I/O in Haskell

Generally, I/O functions in Haskell have type \texttt{IO a}, where \texttt{a} could be any type. The purpose and use of \texttt{a} will be explained later. We call these “commands” or “actions”, for we think of them as performing the side effect of I/O, unlike ordinary, pure functions.

- To output a character:
  
  \texttt{putChar :: Char \rightarrow IO ()}

  e.g., \texttt{putChar 'c'}

- To output a string:
  
  \texttt{putStr :: String \rightarrow IO ()}

  e.g., \texttt{putStr "Hello"}
Chaining I/O Actions

How do we perform several I/O actions in sequence? Use the do construct.

- To perform the previous two commands in sequence:

  ```hs
  do putChar 'c'
  putStrLn "Hello"
  ```

  The overall type is `IO ()`, taken from the last command.

- To perform those two commands, then loop back:

  ```hs
  myloop :: IO ()
  myloop = do putChar 'c'
             putStrLn "Hello"
             myloop
  ```
Commands That Return Data

What is the function that reads a character, and how do we use it?
This is where the a in IO a comes in. The character input function is:

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

It says, “getChar is a command that returns a character”.
(In retrospect, IO () returns “void”.)

\[
\text{do } c \leftarrow \text{getChar} --\text{this is how we obtain the character} \\
\quad --c \text{ is now a Char and you can use it, e.g.,} \\
\quad \text{putChar } c \\
\quad --\text{more reading and writing} \\
\quad \text{d } \leftarrow \text{getChar} \\
\quad \text{putChar } d
\]
Commands That Return Data

The `getLine` command reads a whole line and returns it as a string:

```haskell
getLine :: IO String
```

The `return` command does nothing but just returns data:

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

Example: a command that reads a line and returns the length:

```haskell
getLineLength :: IO Int
getLineLength = do s <- getLine
                  return (length s)

-- example use
do len <- getLineLength
   print len
```
Detecting End of File

The isEOF command returns True iff there is nothing more to read:

\[
\text{isEOF} :: \text{IO} \ \text{Bool}
\]

Note: this is like Pascal, not C. Also, Hugs does not implement it; instead, it implements hugsIsEOF, which is like C.

A command that reads characters and prints them until the end.

\[
dump = \text{do } b \leftarrow \text{isEOF} \\
\quad \text{if } b \text{ then return } () \\
\quad \text{else do } c \leftarrow \text{getChar} \\
\quad \quad \text{putChar } c \\
\quad \text{dump}
\]
Exceptions

What if you read past the end? An exception will be thrown. I/O Exceptions are of type IOError. You can catch them with:

\[
\text{catch} :: \text{IO } a \rightarrow (\text{IOError } \rightarrow \text{IO } a) \rightarrow \text{IO } a
\]

The first argument is the normal command to perform, and the second is the exception handler. If the normal command throws an exception, the exception is passed to the exception handler. In the handler, you can determine if the exception is caused by eof-of-file:

\[
\text{isEOFError} :: \text{IOError } \rightarrow \text{Bool} \quad \text{--in module IO}
\]

You can also re-throw the exception with:

\[
\text{ioError} :: \text{IOError } \rightarrow \text{IO } a
\]
Exception Example

The `getLineLength` command rewritten to return 0 on all exceptions:

```haskell
getLineLength = do s <- getLine
    return (length s)
    `catch`
    _ -> return 0
```

The `dump` command rewritten using `catch` (rethrows non-eof exceptions):

```haskell
dump = do c <- getChar
    putStrLn
    dump
    `catch`
    e -> if isEOFError e then return ()
         else ioError e
```
More Exception-Handling Commands

The IO.try command performs your command and catches all exceptions:

```haskell
try :: IO a -> IO (Either IOError a)
try f = catch (do r <- f
                   return (Right r))
                   (\e -> return (Left e))
```

The IO.bracket command is similar to Java try-finally:

```haskell
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket alloc fin m = do x <- alloc
                          rs <- try (m x)
                          fin x
                          case rs of Right r -> return r
                                    Left e -> ioError e
```
The Magic of IO a Explained

The IO a type is really a state transformer—a function that maps old state to new state. Generally, state transformers belong to the Monad class:

```haskell
class Monad m where return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    c >> d = c >>= \_ -> d
```

- **return** simply returns data and keeps the state unchanged
- **(>>=)** runs the first command and passes its return value and resulting state to the second command—a glorified function composition
- **(>>)*** is like (>>=) but discards the return value of the first command.
The Magic of do Explained

The do construct is just syntactic sugar. It is translated to monad operators:

- \( \text{do } m \) becomes \( m \)

- \( \text{do } \{ m; n; \ldots \} \) becomes \( m \gg \text{do } \{ n; \ldots \} \)

- \( \text{do } \{ v <- m; n; \ldots \} \) becomes \( m \gg= \{ v -> \text{do } \{ n; \ldots \} \} \)

Example:

\[
\text{do } c <- \text{getChar} \\
\quad \text{putChar } c \\
\quad \text{return } c
\]

is simply: \( \text{getChar } \gg= (c \rightarrow \text{putChar } c \gg \text{return } c) \)