do
  putStrLn task
  printTasks tasks
```

Lists and programs

```haskell
type Task = String
type State = [Task]
```

```haskell
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."
printTasks [task] = putStrLn task
printTasks (task : tasks) = do
  putStrLn task
  printTasks tasks
```

Lists and programs

```
type Task = String
type State = [Task]
```

```
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."
printTasks tasks = map putStrLn tasks
```

Lists and programs

```
type Task = String
type State = [Task]
```

```
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."
printTasks tasks = map putStrLn tasks
```

```
Expected type: IO ()
  Actual type: [IO ()]
```

Lists and programs: sequence_

```haskell
type Task = String
type State = [Task]

printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."
printTasks tasks = sequence_ (map putStrLn tasks)

sequence_ :: [IO ()] -> IO ()
```

Lists and programs: sequence_

```
type Task = String
type State = [Task]
```

```
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."
printTasks tasks = sequence_ (map putStrLn tasks)

sequence_ :: [IO ()] -> IO ()
sequence_ [] =
sequence_ (program : programs) =
```

Lists and programs: sequence_

```haskell
type Task = String
type State = [Task]

printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."
printTasks tasks = sequence_ (map putStrLn tasks)

sequence_ :: [IO ()] -> IO ()
sequence_ [] =
sequence_ (program : programs) = do
  program
  sequence_ programs
```

Lists and programs: sequence_

```haskell
type Task = String
type State = [Task]

printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."
printTasks tasks = sequence_ (map putStrLn tasks)

sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (program : programs) = do
  program
  sequence_ programs
```

Lists and programs: sequence_

```haskell
type Task = String
type State = [Task]

printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."
printTasks tasks = sequence_ (map putStrLn tasks)

sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (program : programs) = do
  program
  sequence_ programs
```

```haskell
return :: a -> IO a
```

Lists and programs: mapM_

```haskell
type Task = String
type State = [Task]

printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."
printTasks tasks = sequence_ (map putStrLn tasks)

sequence_ :: [IO ()] -> IO ()

mapM_ :: (a -> IO b) -> [a] -> IO ()
```

Lists and programs: mapM_

```haskell
type Task = String
type State = [Task]
```

```haskell
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "No tasks."
printTasks tasks = mapM_ putStrLn tasks

sequence_ :: [IO ()] -> IO ()

mapM_ :: (a -> IO b) -> [a] -> IO ()
```

Parsing and handling commands

```haskell
module Main where

type Task = String
type State = [Task]

runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case input of
    "EXIT" -> putStrLn "Goodbye!"
    "PRINT" -> printTasks state
    newTask -> do
      putStrLn ("New task: " ++ newTask ++ "!")
      runWith (newTask : state)
```

Parsing and handling commands

```
data Command
  = PrintTasks
  | AddTask Task
```

```
type Task = String
type State = [Task]
```

Parsing and handling commands

```
data Command
  = PrintTasks
  | AddTask Task
```

```
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

Parsing and handling commands

```
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit

parseCommand :: String -> Command
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

Parsing and handling commands

```
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit

parseCommand :: String -> Command
```

```
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

Is this a good type for `parseCommand`?

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

```haskell
parseCommand :: String -> Maybe Command
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

```haskell
parseCommand :: String -> Maybe Command
parseCommand input =
  case words input of
```

```haskell
words :: String -> [String]
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

```haskell
parseCommand :: String -> Maybe Command
parseCommand input =
  case words input of
    ["PRINT"] -> Just PrintTasks
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

```haskell
parseCommand :: String -> Maybe Command
parseCommand input =
  case words input of
    ["PRINT"] -> Just PrintTasks
    ["DONE", idStr] ->
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

```haskell
parseCommand :: String -> Maybe Command
parseCommand input =
  case words input of
    ["PRINT"] -> Just PrintTasks
    ["DONE", idStr] ->
      case readMaybe idStr of
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

```haskell
parseCommand :: String -> Maybe Command
parseCommand input =
  case words input of
    ["PRINT"] -> Just PrintTasks
    ["DONE", idStr] ->
      case readMaybe idStr of
```

```haskell
readMaybe :: Read a => String -> Maybe a
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

```haskell
parseCommand :: String -> Maybe Command
parseCommand input =
  case words input of
    ["PRINT"] -> Just PrintTasks
    ["DONE", idStr] ->
      case readMaybe idStr of
```

```haskell
readMaybe :: String -> Maybe Int
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

```haskell
parseCommand :: String -> Maybe Command
parseCommand input =
  case words input of
    ["PRINT"] -> Just PrintTasks
    ["DONE", idStr] ->
      case readMaybe idStr of
        Nothing -> Nothing
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

```haskell
parseCommand :: String -> Maybe Command
parseCommand input =
  case words input of
    ["PRINT"] -> Just PrintTasks
    ["DONE", idStr] ->
      case readMaybe idStr of
        Nothing -> Nothing
        Just taskId -> Just (RemoveTask taskId)
```

Parsing and handling commands

```haskell
data Command
  = PrintTasks
  | AddTask Task
  | RemoveTask TaskId
  | Exit
```

```haskell
type Task = String
type TaskId = Int
type State = [(TaskId, Task)]
```

```haskell
parseCommand :: String -> Maybe Command
parseCommand input =
  case words input of
    ["PRINT"] -> Just PrintTasks
    ["DONE", idStr] ->
      case readMaybe idStr of
        Nothing -> Nothing
        Just taskId -> Just (RemoveTask taskId)
    _ -> Just (AddTask input)
```

Parsing and handling commands

```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
```

Parsing and handling commands

```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
```

Parsing and handling commands

```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just Exit -> putStrLn "Goodbye!"
```

Parsing and handling commands

```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just Exit -> putStrLn "Goodbye!"
    Just PrintTasks -> do
      printTasks state
      runWith state
```

Parsing and handling commands

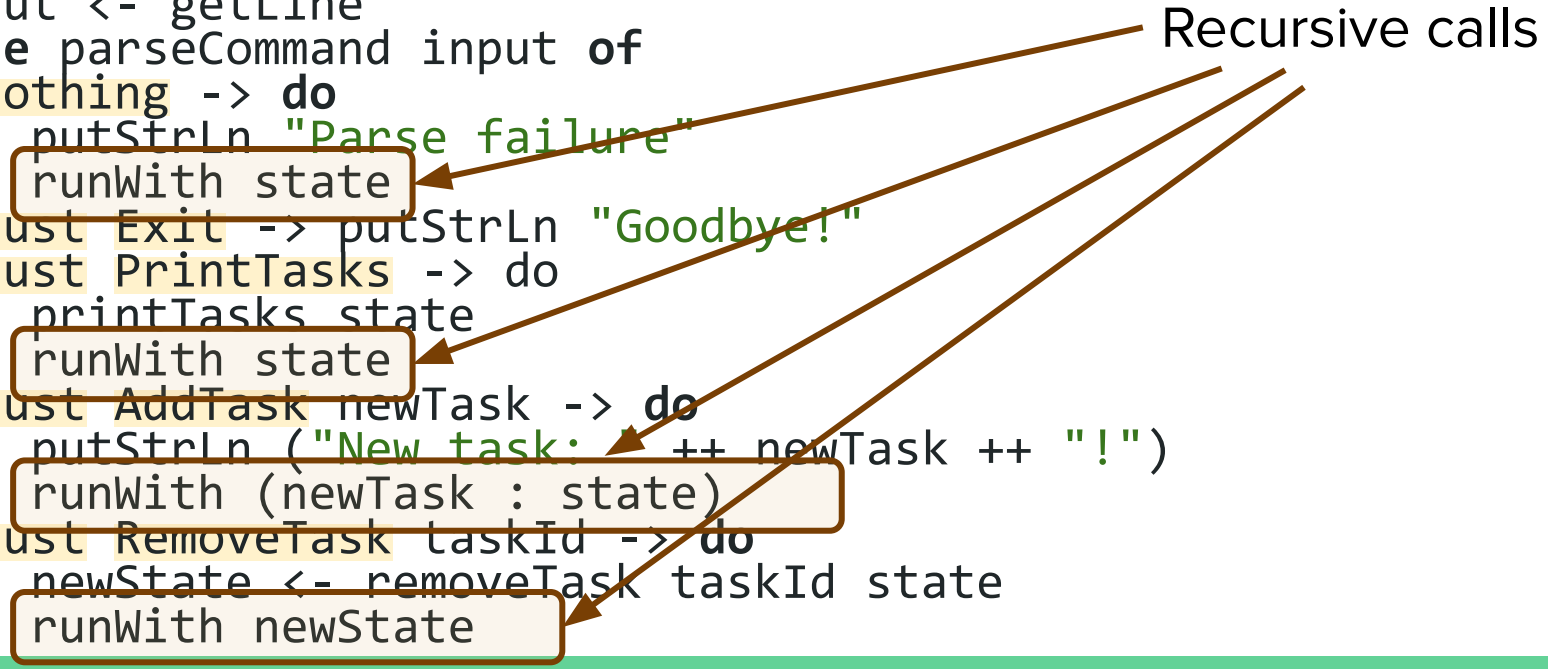
```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just Exit -> putStrLn "Goodbye!"
    Just PrintTasks -> do
      printTasks state
      runWith state
    Just AddTask newTask -> do
      putStrLn ("New task: " ++ newTask ++ "!")
      runWith (newTask : state)
```

Parsing and handling commands

```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just Exit -> putStrLn "Goodbye!"
    Just PrintTasks -> do
      printTasks state
      runWith state
    Just AddTask newTask -> do
      putStrLn ("New task: " ++ newTask ++ "!")
      runWith (newTask : state)
    Just RemoveTask taskId -> do
      newState <- removeTask taskId state
      runWith newState
```

Parsing and handling commands

```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just Exit -> putStrLn "Goodbye!"
    Just PrintTasks -> do
      printTasks state
      runWith state
    Just AddTask newTask -> do
      putStrLn ("New task: " ++ newTask ++ "!")
      runWith (newTask : state)
    Just RemoveTask taskId -> do
      newState <- removeTask taskId state
      runWith newState
```



Pure vs IO

```
type Handler = State -> (String, Maybe State)
```

Pure vs IO

```
type Handler = State -> (String, Maybe State)

handlePrintTasks :: Handler
handlePrintTasks state =
```

Pure vs IO

```
type Handler = State -> (String, Maybe State)

handlePrintTasks :: Handler
handlePrintTasks state = (prettyTasks state, Just state)
```

Pure vs IO

```
type Handler = State -> (String, Maybe State)

handlePrintTasks :: Handler
handlePrintTasks state = (prettyTasks state, Just state)
  where
    prettyTasks [] = "No tasks."
    prettyTasks tasks = unlines tasks
```

Pure vs IO

```
type Handler = State -> (String, Maybe State)

handlePrintTasks :: Handler
handlePrintTasks state = (prettyTasks state, Just state)
  where
    prettyTasks [] = "No tasks."
    prettyTasks tasks = unlines tasks

handleAddTask :: Task -> Handler
```

Pure vs IO

```
type Handler = State -> (String, Maybe State)

handlePrintTasks :: Handler
handlePrintTasks state = (prettyTasks state, Just state)
  where
    prettyTasks [] = "No tasks."
    prettyTasks tasks = unlines tasks

handleAddTask :: Task -> Handler
handleAddTask newTask state
  = (response, Just (addTask newTask state))
```

Pure vs IO

```
type Handler = State -> (String, Maybe State)

handlePrintTasks :: Handler
handlePrintTasks state = (prettyTasks state, Just state)
  where
    prettyTasks [] = "No tasks."
    prettyTasks tasks = unlines tasks

handleAddTask :: Task -> Handler
handleAddTask newTask state
  = (response, Just (addTask newTask state))
  where
    response = "New task: " ++ newTask ++ "!"
```

Pure vs IO

```
type Handler = State -> (String, Maybe State)

handlePrintTasks :: Handler
handlePrintTasks state = (prettyTasks state, Just state)
  where
    prettyTasks [] = "No tasks."
    prettyTasks tasks = unlines tasks

handleAddTask :: Task -> Handler
handleAddTask newTask state
  = (response, Just (addTask newTask state))
  where
    response = "New task: " ++ newTask ++ "!"

handleRemoveTask :: TaskId -> Handler
```

Pure vs IO

```
handleCommand :: Command -> Handler
```

Pure vs IO

```
handleCommand :: Command -> Handler
handleCommand command =
  case command of
    Exit                -> handleExit
    PrintTasks          -> handlePrintTasks
    AddTask newTask     -> handleAddTask newTask
    RemoveTask taskId -> handleRemoveTask taskId
```

Pure vs IO

```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
```

Pure vs IO

```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just command ->
```

Pure vs IO

```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just command ->
      case handleCommand command of
        (response, newState) ->
```

Pure vs IO

```
runWith :: State -> IO ()
runWith state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just command ->
      case handleCommand command of
        (response, newState) -> do
          putStrLn response
          runWith newState
```

Pure vs IO

```
runWith :: … -> … -> State -> IO ()
runWith parseCommand handleCommand state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just command ->
      case handleCommand command of
        (response, newState) -> do
          putStrLn response
          runWith newState
```

Pure vs IO

```
runWith
  :: (String -> Maybe Command)
  -> (Command -> Handler)
  -> State -> IO ()
runWith parseCommand handleCommand state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just command ->
      case handleCommand command of
        (response, newState) -> do
          putStrLn response
          runWith newState
```

Pure vs IO

```
runWith
  :: (String -> Maybe Command)
  -> (Command -> State -> (String, State))
  -> State -> IO ()
runWith parseCommand handleCommand state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just command ->
      case handleCommand command of
        (response, newState) -> do
          putStrLn response
          runWith newState
```

Pure vs IO

```
runWith
  :: (String -> Maybe command)
  -> (command -> state -> (String, state))
  -> State -> IO ()
runWith parseCommand handleCommand state = do
  putStrLn "Enter a command:"
  input <- getLine
  case parseCommand input of
    Nothing -> do
      putStrLn "Parse failure"
      runWith state
    Just command ->
      case handleCommand command of
        (response, newState) -> do
          putStrLn response
          runWith newState
```

What was the most unclear part of the lecture for you?

See Moodle