INTERACTION FUNCTIONS

- Definition of interaction function
- Function `main` is executed when we run a Haskell pro

```
ask           :: String -> IO String
ask question  = do
                  putStrLn question
                  getLine

main :: IO ()
main  = do
          name <- ask "May I ask your name?
          putStrLn ("Welcome " ++ name ++ "
```

- What is the difference between a function argumen program input?

INTERACTION FUNCTIONS

- Definition of interaction function
- Function `main` is executed when we run a Haskell pro

```
ask           :: String -> IO String
ask question  = do
                  putStrLn question
                  getLine

main :: IO ()
main  = do
          name <- ask "May I ask your name?
          putStrLn ("Welcome " ++ name ++ "
```

- What is the difference between a function argumen program input?

INTERACTION FUNCTIONS

- Definition of interaction function
- Function `main` is executed when we run a Haskell pro

```
ask           :: String -> IO String
ask question  = do
                  putStrLn question
                  getLine

main :: IO ()
main  = do
          name <- ask "May I ask your name?
          putStrLn ("Welcome " ++ name ++ "
```

- What is the difference between a function argumen program input?

SHOWING AND READING VALUES

Reading and writing values other than strings

- ➜ `getLine` and `putStr` read and write strings
- ➜ How about other types of values?
- ➜ `readLn` and `print` read and write other types of valu
- ➜ They can read/write a whole set of different types (c

An Example:

- ➜ Function `getNumber` prompting the user for a number

```
getNumber :: IO Int
```

- ➜ Main action: read two numbers, add them, and out result ✧
- ➜ Changing the type of `getNumber` changes the behav

SHOWING AND READING VALUES

Reading and writing values other than strings

- ➜ `getLine` and `putStr` read and write strings
- ➜ How about other types of values?
- ➜ `readLn` and `print` read and write other types of valu
- ➜ They can read/write a whole set of different types (c

An Example:

- ➜ Function `getNumber` prompting the user for a number

```
getNumber :: IO Int
```

- ➜ Main action: read two numbers, add them, and out result ✧
- ➜ Changing the type of `getNumber` changes the behav

SHOWING AND READING VALUES

Reading and writing values other than strings

- ➜ `getLine` and `putStr` read and write strings
- ➜ How about other types of values?
- ➜ `readLn` and `print` read and write other types of valu
- ➜ They can read/write a whole set of different types (c

An Example:

- ➜ Function `getNumber` prompting the user for a number

```
getNumber :: IO Int
```

- ➜ Main action: read two numbers, add them, and out result ✧
- ➜ Changing the type of `getNumber` changes the behav

SHOWING AND READING VALUES

Reading and writing values other than strings

- ➜ `getLine` and `putStr` read and write strings
- ➜ How about other types of values?
- ➜ `readLn` and `print` read and write other types of valu
- ➜ They can read/write a whole set of different types (c

An Example:

- ➜ Function `getNumber` prompting the user for a number

```
getNumber :: IO Int
```

- ➜ Main action: read two numbers, add them, and out result ✧
- ➜ Changing the type of `getNumber` changes the behav

SHOWING AND READING VALUES

Reading and writing values other than strings

- ➜ `getLine` and `putStr` read and write strings
- ➜ How about other types of values?
- ➜ `readLn` and `print` read and write other types of valu
- ➜ They can read/write a whole set of different types (c

An Example:

- ➜ Function `getNumber` prompting the user for a number

```
getNumber :: IO Int
```

- ➜ Main action: read two numbers, add them, and out result ✧
- ➜ Changing the type of `getNumber` changes the behav

Returning a Result

➜ So far, the last action in a interaction function alway
the interaction function's result, e.g.,

```
ask           :: String -> IO String
ask question  = do
                  putStrLn question
                  getLine                -- ::
```

and

```
getNumber :: IO Int
getNumber  = do
               putStrLn "Please input a numbe
               readLn                     -- ::
```

- How can we implement an interaction function reac numbers and returning a pair?
- What would be its type?
- Let us reuse `getNumber` to input each number
- How can we return the compound result? ✧

```
return ::  a -> IO a

getNumber :: IO Int
getNumber  = do
               putStrLn "Please input a num
               readLn
getTwoNumbers :: IO (Int, Int)
getTwoNumbers  = do
                   num1 <- getNumber
                   num2 <- getNumber
                   return (num1, num2)
```

- How can we implement an interaction function reac numbers and returning a pair?
- What would be its type?
- Let us reuse `getNumber` to input each number
- How can we return the compound result? ✧

```
return ::  a -> IO a

getNumber :: IO Int
getNumber  = do
               putStrLn "Please input a num
               readLn
getTwoNumbers :: IO (Int, Int)
getTwoNumbers  = do
                   num1 <- getNumber
                   num2 <- getNumber
                   return (num1, num2)
```

- How can we implement an interaction function reac numbers and returning a pair?
- What would be its type?
- Let us reuse `getNumber` to input each number
- How can we return the compound result? ✧

```
return ::  a -> IO a

getNumber :: IO Int
getNumber  = do
               putStrLn "Please input a num
               readLn
getTwoNumbers :: IO (Int, Int)
getTwoNumbers  = do
                   num1 <- getNumber
                   num2 <- getNumber
                   return (num1, num2)
```

- How can we implement an interaction function reac numbers and returning a pair?
- What would be its type?
- Let us reuse `getNumber` to input each number
- How can we return the compound result? ✧

```
return ::  a -> IO a

getNumber :: IO Int
getNumber  = do
               putStrLn "Please input a num
               readLn
getTwoNumbers :: IO (Int, Int)
getTwoNumbers  = do
                   num1 <- getNumber
                   num2 <- getNumber
                   return (num1, num2)
```

- How can we implement an interaction function reac numbers and returning a pair?
- What would be its type?
- Let us reuse `getNumber` to input each number
- How can we return the compound result? ✧

```
return ::  a -> IO a

getNumber :: IO Int
getNumber  = do
               putStrLn "Please input a num
               readLn
getTwoNumbers :: IO (Int, Int)
getTwoNumbers  = do
                   num1 <- getNumber
                   num2 <- getNumber
                   return (num1, num2)
```

- How can we implement an interaction function reac numbers and returning a pair?
- What would be its type?
- Let us reuse `getNumber` to input each number
- How can we return the compound result? ✧

```
return ::  a -> IO a

getNumber :: IO Int
getNumber  = do
               putStrLn "Please input a num
               readLn
getTwoNumbers :: IO (Int, Int)
getTwoNumbers  = do
                   num1 <- getNumber
                   num2 <- getNumber
                   return (num1, num2)
```

Conditionals and Recursion in I/O Ac

- ➜ We can use recursion as usual in interaction function
- ➜ Let's implement a function outputting a list of strings, line by its own
- ➜ What is the type?

```
putStringList :: [String] -> IO ()
```

- ➜ How about the implementation? ✧

Conditionals and Recursion in I/O Ac

- ➜ We can use recursion as usual in interaction function
- ➜ Let's implement a function outputting a list of strings, line by its own
- ➜ What is the type?

```
putStringList :: [String] -> IO ()
```

- ➜ How about the implementation? ✧

Interaction Versus Computation:

- ➜ It is not always clear what to realise as interaction ar computation
- ➜ A character `\n` (newline) in a string forces output to t
- ➜ `"one\ntwo"`
- ➜ How can we use it to pull the recursion of `putStringL` computational part?
- ➜ Concatenate the strings attaching a `\n` ✧

Interaction Versus Computation:

- ➜ It is not always clear what to realise as interaction ar computation
- ➜ A character `\n` (newline) in a string forces output to t
- ➜ `"one\ntwo"`
- ➜ How can we use it to pull the recursion of `putStringL` computational part?
- ➜ Concatenate the strings attaching a `\n` ✧

Interaction Versus Computation:

- ➜ It is not always clear what to realise as interaction ar computation
- ➜ A character `\n` (newline) in a string forces output to t
- ➜ `"one\ntwo"`
- ➜ How can we use it to pull the recursion of `putStringL` computational part?
- ➜ Concatenate the strings attaching a `\n` ✧

Interaction Versus Computation:

- ➜ It is not always clear what to realise as interaction ar computation
- ➜ A character `\n` (newline) in a string forces output to t
- ➜ `"one\ntwo"`
- ➜ How can we use it to pull the recursion of `putStringL` computational part?
- ➜ Concatenate the strings attaching a `\n` ✧

Recursive Interaction Loops:

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops:

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops:

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops:

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops:

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops:

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Recursive Interaction Loops

Pure function to convert string to upper case:

```
allUpper     :: String -> String
allUpper str  = map toUpper str
```

Interaction:

- ➜ Interaction steps:
  1. Read a line
  2. If it is empty, stop
  3. Otherwise, convert it to uppercase and repeat
- ➜ What's the type of this interaction?

```
upperLine :: IO ()
```

✧

Notes on the syntax:

- The `then` and `else` branch of the conditional express
  indented
- We have to use a second `do` expression for the `else`

```
upperLine :: IO ()
upperLine  = do
               line <- getLine
               if line == ""
                 then                  -- base ca
                   return ()
                 else do               -- steppin
                   putStrLn (allUpper line)
                   upperLine
```

Adding up a list of numbers:

① Read a list of numbers from the keyboard:
   ➜ Repeatedly read a number from the console
   ➜ Stop the process, when a `0` is entered
   ➜ Construct a list of these numbers
② Main routine that after reading the list, calculates the
   prints it ✧

Adding up a list of numbers:

① Read a list of numbers from the keyboard:
   ➜ Repeatedly read a number from the console
   ➜ Stop the process, when a `0` is entered
   ➜ Construct a list of these numbers
② Main routine that after reading the list, calculates the
   prints it ✧

File I/O

- Reading and writing to the console is not enough
- Programs must be able to read and write files

```
type FilePath = String

writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath           -> IO Stri
```

- `FilePath` contains Unix-style file names, e.g.,

  `/home/chak/.emacs`

- As an example, consider a program that
  - Reads a file name form the console
  - Then, reads the file contents and prints it to the c

File I/O

- Reading and writing to the console is not enough
- Programs must be able to read and write files

```
type FilePath = String

writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath           -> IO Stri
```

- `FilePath` contains Unix-style file names, e.g.,

  `/home/chak/.emacs`

- As an example, consider a program that
  - Reads a file name form the console
  - Then, reads the file contents and prints it to the c

File I/O

- Reading and writing to the console is not enough
- Programs must be able to read and write files

```
type FilePath = String

writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath           -> IO Stri
```

- `FilePath` contains Unix-style file names, e.g.,

  `/home/chak/.emacs`

- As an example, consider a program that
  - Reads a file name form the console
  - Then, reads the file contents and prints it to the c

Showing and Reading Values Revisit

- Previously, we discussed `readLn` and `print`
- They are in fact not primitive, but constructed from s functions

Converting values from and to strings:

- These simpler functions are `read` and `show`
- They are pure functions converting from and to string

```
show 42          ⇒  "42"
read "42"        ⇒  42
```

- It is important to distinguish values (e.g., `42`) and thei denotation (e.g., `"42"`)
- Remember that `"42"` is just a shorthand for `['4', '2'`

Showing and Reading Values Revisit

- Previously, we discussed `readLn` and `print`
- They are in fact not primitive, but constructed from s functions

Converting values from and to strings:

- These simpler functions are `read` and `show`
- They are pure functions converting from and to string

```
show 42          ⇒  "42"
read "42"        ⇒  42
```

- It is important to distinguish values (e.g., `42`) and thei denotation (e.g., `"42"`)
- Remember that `"42"` is just a shorthand for `['4', '2'`

Showing and Reading Values Revisit

- Previously, we discussed `readLn` and `print`
- They are in fact not primitive, but constructed from s functions

Converting values from and to strings:

- These simpler functions are `read` and `show`
- They are pure functions converting from and to string

```
show 42          ⇒  "42"
read "42"        ⇒  42
```

- It is important to distinguish values (e.g., `42`) and thei denotation (e.g., `"42"`)
- Remember that `"42"` is just a shorthand for `['4', '2'`

Overloading:

- `show` and `read` are overloaded (similar to + and ==):

```
show :: Show a => a -> String
read :: Read a => String -> a
```

- `Show` and `Read` are type classes
- They include all types, we discussed so far, except fu

```
show 42                          ⇒  "42"
show 3.141                       ⇒  "3.141"
show True                        ⇒  "True"
show [1..3]                      ⇒  "[1,2,3
show [(1, True), (5, False)]     ⇒  "[(1,Tr
```

Definition of `print` and `readLn`:

```
print        :: Show a => a -> IO ()
print value  = putStrLn (show value)

readLn :: Read a => IO a
readLn  = do
            string <- getLine
            return (read string)
```

Using `show` and `read` With File Operations:

- ➜ `readFile`, `writeFile`, and `appendFile` only handle stri
- ➜ In combination with `show` and `read`, we can handle a range of values
- ➜ For example, consider
  - reading a number from the console and
  - writing a list from 1 up to that number to a file ✧

Definition of `print` and `readLn`:

```
print        :: Show a => a -> IO ()
print value  = putStrLn (show value)

readLn :: Read a => IO a
readLn  = do
            string <- getLine
            return (read string)
```

Using `show` and `read` With File Operations:

- ➜ `readFile`, `writeFile`, and `appendFile` only handle stri
- ➜ In combination with `show` and `read`, we can handle a range of values
- ➜ For example, consider
  - reading a number from the console and
  - writing a list from 1 up to that number to a file ✧

A Larger Example

- Let us extend the supermarket example
- We reuse the types

```
type Cents        = Int
type PriceList    = [(String, Cents)]
type ShoppingList = [(String, Int)]
```

  and function

```
cost :: PriceList -> ShoppingList -> Cents
```

- A POS (Point Of Sale) system should store the price lis
  permanent storage
- Sequence of interactions:
  1. Read price list from a file `pricelist`
  2. Read a list of items and quantities from the conso
  3. Calculate the total cost and print it

A Larger Example

- Let us extend the supermarket example
- We reuse the types

```
type Cents        = Int
type PriceList    = [(String, Cents)]
type ShoppingList = [(String, Int)]
```

  and function

```
cost :: PriceList -> ShoppingList -> Cents
```

- A POS (Point Of Sale) system should store the price lis
  permanent storage
- Sequence of interactions:
  1. Read price list from a file `pricelist`
  2. Read a list of items and quantities from the conso
  3. Calculate the total cost and print it

A Larger Example

- Let us extend the supermarket example
- We reuse the types

```
type Cents        = Int
type PriceList    = [(String, Cents)]
type ShoppingList = [(String, Int)]
```

  and function

```
cost :: PriceList -> ShoppingList -> Cents
```

- A POS (Point Of Sale) system should store the price lis
  permanent storage
- Sequence of interactions:
  1. Read price list from a file `pricelist`
  2. Read a list of items and quantities from the conso
  3. Calculate the total cost and print it