3.7 In- and Output

Conceptually, functional languages have a problem with input and output, since reading in data is not well modelled using functions (which usually can become a side-effect of functions (and, as such, outside the usual semantical treatment of function evaluation by reductions). Other languages try to stay within the functional IO part (usually copied from an imperative language) simply and to their functional part a rather standard I/O part add to their functional part. Some functional languages do not care about this and rather confusing, see Haskell)
The monad construct was introduced in category theory, a rather theoretic field. Haskell has three type classes that are based on the monad, a rather theoretic field.

Haskell: the monadic classes (I)
This naturally allows for a (theoretically) better treatment of IO (as actions in the outside world).

The Monad class essentially defines around "normal" functions an environment that is (or can be) changed when these functions are performed (i.e., we convert side effects into valid function results in the extended "world").

Monads are similar to abstract data types since they require each subclass/instantiation of them to obey certain laws (i.e., there are certain equations between expressions that we expect to be fulfilled). Haskell: the monadic classes (II)
For the basic data types `Char` and `String` ([`Char`]), Haskell has a corresponding IO type that represents values of the basic type with the added "world environment" (corresponding to the unit type `()`) to the function as result type `IO ()` (the IO data type corresponding to the unit type).

If we are only interested in the effects on the environment (i.e., if we write out data) then we assign the monad functions `>>=` and `<<=` or we can use the do construct to produce sequences of actions, we can either use the monad functions `>>=` and `<<=` or we can use the do construct.

For the basic data types `Char` and `String ([`Char`]) Haskell

IO in Haskell (I)
For reading and writing a character, we use the built-in functions `getChar` and `putChar`.

For other data types, the type classes `Show` and `Read` force the existence of functions that convert values of the types into characters or strings, resp.; functions that convert characters or strings into values of the other types:

```
show (2+5) returns "7"
reads ("True") returns True
```

For other data types, the type classes `Show` and `Read` build-in functions `getChar` and `putChar`.

For reading and writing a character, we use the `IO` in Haskell.
The following little program reads in one character and then prints it out:

```haskell
main :: IO ()
main = do
    c <- getChar
    putChar c
```

Note that do allows for the sequence of the two actions and that `c` acts here very much like a variable.

The following little program reads in one character:

```haskell
IO in Haskell (III)
```
File handling

Files (many in the IO library), to read/write lines or whole

There are quite a few additional functions available

and hgetChar (with a handle as first argument)

The handle variants of putChar and getChar are hputChar

to read or write from the file associated with it

an IOMode and can then be used by several functions

handles for them. A handle requires a file path and

Other channels and files can be used by creating

operating systems we call this streams

stdin (which Haskell calls channels, in modern

putChar and getChar write to stdout and read from
While for normal functions it might be acceptable to let the run-time system terminate with an error when they produce an error, there are a lot of "normal" error conditions associated with IO (like end-of-file) that convert values from `IOError` to the normal `IO a` values.

IO exception handling (1)
IO exception handling (II)

Example (see "Gentle Introduction"):  

```
return (c:l)
else do l -> getLineWErr
  if c == \n then return ""
  gl = do c -> getChar
  where
getLineWErr = catch gl (\err -> return ("Error: " ++ show err))
getLineWErr :: IO String
```
3.8 Paradigm-specific and language-specific constructs

We have already seen several rather Haskell-specific language-specific constructs. There are no additional language-specific constructs outside of the topics we covered briefly.

- Type classes
- List comprehensions
- Monads
- Layout

We have already seen several rather Haskell-specific language-specific constructs.
One construct that most functional languages have is some kind of eval statement. Usually this requires the language to be interpreted. eval takes a list of strings (that is generated by the program somewhere) and treats it as addition to the already defined functions, thus essentially allowing a program to modify itself. This allows modifying a program to be done by evaluation.
3.9 Functional Programming in comparison to other paradigms

Control Flow: If several reductions are possible, then...

Memory management: What memory? Evaluation can require a sequential flow.

Program state: not explicitly represented, at best the current evaluation state of the main expression could serve as state.

Declaration = instruction; Declaration and instruction:

Jörg Denzinger

CPSC 449 Principles of Programming Languages
Pattern matching (albeit only primitive form)

Memory management/ Garbage Collection

Type checking

Polymorphism

Compiler/Run time system intelligence level:

Haskell: quite high, especially in newer languages like

Comparison to other paradigms:

Functional Programming: