I/O in Haskell

Generally, I/O functions in Haskell have type `IO a`, where `a` could be any type. The purpose and use of `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:

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

  e.g., `putChar 'c'`

- To output a string:

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

  e.g., `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:

  ```
  do putChar 'c'
     putStr "Hello"
  ```

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

- To perform those two commands, then loop back:

  ```
  myloop :: IO ()
  myloop = do putChar 'c'
              putStr "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:

```
getChar :: IO Char
```

It says, "`getChar` is a command that returns a character".

(In retrospect, `IO ()` returns "void".)

```
do c <- getChar  --this is how we obtain the character
   --c is now a Char and you can use it, e.g.,
   putChar c
   --more reading and writing
   d <- getChar
   putChar d
```


Commands That Return Data

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

```
getLine :: IO String
```

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

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

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

```
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:

```
isEOF :: IO 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 = do b <- isEOF
          if b then return ()
               else do c <- getChar
                       putChar c
                       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:

```
catch :: IO a -> (IOError -> IO a) -> 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:

```
isEOFError :: IOError -> Bool  --in module IO
```

You can also re-throw the exception with:

```
ioError :: IOError -> IO a
```


Exception Example

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

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

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

```
dump = do c <- getChar
          putChar
          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:

```
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:

```
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:

```
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:

- `do m` becomes `m`
- `do { m; n; ... }` becomes `m >> do { n; ... }`
- `do { v <- m; n; ... }` becomes `m >>= \v -> do { n; ... }`

Example:

```
do c <- getChar
   putChar c
   return c
```

is simply: `getChar >>= (\c -> putChar c >> return c)`

(blank)

(blank)