CS331
Haskell Tutorial 04

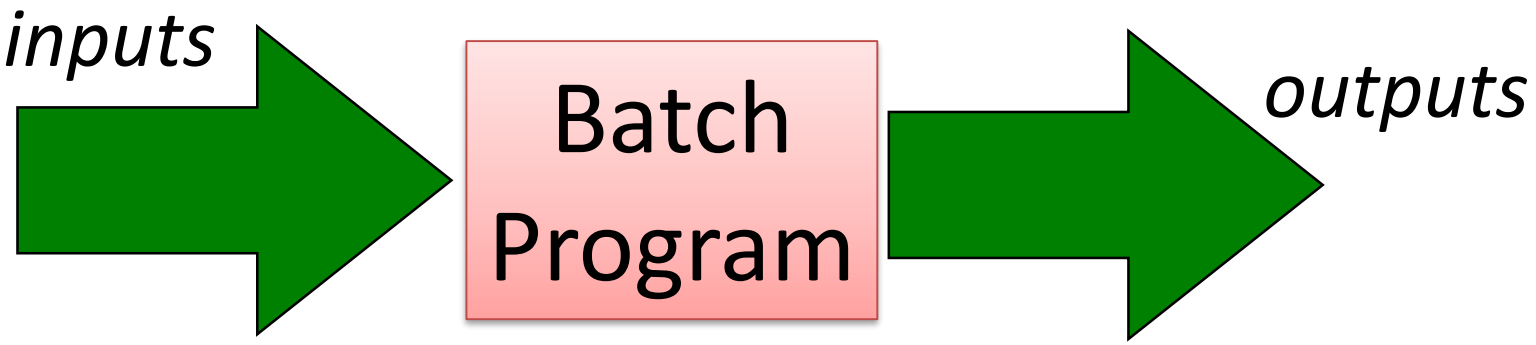
A. Sahu
Dept of Comp. Sc. & Engg.
Indian Institute of Technology Guwahati

Outline

- Interactive Program in Haskell Programming
- Defining Data Types
- May be or Just
- Binary Tree Example
- Advanced type

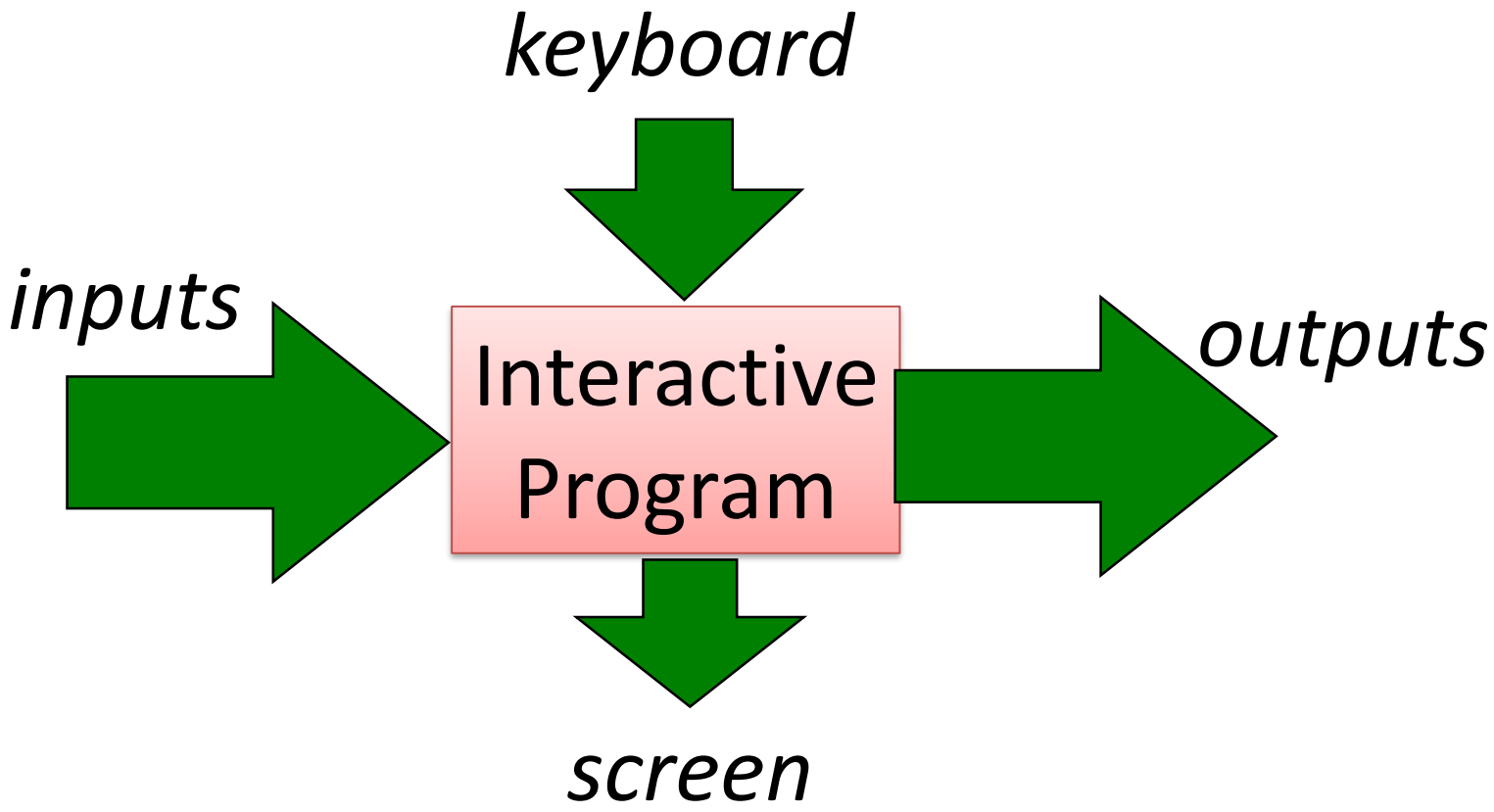
Haskell Batch Program

To date, we have seen how Haskell can be used to write batch programs that take all their inputs at the start and give all their outputs at the end.



Haskell Interactive Program

However, we would also like to use Haskell to write interactive programs that read from the keyboard and write to the screen, as they are running.



The Problem

Haskell programs are pure mathematical functions:

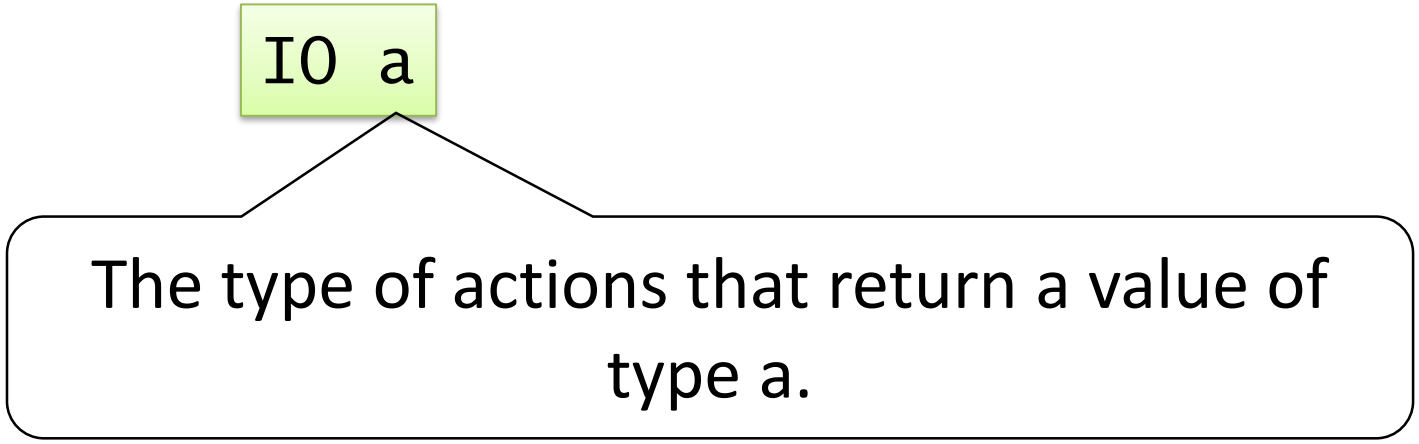
Haskell programs have no side effects.

However, reading from the keyboard and writing to the screen are side effects:

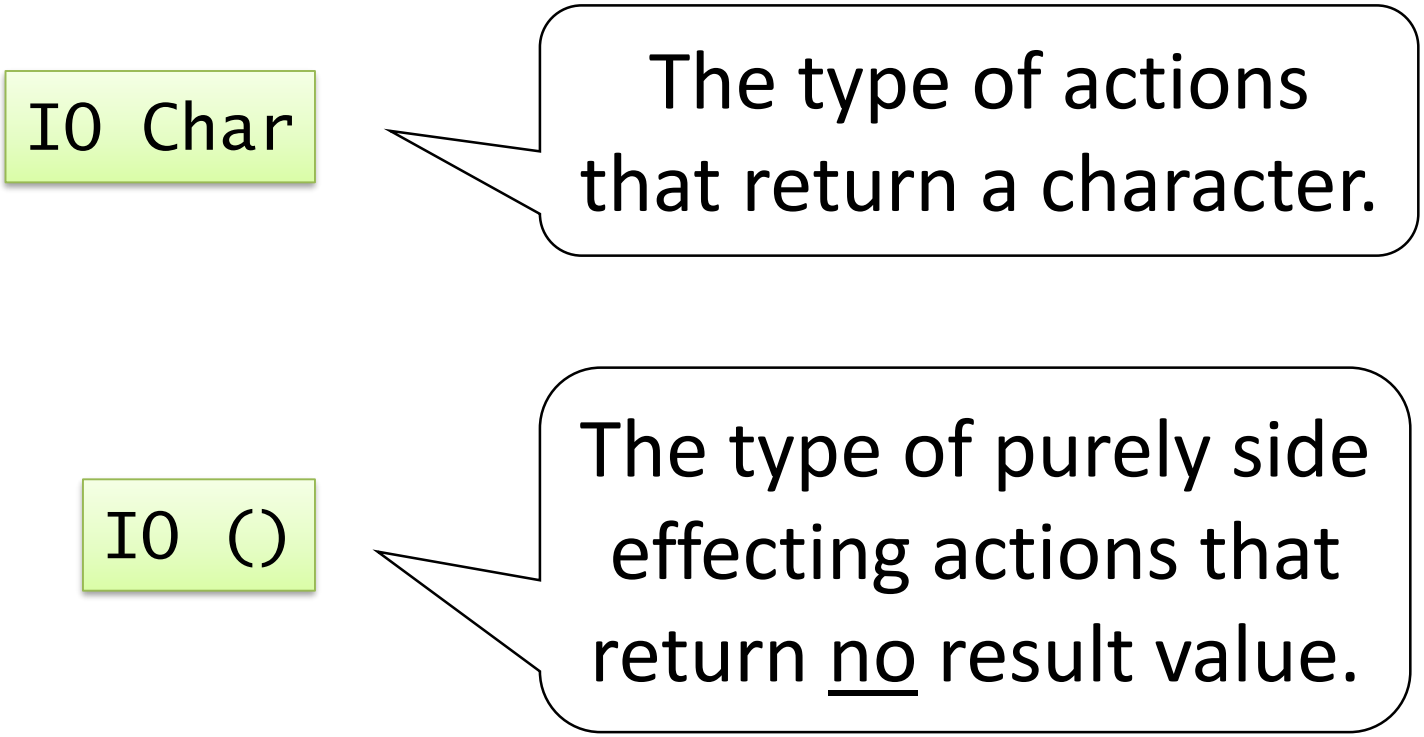
Interactive programs have side effects.

The Solution

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.



For example : No side effect IO functions



() is the type of tuples with no components.

Primitive Actions

The standard library provides a number of actions, including the following three primitives:

- The action getChar

```
getChar :: IO Char
```

  - reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

- The action putChar c

```
putChar :: Char → IO ()
```

  - writes the character c to the screen, and returns no result value:

- The action return v

```
return :: a → IO a
```

  - simply returns the value v, without performing any interaction:

Sequencing Actions

A sequence of actions can be combined as a single composite action using the keyword do.
For example:

```
getTwo :: IO (Char,Char)
getTwo  = do x ← getChar
             y ← getChar
             return (x,y)
```

Sequencing Actions

- Each action must begin in precisely the same column. That is, the layout rule applies;
- The values returned by intermediate actions are discarded by default, but if required can be named using the $\leftarrow$ operator;
- The value returned by the last action is the value returned by the sequence as a whole.

Other Library Actions

Reading a string from the keyboard:

```
getLine :: IO String
getLine  = do x ← getChar
              if x == '\n' then
                 return []
               else
                 do xs ← getLine
                    return (x:xs)
```

Other Library Actions

Writing a string to the screen:

```
putStr        :: String → IO ()
putStr []      = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

Writing a string and moving to a new line:

```
putStrLn    :: String → IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Example

We can now define an action that prompts for a string to be entered and displays its length:

```
strlen :: IO ()
strlen  = do putStr "Enter a string: "
             xs ← getLine
             putStr "The string has "
             putStr (show (length xs))
             putStrLn " characters"
```

Example

```
> strlen

Enter a string: hello there
The string has 11 characters
```

Evaluating an action executes its side effects, with the final result value being discarded.

Defining Types: Data Declarations

A new type can be defined by specifying its set of values using a data declaration.

```
data Bool = False | True
```

Bool is a new type, with two new values False and True.

Defining Types: Data Declarations

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers      :: [Answer]
answers       = [Yes,No,Unknown]

flip         :: Answer → Answer
flip Yes      = No
flip No       = Yes
flip Unknown  = Unknown
```

Defining Types: Data Declarations

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square          :: Float → Shape
square n         = Rect n n

area            :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Defining Types: Data Declarations

```
prelude>data Shape = Circle Float Float Float | Rectangle
                                    Float Float Float Float
Prelude> surface (Circle _ _ r) = pi * r ^ 2
Prelude> surface (Circle 10 20 10)
314.15927
Prelude>surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) *
(abs $ y2 - y1)
Prelude>
Prelude> surface (Rectangle 0 0 100 100)
10000.0
Prelude>surface (Circle 10 20 10)
Err....
```

Defining Types: Data Declarations

- if we add deriving (Show) at the end of a *data* declaration
- Haskell automagically makes that type part of the Show typeclass

```
Prelude>data Shape = Circle Float Float Float | Rectangle
Float Float Float Float deriving (Show)
Prelude>Circle 10 20 5
Circle 10.0 20.0 5.0
Prelude> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
Prelude>  map (Circle 10 20) [4,5,6,6]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0,Circle 10.0 20.0 6
.0,Circle 10.0 20.0 6.0]
```

If we want a list of concentric circles with different radii, we can do this.

Defining Types: Records

```
Prelude>data Person = Person String String Int Float String String de
riving (Show)
Prelude>let guy = Person "Buddy" "Finklestein" 43 184.2 "526-
2928" "Chocolate"
Prelude> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
Prelude>firstName (Person firstname _ _ _ _ _) = firstname
Prelude>lastName (Person _ lastname _ _ _ _) = lastname
Prelude>age (Person _ _ age _ _ _) = age
Prelude>height (Person _ _ _ height _ _) = height
Prelude>phoneNumber (Person _ _ _ _ number _) = number
Prelude> flavor (Person _ _ _ _ _ flavor) = flavor
```

Defining Types: Records

```
Prelude> data Person =
Person { firstName :: String,  lastName :: String,  age :: Int ,  height :: Flo
at ,  phoneNumber :: String , flavor :: String } deriving (Show)
Prelude>    :t flavor
flavor :: Person -> String
Prelude>data Car = Car String String Int deriving (Show)
Prelude>  Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
Prelude> data Car a b c = Car { company :: a, model :: b
                  , year :: c    } deriving (Show)
Prelude>tellCar (Car {company = c, model = m, year = y}) = "This " ++ c
++ " " ++ m ++ " was made in " ++ show y
Prelude> let stang = Car {company="Ford", model="Mustang", year=1967}
Prelude>tellCar stang
"This Ford Mustang was made in 1967"
```

Data Type vector Example

```
Prelude>  data Vector a = Vector a a a deriving (Show)
Prelude>
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)
Prelude> (Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)
Prelude>(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

```
Prelude> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
Prelude>Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
12 9 19
Prelude>Vector 3 9 7 `vectMult` 10
Vector 30 90 70
Prelude> Vector 4 9 5 `scalarMult` Vector 9.0 2.0 4.0
74.0
Prelude> Vector 2 9 3 `vectMult` (Vector 4 9 5 `scalarMult` Vector 9 2 4)
Vector 148 666 222
```

Data Type : True False Ordering

Prelude> **data Bool = False | True deriving (Ord)**

- Because the False value constructor is specified first and the True value constructor is specified after it, we can consider True as greater than False.

Prelude> True `compare` False

GT

Prelude> True > False

True

Data Type : May be and Just

Prelude> **data Maybe a = Just a | Nothing z**

That declaration defines a type, Maybe a, which is parameterized by a type variable a, which just means that you can use it with any type in place of a.

```
lend amount balance =
          let reserve = 100
           newBalance = balance - amount
          in if balance < reserve then Nothing
          else Just newBalance
```

Defining Types: Data Declarations

Similarly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

we can define:

```
return          :: a → Maybe a
return x         = Just x

(>>=) :: Maybe a → (a → Maybe b) → Maybe b
Nothing >>= _ = Nothing
Just x  >>= f = f x
```

Data Type : May be and Just

The Nothing value constructor is specified before the Just value constructor

Prelude> Nothing < Just 100

True

Prelude> Nothing > Just (-49999)

False

Prelude> Just 3 `compare` Just 2

GT

Prelude> Just 100 > Just 50

True

BST Creation in Haskell

```
data Tree a = Nil | Node (Tree a) a (Tree a) deriving Show
-- Checking Empty function
empty Nil = True
empty  _  = False
-- Insert an  element to  the Tree
insert Nil x = Node Nil x Nil
insert (Node t1 v t2) x
        | v == x = Node t1 v t2
        | v  < x = Node t1 v (insert t2 x)
        | v  > x = Node (insert t1 x) v t2
```

BST Creation in Haskell

```
--Contain : if element is present return true
contains Nil _ = False
contains (Node t1 v t2) x
        | x == v = True
        | x  < v = contains t1 x
        | x  > v = contains t2 x
-- Creation of Tree from list of number
ctree [] = Nil
ctree (h:t) = ctree2 (Node Nil h Nil) t
      where
            ctree2 tr [] = tr
            ctree2 tr (h:t) = ctree2 (insert tr h) t
-- Creation of Tree from list of number Example
ctree [1,2,4,5,8,7]
```

Advanced Data Types in Haskell (Program not tested in GHCI)

Recursive Types

In Haskell, new types can be defined in terms of themselves. That is, types can be recursive.

Define Natural Number

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors

Zero :: Nat and Succ :: Nat $\longrightarrow$ Nat.

Recursive Types

- A value of type Nat is
  - either Zero, or of the form Succ n
  - where n :: Nat.
- That is, Nat contains the following infinite sequence of values:

```
Zero
```

```
Succ Zero
```

```
Succ (Succ Zero)
```

⋮

Recursive Types

- We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the successor function (1 +).
- For example, the value

```
Succ (Succ (Succ Zero))
```

represents the natural number

```
1 + (1 + (1 + 0))  =  3
```

Recursive Types

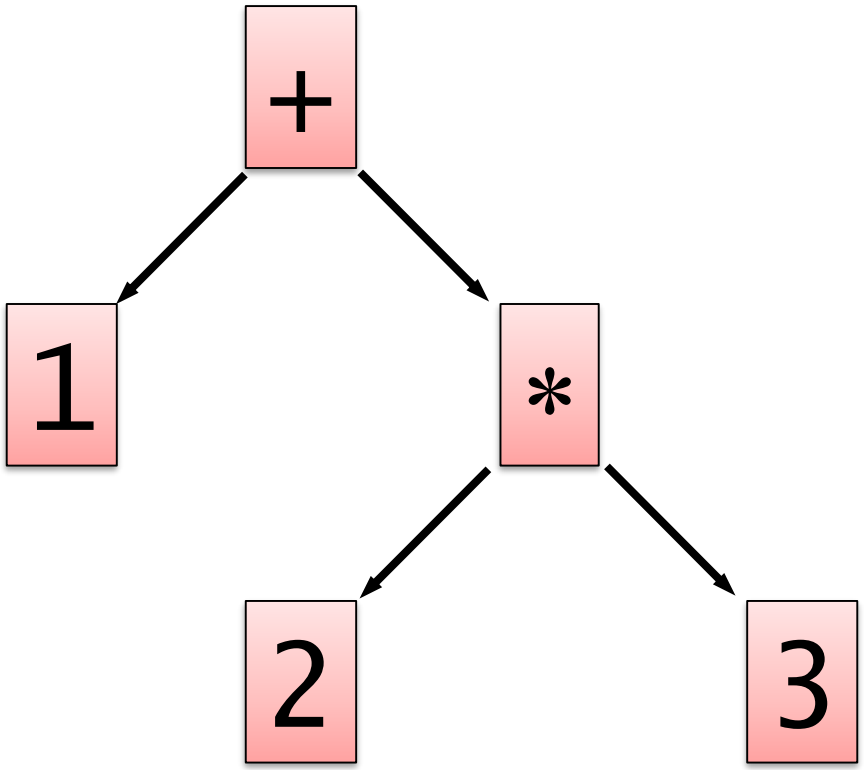
Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int           :: Nat → Int
nat2int Zero       = 0
nat2int (Succ n) = 1 + nat2int n


int2nat           :: Int → Nat
int2nat 0          = Zero
int2nat n     = Succ (int2nat (n-1))
```

Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.



Arithmetic Expressions

Using recursion, a suitable new type to represent such expressions can be defined by:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Arithmetic Expressions

Using recursion, it is now easy to define functions that process expressions. For example:

```
size            :: Expr → Int
size (Val n)     = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval            :: Expr → Int
eval (Val n)     = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Thanks