CSCE 314
Programming Languages

Interactive Programming: I/O

Dr. Hyunyoung Lee
Introduction

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 (e.g., a compiler).
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 (e.g., an interpreter).
The Problem: Haskell functions are pure mathematical functions

Haskell programs **have no side effects.**

**referential transparency:** called with the same arguments, a function always returns the same value

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

Interactive programs **have side effects.**
The Solution – The IO Type

Interactive programs can be viewed as a pure function whose domain and codomain are the current state of the world:

\[
\text{type IO} = \text{World} \rightarrow \text{World}
\]

However, an interactive program may return a result value in addition to performing side effects:

\[
\text{type IO } a = \text{World} \rightarrow (a, \text{World})
\]

What if we need an interactive program that takes an argument of type \( b \)?

\[
b \rightarrow \text{IO } a \\
\text{b} \rightarrow \text{World} \rightarrow (a, \text{World})
\]
The Solution (Cont.)

Now, interactive programs (impure actions) can be defined using the IO type:

\[ \text{IO } a \]

The type of actions that return a value of type \( a \)

For example:

\[ \text{IO } \text{Char} \]

The type of actions that return a character

\[ \text{IO } () \]

The type of actions that return the empty tuple (a dummy value); purely side-effecting actions
Basic Actions (built into the GHC system)

1. The action `getChar` reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

   ```haskell
getChar :: IO Char
   ```

2. The action `putChar c` writes the character `c` to the screen, and returns no result value:

   ```haskell
putChar :: Char -> IO ()
   ```

3. The action `return v` simply returns the value `v`, without performing any interaction with the user:

   ```haskell
return :: a -> IO a
   ```
Sequencing – do notation

A sequence of IO actions can be combined into a single composite action using the do notation:

```
  do v1 <- a1
     v2 <- a2
     a3
     ...
     vn <- an
   return (f v1 v2 ... vn)
```

First perform action $a_1$ and call its result value $v_1$, ..., and finally, apply the function $f$ to combine all the results into a single value, and return it as the result value from the expression as a whole.

The layout rule applies.

If the value $v_i$ is not used, simply write $a_i$.

Called “generator” because $a_i$ generates value for $v_i$.
Sequencing Example

Define an action (\texttt{act1}) that reads three characters, discards the second, and returns the first and third as a pair.

\begin{verbatim}
act1 :: IO (Char,Char)
act1 = do x <- getChar
         getChar
         y <- getChar
         return (x,y)
\end{verbatim}

The character read by the second \texttt{getChar} is not used.
Derived Primitives

Reading a string from the keyboard:

```haskell
geline :: IO String
geline = do x <- getChar
            if x == '\n' then return []
            else do xs <- getline
                    return (x:xs)
```

Writing a string to the screen:

```haskell
putstr :: String -> IO ()
putstr [] = return ()
putstr (x:xs) = do putchar x
                 putstr xs
```

Writing a string and moving to a new line:

```haskell
putstrln :: String -> IO ()
putstrln xs = do putstr xs
               putchar '\n'
```
Building More Complex IO Actions

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

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

Now, try:

```haskell
> strlen
Enter a string: Haskell Rocks!
The string has 14 characters.
```
The Type of main

A complete Haskell program is a single IO action. For example:

```haskell
main :: IO ()
main = getLine >>= \cs ->
    putLine (reverse cs)
```

Typically, IO “contaminates” a small part of the program (outermost part), and a larger portion of a Haskell program does not perform any IO. For example, in the above definition of main, reverse is a non-IO function.
Hangman

Consider the following version of **hangman**:

1. One player secretly types in a word.

2. The other player tries to deduce the word, by entering a sequence of guesses.

3. For each guess, the computer indicates which letters in the secret word occur in the guess.

4. The game ends when the guess is correct.
Hangman (Cont.)

We adopt a top down approach to implementing hangman in Haskell, starting as follows:

```haskell
hangman :: IO ()
hangman =
  do putStrLn "Think of a word: "
     word <- sgetLine
     putStrLn "Try to guess it:"
     guess word
```
Hangman (Cont.)

The action \texttt{sgetLine} reads a line of text from the keyboard, echoing each character as a dash:

\begin{verbatim}
 sgetLine :: IO String
 sgetLine = do x <- getCh
               if x == '\n' then
                   do putChar x
                   return []
               else
                   do putChar '-'
                   xs <- sgetLine
                   return (x:xs)
\end{verbatim}
Hangman (Cont.)

The action `getCh` reads a single character from the keyboard, without echoing it to the screen:

```haskell
import System.IO

getCh :: IO Char
getCh = do hSetEcho stdin False -- echo off
           c <- getChar
           hSetEcho stdin True   -- echo on
           return c
```
The function `guess` is the main loop, which requests and processes guesses until the game ends.

```haskell
guess :: String -> IO ()
guess word =
    do putStr "> 
        xs <- getLine
        if xs == word then
            putStrLn "You got it!"
        else
            do putStrLn (diff word xs)
               guess word
```
Hangman (Cont.)

The function \textit{diff} indicates which characters in one string occur in the second string:

\begin{align*}
\text{diff} & : \text{String} \to \text{String} \to \text{String} \\
\text{diff} \ \text{xs} \ \text{ys} & = \\
& \quad \left[\text{if elem } x \ \text{ys} \ \text{then } x \ \text{else } '-' \ | \ x \ < - \ \text{xs}\right]
\end{align*}

For example:

\begin{verbatim}
> \text{diff} "haskell" "pascal"
"-as--ll"
\end{verbatim}