Informatics 1
Functional Programming Lectures 15 and 16

IO and Monads

Don Sannella
University of Edinburgh
Everything in Haskell is "pure" in the sense of having no SIDE EFFECTS.
A side effect is something that happens "off to the side" during evaluation - input/output, change of state, etc.

A function in Java, C etc. might not be a real function - it might give different results when applied twice to the same value, e.g. because the result is dependent on input from the keyboard, or on the date. Then the order of evaluation matters.
If parts of the computation are done in parallel, then their relative timings can affect the result.

So side effects make things very complicated. But pure functions can be very inconvenient.
How do I get Haskell to DO SOMETHING - print, draw a picture, etc.?
Philosophers worried about the connection between THOUGHTS (mind) and ACTIONS (body). Where is the connection? Descartes: the pineal gland (teardrop-shaped object in picture) is the connection between the soul (mind) and the brain/nerves (connected to sensory organs and muscles). Haskell has something like a pineal gland, to make the connection between thinking (computation) and acting.
Part II

 Commands
Print a character

\[
\text{putChar :: Char \to IO ()}
\]

For instance,

\[
\text{putChar ‘!’}
\]

denotes the command that, \textit{if it is ever performed}, will print an exclamation mark.

Think of IO () as the type of COMMANDS. () is the type of 0-tuples. The only value of type () is the 0-tuple, also written ().

putChar yields a command. It doesn't PERFORM the command, just PRODUCES it. Production of commands is purely functional.
Combine two commands

\[(\gg\gg) :: \text{IO ()} \rightarrow \text{IO ()} \rightarrow \text{IO ()}\]

For instance,

\[
\text{putChar } '?' \gg \text{putChar } '!' \\
\]

denotes the command that, \textit{if it is ever performed}, prints a question mark followed by an exclamation mark.

\gg (pronounced "then") combines two commands sequentially.
\gg is associative but not commutative. Its identity is the command that does nothing.
Do nothing

\texttt{done :: IO ()}

The term \texttt{done} doesn’t actually do nothing; it just specifies the command that, \textit{if it is ever performed}, won’t do anything. (Compare thinking about doing nothing to actually doing nothing: they are distinct enterprises.)

done is the identity for \texttt{>>}.
\texttt{c \texttt{>>} done = c = done \texttt{>>} c}
Print a string

\[
\begin{align*}
\text{putStr} :: \text{String} & \rightarrow \text{IO} () \\
\text{putStr} \; [] & = \text{done} \\
\text{putStr} \; (x:xs) & = \text{putChar} \; x \; >> \; \text{putStr} \; xs
\end{align*}
\]

So \text{putStr} \; "?!" is equivalent to

\[
\text{putChar} \; '\' \; >> \; (\text{putChar} \; '!' \; >> \; \text{done})
\]

and both of these denote a command that, \textit{if it is ever performed}, prints a question mark followed by an exclamation mark.

\begin{itemize}
  \item \text{putStr} produces a command that prints a string.
\end{itemize}
Higher-order functions

More compactly, we can define `putStr` as follows.

- `putStr :: String -> IO ()`
- `putStr = foldr (>>) done . map putChar`

The operator `>>` has identity `done` and is associative.

- `m >> done = m`
- `done >> m = m`
- `(m >> n) >> o = m >> (n >> o)`

Here's how we can define `putStr` more compactly.
You can use all of the features of Haskell to produce commands.
Main

By now you may be desperate to know *how is a command ever performed?* Here is the file Confused.hs:

```haskell
module Confused where

main :: IO ()
main = putStr "?!"
```

Running this program prints an indicator of perplexity:

```
[melchior]dts: runghc Confused.hs
?![melchior]dts:
```

Thus `main` is the link from Haskell’s mind to Haskell’s body — the analogue of Descartes’s pineal gland.

This is enough, because we can make arbitrarily large combinations of things into a single command.
Print a string followed by a newline

```haskell
putStrLn :: String -> IO ()
putStrLn xs = putStr xs >> putChar '\n'
```

Here is the file `ConfusedLn.hs`:

```haskell
module ConfusedLn where

main :: IO ()
main = putStrLn "?!"
```

This prints its result more neatly:

```
[melchior]dts: runghc ConfusedLn.hs
?!
[melchior]dts:
```

ghci evaluates expression and then uses show to display the resulting value. For commands, show performs the command. So you don't need to use runghc.
Part III

Equational reasoning
Equational reasoning lost

In languages with side effects, this program prints “haha” as a side effect.

```
print "ha"; print "ha"
```

But this program only prints “ha” as a side effect.

```
let x = print "ha" in x; x
```

The value x of print "ha" is unimportant. It’s the side effect that matters.

This program again prints “haha” as a side effect.

```
let f () = print "ha" in f (); f ()
```

Each evaluation of f () produces the side effect.

Side effects make it harder to reason about programs. Here is an example of what happens in a language (not Haskell) with side effects.
Equational reasoning regained

In Haskell, the term

\[(1+2) \times (1+2)\]

and the term

```haskell
let x = 1+2 in x \times x
```

are equivalent (and both evaluate to 9).

In Haskell, the term

```haskell
putStr "ha" >> putStr "ha"
```

and the term

```haskell
let m = putStr "ha" in m >> m
```

are also entirely equivalent (and both print "haha").

In Haskell, the simple equivalence rule works, even with commands that involve printing things. You can always use a variable to factor out a common subexpression without changing the meaning.
Part IV

Commands with values
Read a character

Previously, we wrote $\texttt{IO ()}$ for the type of commands that yield no value. Here, $()$ is the trivial type that contains just one value, which is also written $()$. We write $\texttt{IO Char}$ for the type of commands that yield a value of type $\texttt{Char}$. Here is a command to read a character.

$$\texttt{getChar :: IO Char}$$

Performing the command $\texttt{getChar}$ when the input contains "abc" yields the value 'a' and remaining input "bc".
Do nothing and return a value

More generally, we write \( \text{IO } a \) for commands that return a value of type \( a \).

The command

\[
\text{return } :: a \rightarrow \text{IO } a
\]

is similar to \( \text{done} \), in that it does nothing, but it also returns the given value.

Performing the command

\[
\text{return } [] :: \text{IO } \text{String}
\]

when the input contains "bc" yields the value [ ] and an unchanged input "bc".

\( \text{return} \) is useful when combined with other commands - we'll see how in a minute.
Combining commands with values

We combine command with an operator written >>= and pronounced “bind”.

\[
(\gg\gg) :: \text{IO } a \to (a \to \text{IO } b) \to \text{IO } b
\]

For example, performing the command

\[
\text{getChar } \gg\gg \ \lambda x \to \text{putChar} \ (\text{toUpper } x)
\]

when the input is "abc" produces the output "A", and the remaining input is "bc".

>>= is a generalisation of >> which handles the values produced when commands are performed.
The “bind” operator in detail

\[(>>=) :: \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b\]

If

\[m :: \text{IO } a\]

is a command yielding a value of type \(a\), and

\[k :: a \rightarrow \text{IO } b\]

is a function from a value of type \(a\) to a command yielding a value of type \(b\), then

\[m >>= k :: \text{IO } b\]

is the command that, if it is ever performed, behaves as follows:

first perform command \(m\) yielding a value \(x\) of type \(a\); then perform command \(k \ x\) yielding a value \(y\) of type \(b\); then yield the final value \(y\).

Performing this command performs \(m\) and then \(k\). The rest is about passing a value from \(m\) to \(k\), and the fact that \(k\) is affected by the value passed.
Reading a line

Here is a program to read the input until a newline is encountered, and to return a list of the values read.

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

For example, given the input "abc\ndef" This returns the string "abc" and the remaining input is "def".

1. Get a character, call it x.
2. if it's a new line, finished - return "".
3. otherwise, read the rest of the line (recursive call of geline). This produces a string, call it xs.
4. return x:xs.

Without return, we couldn't write this!
Commands as a special case

The general operations on commands are:

\[
\begin{align*}
\text{return} & : : \ a \rightarrow \text{IO} \ a \\
(\gg\gg=) & : : \text{IO} \ a \rightarrow (a \rightarrow \text{IO} \ b) \rightarrow \text{IO} \ b
\end{align*}
\]

The command \textit{done} is a special case of \textit{return},
and the operator \textit{>>} is a special case of \textit{>>=}.

\[
\begin{align*}
\text{done} & : : \text{IO} \ () \\
\text{done} & = \text{return} \ ()
\end{align*}
\]

\[
\begin{align*}
(\gg) & : : \text{IO} \ () \rightarrow \text{IO} \ () \rightarrow \text{IO} \ () \\
m \gg n & = m \gg= \ \backslash(()) \rightarrow n
\end{align*}
\]
An analogue of “let”  

"let x = m in n" is another way of writing "n where x = m".

Although it may seem odd at first sight, this combinator is reassuringly similar to the familiar Haskell “let” expression. Here is a type rule for “let”.

\[
\begin{align*}
E \vdash m :: a & \quad \text{m is an expression of type } a \\
E, x :: a \vdash n :: b & \quad \text{n is an expression of type } b, \text{ with a variable } x \text{ of type } a. \\
E \vdash \text{let } x = m \text{ in } n :: b & \quad \text{then let } x=m \text{ in } n \text{ is an expression of type } b
\end{align*}
\]

Typically, “bind” is combined with lambda expressions in a way that resembles “let” expressions. Here is the corresponding type rule.

\[
\begin{align*}
E \vdash m :: \text{IO } a & \quad \text{m returns a value of type } a \\
E, x :: a \vdash n :: \text{IO } b & \quad \text{n returns a value of type } b, \text{ contains a variable of type } a. \\
E \vdash m >>= \ \lambda x \to n :: \text{IO } b & \quad m >>= \lambda x->n \text{ does } m, \text{ uses result to do } n, \text{ returns value of type } b
\end{align*}
\]
Echoing input to output

This program echoes its input to its output, putting everything in upper case, until an empty line is entered.

```haskell
  echo :: IO ()
  echo = getLine >>= \line ->
    if line == "" then
      return ()
    else
      putStrLn (map toUpper line) >>
      echo

  main :: IO ()
  main = echo
```

1. Get a line of input, call it line.
2. If it's empty, we're done.
3. Otherwise, print the line in upper case.
4. Then do it again.
Testing it out

[melchior]dts: runghc Echo.hs
One line
ONE LINE
And, another line!
AND, ANOTHER LINE!
[melchior]dts:
Part V

“Do” notation
Reading a line in “do” notation

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

is equivalent to

```
getLine :: IO String
getLine = do {
    x <- getChar;
    if x == '\n' then
        return []
    else do {
        xs <- getLine;
        return (x:xs)
    }
}
```

This is a special notation that makes this easier.

```
do { x <- cmd; exp }  is the same as  cmd >>= \x -> exp
```
Echoing in “do” notation

```haskell
echo :: IO ()
echo = getLine >>= \line ->
  if line == "" then
    return ()
  else
    putStrLn (map toUpper line) >>
    echo
```

is equivalent to

```haskell
echo :: IO ()
echo = do {
  line <- getLine;
  if line == "" then
    return ()
  else do {
    putStrLn (map toUpper line);
    echo
  }
}
```

do { cmd; exp } is the same as cmd >>= exp which is short for cmd >>= \() -> exp"
“Do” notation in general

Each line \( x \leftarrow e; \ldots \) becomes \( e >>= \lambda x \rightarrow \ldots \)

Each line \( e; \ldots \) becomes \( e >> \ldots \)

For example,

\[
\text{do } \{ \text{x1 } \leftarrow \text{e1}; \\
\text{x2 } \leftarrow \text{e2}; \\
\text{e3}; \\
\text{x4 } \leftarrow \text{e4}; \\
\text{e5}; \\
\text{e6 } \} \\
\]

is equivalent to

\[
\text{e1 } >>= \lambda \text{x1 } \rightarrow \\
\text{e2 } >>= \lambda \text{x2 } \rightarrow \\
\text{e3 } >> \\
\text{e4 } >>= \lambda \text{x4 } \rightarrow \\
\text{e5 } >> \\
\text{e6 } \\
\]

Here is the general case. The result is the value returned by \( e6 \).
Part VI

Monads
Monoids

A *monoid* is a pair of an operator \((@@)\) and a value \(u\), where the operator has the value as identity and is associative.

\[
\begin{align*}
u @@ x & = x \\
x @@ u & = x \\
(x @@ y) @@ z & = x @@ (y @@ z)
\end{align*}
\]

Examples of monoids:

\[
\begin{align*}
(+) & \text{ and } 0 \\
(*) & \text{ and } 1 \\
(||) & \text{ and } \text{False} \\
(&&) & \text{ and } \text{True} \\
(++) & \text{ and } [] \\
(>>/) & \text{ and } \text{done}
\end{align*}
\]
Monads

We know that (>>) and done satisfy the laws of a monoid.

\[
\begin{align*}
done \gg m &= m \\
m \gg done &= m \\
(m \gg n) \gg o &= m \gg (n \gg o)
\end{align*}
\]

Similarly, (>>=) and return satisfy the laws of a monad.

\[
\begin{align*}
\text{return } v \gg= \ \lambda x \to m &= m[x:=v] \\
m \gg= \ \lambda x \to \text{return } x &= m \\
(m \gg= \ \lambda x \to n) \gg= \ \lambda y \to o &= m \gg= \ \lambda x \to (n \gg= \ \lambda y \to o)
\end{align*}
\]

A monad is a generalised version of a monoid.
The monad laws are the closest we can get to the monoid laws when >>= and return have the types they have.
**Laws of Let**

We know that \( (>>) \) and \texttt{done} satisfy the laws of a \textit{monoid}.

\[
\begin{align*}
\texttt{done} >> m & = m \\
\texttt{m} >> \texttt{done} &= \texttt{m} \\
(\texttt{m} >> \texttt{n}) >> \texttt{o} &= \texttt{m} >> (\texttt{n} >> \texttt{o})
\end{align*}
\]

Similarly, \((>>=)\) and \texttt{return} satisfy the laws of a \textit{monad}.

\[
\begin{align*}
\texttt{return} v >>= \lambda x -> m &= m[x:=v] \\
\texttt{m} >>= \lambda x -> \texttt{return} x &= \texttt{m} \\
(\texttt{m} >>= \lambda x -> \texttt{n}) >>= \lambda y -> \texttt{o} &= \texttt{m} >>= \lambda x -> (\texttt{n} >>= \lambda y -> \texttt{o})
\end{align*}
\]

The three monad laws have analogues in “let” notation.

\[
\begin{align*}
\texttt{let} \ x = v \ \texttt{in} \ \texttt{m} &= m[x:=v] \\
\texttt{let} \ x = m \ \texttt{in} \ x &= m \\
\texttt{let} \ y = (\texttt{let} \ x = m \ \texttt{in} \ n) \ \texttt{in} \ o &= \texttt{let} \ x = m \ \texttt{in} \ (\texttt{let} \ y = n \ \texttt{in} \ o)
\end{align*}
\]
“Let” in languages with and without effects

\[
\begin{align*}
\text{let } x = v \text{ in } m &= m[x:=v] \\
\text{let } x = m \text{ in } x &= m \\
\text{let } y = (\text{let } x = m \text{ in } n) \text{ in } o &= \text{let } x = m \text{ in } (\text{let } y = n \text{ in } o)
\end{align*}
\]

These laws hold even in languages with side effects. For the first law to be true, \(v\) must be not an arbitrary term but a \textit{value}, such as a constant or a variable (but not a function application). A value immediately evaluates to itself, hence it can have no side effects.

While in such languages one only has the above three laws for “let”, in Haskell one has a much stronger law, where one may replace a variable by any term, rather than by any value.

\[
\text{let } x = n \text{ in } m = m[x:=n]
\]
Part VII

Roll your own monad—IO
The Monad type class

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

There is a type class for monads, and you can create instances yourselves.
Notice: in previous type classes, the variable (m here) referred to a TYPE.
But m here is a TYPE CONSTRUCTOR - a function that takes a type and yields a type.
We've seen the case where m is IO.
My own IO monad (1)

```haskell
module MyIO(MyIO, myPutChar, myGetChar, convert) where

type Input = String

type Remainder = String

type Output = String

data MyIO a = MyIO (Input -> (a, Remainder, Output))

apply :: MyIO a -> Input -> (a, Remainder, Output)
apply (MyIO f) inp = f inp

Note that the type MyIO is abstract. The only operations on it are the monad operations, myPutChar, myGetChar, and convert. The operation apply is not exported from the module.
```

Here's how you can build your own version of IO. Doing input/output, I take an input string and when I'm done I return a value, the unread part of the input string, and an output string. MyIO is an abstract type - the constructor of MyIO is not exported.
My own IO monad (2)

```haskell
myPutChar :: Char -> MyIO ()
myPutChar c = MyIO (\inp -> ((), inp, [c]))

myGetChar :: MyIO Char
myGetChar = MyIO (\(ch:rem) -> (ch, rem, ""))
```

For example,

```haskell
apply myGetChar "abc" == ('a', "bc", "")
apply myGetChar "bc" == ('b', "c", "")
apply (myPutChar 'A') "def" == ((), "def", "A")
apply (myPutChar 'B') "def" == ((), "def", "B")
```

myPutChar \(c\) takes input, returns no value, leaves input unchanged, and returns the string "c" as input.
myGetChar takes input, returns the first character, the remainder of the input is everything after that, and does no output.
My own IO monad (3)

```haskell
instance Monad MyIO where
    return x = MyIO (\inp -> (x, inp, ""))  -- return x takes input, returns x,
    m >>= k = MyIO (\inp ->                  -- leaves input unchanged, does no output.
        let (x, rem1, out1) = apply m inp in
        let (y, rem2, out2) = apply (k x) rem1 in
        (y, rem2, out1++out2))

For example
(1) applies m to input, getting value x, remaining input rem1, output out1
(2) applies k x to remaining input, getting value y, remaining input rem2, output out2
(3) result is y, remaining input is still rem2, output is concatenation of out1 and out2

apply
  (myGetChar >>= \x -> myGetChar >>= \y -> return [x,y])
    "abc"
== ("ab", "c", "")

apply
  (myPutChar 'A' >>= myPutChar 'B')
    "def"
== ((), "def", "AB")

apply
  (myGetChar >>= \x -> myPutChar (toUpper x))
    "abc"
== ((), "bc", "A")
```
My own IO monad (4)

convert :: MyIO () -> IO ()
convert m = interact (\inp ->
    let (x, rem, out) = apply m inp in
    out)

Here

interact :: (String -> String) -> IO ()

is part of the standard prelude. The entire input is converted to a string (lazily) and passed to the function, and the result from the function is printed as output (also lazily).

interact connects a function to actual input/output, making the connection to the keyboard and screen.
Using my own IO monad (1)

```haskell
module MyEcho where

import Char
import MyIO

myPutStr :: String -> MyIO ()
myPutStr = foldr (>>) (return ()) . map myPutChar

myPutStrLn :: String -> MyIO ()
myPutStrLn s = myPutStr s >> myPutChar '\n'
```

I can use this to build another version of the echo function. First I versions of putStr and putStrLn for MyIO.
Using my own IO monad (2)

```haskell
myGetLine :: MyIO String
myGetLine = myGetChar >>= \x ->
    if x == '\n' then
        return []
    else
        myGetLine >>= \xs ->
        return (x:xs)

myEcho :: MyIO ()
myEcho = myGetLine >>= \line ->
    if line == "" then
        return ()
    else
        myPutStrLn (map toUpper line) >>
        myEcho

main :: IO ()
main = convert myEcho
```

Then I need a version of getline for MyIO.
Finally, I write echo as before.
convert myEcho turns myEcho into IO().
Trying it out

[melchior]dts: runghc MyEcho
This is a test.
THIS IS A TEST.
It is only a test.
IT IS ONLY A TEST.
Were this a real emergency, you’d be dead now.
WERE THIS A REAL EMERGENCY, YOU’D BE DEAD NOW.

If you run it, the result is the same as before.
The only thing that wasn't pure Haskell functions and values was interact.
You can use “do” notation, too

```haskell
myGetLine :: MyIO String
myGetLine = do {
    x <- myGetChar;
    if x == '
' then
        return []
    else do {
        xs <- myGetLine;
        return (x:xs)
    }
}

myEcho :: MyIO ()
myEcho = do {
    line <- myGetLine;
    if line == "" then
        return ()
    else do {
        myPutStrLn (map toUpper line);
        myEcho
    }
}
```

Because MyIO is a monad, you can use "do" notation.
Part VIII

The monad of lists

A lot of things turn out to have the structure of a monad.
We'll look first at lists, used to model non-deterministic computation.
The monad of lists

In the standard prelude:

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

```haskell
instance Monad [] where

    return :: a -> [a]
    return x = [ x ]

    (>>=) :: [a] -> (a -> [b]) -> [b]
    m >>= k = [ y | x <- m, y <- k x ]
```

Equivalently, we can define:

```haskell
[] >>= k = []
(x:xs) >>= k = (k x) ++ (xs >>= k)
```

or

```haskell
m >>= k = concat (map k m)
```
‘Do’ notation and the monad of lists

pairs :: Int -> [(Int, Int)]
pairs n = [ (i,j) | i <- [1..n], j <- [(i+1)..n] ]

is equivalent to

pairs' :: Int -> [(Int, Int)]
pairs' n = do
  i <- [1..n];
  j <- [(i+1)..n];
  return (i,j)

For example,

[melchior@dts: ghci Pairs
GHCi, version 6.10.4: http://www.haskell.org/ghc/  :? for help
Pairs> pairs 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
Pairs> pairs' 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]

Notice that "do"-notation is now a lot like comprehension notation!
Monads with plus

In the standard prelude:

```haskell
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

instance MonadPlus [] where
  mzero :: [a]
  mzero = []

  mplus :: [a] -> [a] -> [a]
  mplus = (++)

guard :: MonadPlus m => Bool -> m ()
guard False = mzero
guard True = return ()

msum :: MonadPlus m => [m a] -> m a
msum = foldr mplus mzero
```

Some monads have extra structure:
- a plus operation and a zero value
- that is the identity for plus.
- MonadPlus is a monad with these extra things.

Non-deterministic interpretation:
- mzero = [] = no possible result
- return () = [] = one possible result

msum is a generalisation of concat
Using guards

```
pairs'' :: Int -> [(Int, Int)]
pairs'' n = [ (i,j) | i <- [1..n], j <- [1..n], i < j ]
```

is equivalent to

```
pairs''' :: Int -> [(Int, Int)]
pairs''' n = do {
    i <- [1..n];
    j <- [1..n];
    guard (i < j);
    return (i,j)
}
```

The effect of guard \((i < j)\) is to drop the computation or continue.
Think of it as \(z <- \text{guard } (i < j)\):
if False, there is no \(z\) so we can't continue
if True, \(z=()\) and we continue (but don't use \(z\))

For example,

```
[melchior]dts: ghci Pairs
GHCi, version 6.10.4: http://www.haskell.org/ghc/  ?? for help
Pairs> pairs'' 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
Pairs> pairs''' 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

Now we have all of comprehension notation arising via "do"-notation in a monad with plus.
Part IX

Parsers

We will use monads to build a parser for expression trees.
Parser type

First attempt:  
A parser for values of type `a` takes a string and produces a value of type `a`.

```haskell
type Parser a = String -> a
```

Second attempt:  
But at each stage, you have a part that you've parsed and a remainder that is left to parse.

```haskell
type Parser a = String -> (a, String)
```

Third attempt:  
But sometimes there will be more than one parse, or none. So you need a list of possibilities.

```haskell
type Parser a = String -> [(a, String)]
```

---

*A parser for things is a function from strings to lists of pairs*  
*Of things and strings*  

—Graham Hutton
Module Parser

```haskell
module Parser(Parser,apply,parse,char,spot,token,
    star,plus,parseInt) where

import Char
import Monad

-- The type of parsers
data Parser a = Parser (String -> [(a, String)])

-- Apply a parser
apply :: Parser a -> String -> [(a, String)]
apply (Parser f) s = f s

-- Return parsed value, assuming at least one successful parse
parse :: Parser a -> String -> a
parse m s = head [x | (x,t) <- apply m s, t == ""]
```

Parser is an abstract data type - the constructor is not exported. There is an implicit invariant: the string produced is always a final substring of the string consumed. `parse m s` looks for a result `(x,t)` from `apply m s` which completely consumes `s`. We take the first one we find. (If there is more than one, ambiguous - we could return an error.)
Parser is a Monad

-- Parsers form a monad

-- class Monad m where
--     return :: a -> m a
--     (>>=) :: m a -> (a -> m b) -> m b

instance Monad Parser where
    return x = Parser (\s -> [(x, s)])
    m >>= k = Parser (\s ->
        [(y, u) |
            (x, t) <- apply m s,
            (y, u) <- apply (k x) t ])

Parsers form a monad

return x: just return x, without consuming any input

m >>= k:
(1) apply m to s, giving possibilities (x,t) where t is the unparsed remainder
(2) apply (k x) to t, giving possibilities (y,u) where u is the final unparsed remainder
(3) the result is all of the possible (y,u) resulting from all of the possible (x,t)
Parser is a Monad with Plus

-- Some monads have additional structure

-- class MonadPlus m where
--   mzero :: m a
--   mplus :: m a -> m a -> m a

instance MonadPlus Parser where
  mzero = Parser (\s -> [])
  mplus m n = Parser (\s -> apply m s ++ apply n s)

Parsers also form a monad with plus.
The monad structure gives SEQUENCING.
The plus structure gives ALTERNATIVES.
Parsing characters

-- Parse a single character
char :: Parser Char
char = Parser f
  where
    f [] = []
    f (c:s) = [(c,s)]

Here is a parser that parses the first character in the input.
If input is empty, failure: empty list of possibilities.
If input is non-empty, first character is result, tail is remaining string.

-- Parse a character satisfying a predicate (e.g., isDigit)
spot :: (Char -> Bool) -> Parser Char
spot p = Parser f
  where
    f [] = []
    f (c:s) | p c = [(c,s)]
    | otherwise = []

This is similar, but requires the character to satisfy a given predicate.
So spot isDigit will only parse digits.

-- Parse a given character
token :: Char -> Parser Char
token c = spot (== c)  
token c will parse the character c only.
It will fail if the next character is something else.
Parsing characters

-- Parse a single character
char :: Parser Char
char = Parser f

  where
    f []  = []
    f (c:s) = [(c,s)]

-- Parse a character satisfying a predicate (e.g., isDigit)
spot :: (Char -> Bool) -> Parser Char
spot p = do { c <- char; guard (p c); return c }

  Here's the same thing, using do-notation and guard; remember, guard is defined for any MonadPlus.

-- Parse a given character
token :: Char -> Parser Char
token c = spot (== c)

Examples:
apply (spot isDigit) "123ab" = [('1', "23ab")]
apply (spot isDigit) "ab" = []
apply (token 'a') "ab" = [('a', "b")]
apply (do { x <- spot isDigit; token '+'; y <- spot isDigit; return (digitToInt x + digitToInt y) }) "1+2*4" = [(3, "*4")]

Parsing a string

match :: String -> Parser String
match [] = return []
match (x:xs) = do
    y <- token x;
    ys <- match xs;
    return (y:ys)

To parse a whole string:
if it is empty, you're done - return ""
if it is x:xs
(1) y is the result of token x (y = x, but token x will consume a character from the input and fail if it isn't x)
(2) then parse xs, yielding ys (again, ys = xs)
(3) then return y:ys
Parsing a sequence

-- match zero or more occurrences
star :: Parser a -> Parser [a]
star p = plus p 'mplus' return []

-- match one or more occurrences
plus :: Parser a -> Parser [a]
plus p = do { x <- p;
            xs <- star p;
            return (x:xs) }

star p is like p* from regular expressions: it parses zero or more occurrences of p.
plus p parses one or more occurrences of p.
Note the mutual recursion.

Example:
apply (star (spot isDigit)) "123ab" = ["123", "ab"], ["12", "3ab"], ["1", "23ab"], ["", "123ab"]
Parsing an integer

-- match a natural number
parseNat :: Parser Int
parseNat = do { s <- plus (spot isDigit); return (read s) }

-- match a negative number
parseNeg :: Parser Int
parseNeg = do { token '-'; n <- parseNat
    return (-n) }

-- match an integer
parseInt :: Parser Int
parseInt = parseNat 'mplus' parseNeg

parse an integer: parse either a positive natural number or a negative one
Module Exp

module Exp where

import Monad
import Parser

data Exp = Lit Int
    | Exp :+: Exp
    | Exp :*: Exp
    deriving (Eq, Show)

evalExp :: Exp -> Int
evalExp (Lit n)    = n
evalExp (e :+: f) = evalExp e + evalExp f
evalExp (e :*: f) = evalExp e * evalExp f

Here's Exp (expression trees) from before, with an evaluation function.
Parsing an expression

\[
p_{\text{parseExp}} :: \text{Parser Exp} \\
p_{\text{parseExp}} = p_{\text{parseLit}} \text{ `mplus` } p_{\text{parseAdd}} \text{ `mplus` } p_{\text{parseMul}}
\]

where

An Exp is either a literal, or an addition, or a multiplication

\[
p_{\text{parseLit}} = \text{do} \{ n <- p_{\text{parseInt}}; \text{return (Lit } n \text{)} \}
\]

Literal: parse an integer \( n \), return Lit \( n \)

\[
p_{\text{parseAdd}} = \text{do} \{ \text{token `(`}; d <- p_{\text{parseExp}}; \text{token `+`; } e <- p_{\text{parseExp}}; \text{token `)`}; \text{return (} d \text{ :+: } e \text{)} \}
\]

Addition:
(1) parse ( 
(2) parse an Exp \( d \) 
(3) parse + 
(4) parse an Exp \( e \) 
(5) parse ) 
(6) return \( d \text{ :+: } e \)

\[
p_{\text{parseMul}} = \text{do} \{ \text{token `(`}; d <- p_{\text{parseExp}}; \text{token `*`; } e <- p_{\text{parseExp}}; \text{token `)`}; \text{return (} d \text{ : * : } e \text{)} \}
\]

Multiplication: similarly

Notice: after you've read (Exp you don't know if the next thing will be + or *. mplus handles that. Also, expressions can be nested. Recursion handles that.
Testing the parser

And it works! We can parse strings to give Exp values, which can be evaluated.