But first....Lambda Expressions

Before we get into Monads, we first have to become more comfortable with using lambda expressions, which we say before.

As we have seen, a lambda expression is of the form:

\(<\text{var list}> \rightarrow \text{body}> \) \text{args}

\(<\text{x} \rightarrow \text{x} \ + \ \text{x} > 2 \rightarrow 3 >
\(<\text{x} \rightarrow \text{x} \ast \ \text{x} > 5 \rightarrow 25 >
\(<\text{x} \ \text{y} \rightarrow \text{x} \ + \ \text{y} > 2 \ 3 \rightarrow 5 >
\(<\text{x} \ \text{y} \rightarrow \text{x} \ + \ \text{y} > 2 \rightarrow (>\text{y} \rightarrow 2 \ + \ \text{y}) >
\(<\text{(x,y)} \rightarrow \text{x} \ + \ \text{y} > (2,3) \rightarrow 5 >
\(<\text{(x,y)} \rightarrow \text{x} \ + \ \text{y} > 2 \rightarrow >\text{error} >

Lambda expressions

Lambda expressions are most often used with standard higher-order functions, such as foldl/foldr, scanl/scanr, map, etc.

\[
\begin{align*}
\text{id} &= \lambda x \rightarrow x \\
\text{succ} &= \lambda n \rightarrow n + 1 \\
\text{square} &= \lambda x \rightarrow x \times x \\
\text{sum} &= \text{foldl} (\lambda x y \rightarrow x + y) 0 \\
\text{prod} &= \text{foldl} (\lambda x y \rightarrow x \times y) 1 \\
\text{len} &= \text{foldl} (\lambda x _ \rightarrow x + 1) 0 \\
\text{fact} n &= \text{foldl} (\lambda x y \rightarrow x \times y) 1 [1..n] \\
\text{suml} :: (\text{Num a}) \Rightarrow [a] \rightarrow (a,\text{Int}) \\
\text{suml} &= \text{foldl} (\lambda (s,l) x \rightarrow (s+x, l+1)) (0,0)
\end{align*}
\]

Fixed Points

A “fixed point” is a value \( x \) in the domain of a function that is the same in the co-domain \( f(x) \). Functionals may also have fixed points, i.e., a higher-order function returns its function argument as the result.

\[
\begin{align*}
\text{id} x &= x, \text{for all } x \\
\text{fact} 1 &= 1, \text{fact} 2 = 2 \\
\text{fibonacci} 0 &= 0, \text{fibonacci} 1 = 1 \\
\text{square} 0 &= 0, \text{square} 1 = 1 \\
\text{abs} x &= x, \text{if } x \geq 0 \\
\sin 0 &= 0 \\
D(e^x) &= e^x
\end{align*}
\]
A Fixed Point Operator

Fix applied to any function $F$ returns $F$ and repeats $F$ by applying Fix to $F$ one more time. Fix is actually the fixed point operator commonly defined in the untyped $\lambda$-calculus called the (lazy) $Y$ combinator. $Y$ is also called the “paradoxical combinator” because it was the source of the logical paradox of the original $\lambda$-calculus.

$$\text{Fix } F = F (\text{Fix } F)$$

$$Y = (\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx)))$$

$$Y F = (\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))) F$$

$$= (\lambda x. F(xx)) (\lambda x. F(xx))$$

$$= F ((\lambda x. F(xx)) (\lambda x. F(xx)))$$

$$= F (Y F) \Rightarrow F (F (Y F)) \Rightarrow F (F \ldots (F (Y F)) \ldots) \Rightarrow \ldots$$

i.e., $Y (\text{fix})$ is a fixed point operator that when applied to any function $F$ applies $F$ to a copy of itself repeatedly (i.e., recursively) the “recursion” only terminates if $F$ has an internal terminating condition, i.e., $F$ must contain a conditional expression, so “if-then-else” is an important primitive in any language with recursion. See McCarthy’s paper mentioned at the end of the Lecture 02 notes.

Fix in Haskell

A fixed point operator in Haskell is a higher order function that takes a function as an argument and recurses on that function

$$\text{fix } f = f (\text{fix } f) \text{ or fix } f = \text{let } x = f \ x \ \text{in } x$$

If we flip this equation around, we get:

$$f (\text{fix } f) = \text{fix } f$$

So “fix $f$” is a fixed point for any higher order function $f$ since $f (\text{fix } f)$ returns fix $f$ as a result.
Using Fix in Haskell

zeros = fix (\f -> 0:f)

gcd :: Integral a => a -> a -> a
    gcd = fix (\f x y -> case (abs x, abs y) of
        (x,0) -> x
        (x,y) -> f y (x `rem` y))

fact :: Integer->Integer
    fact = fix (\f n -> if n == 0 then 1 else n * f (n-1))

fibs :: [Integer]
    fibs = fix (\xs -> 0 : scanl (+) 1 xs)

len :: [a] -> Int
    len = fix (\f xs -> case xs of
        [] -> 0
        (x:xs) -> 1 + (f xs))

map :: (a->b) -> [a] -> [b]
    map = fix (\g xs -> case xs of
        [] -> []
        (x:xs) -> g x : f g xs)

foldl :: (a -> b -> a) -> a -> [b] -> a
    foldl = fix (\f g z xs -> case xs of
        [] -> z
        (x:xs) -> f g (g z x) xs

Monday, April 27, 2009
Fix is Lazy

Fix only works because of lazy evaluation. If Haskell had eager evaluation, \( f (\text{fix } f) \) would never terminate:

\[
 f (\text{fix } f) \Rightarrow f f (\text{fix } f) \Rightarrow f f f (\text{fix } f) \Rightarrow \ldots
\]

\[
 \text{fact} = \text{fix } (\lambda f \ n \rightarrow \text{if } n=0 \text{ then } 1 \text{ else } n*f(n-1))
\]

\[
 \text{fact } 3 \Rightarrow (\lambda f \ n \rightarrow \text{if } n=0 \text{ then } 1 \text{ else } n*f(n-1))(\text{fix } f) 3
\]

\[
 \Rightarrow (\text{if } 3=0 \text{ then } 1 \text{ else } 3*(\text{fix } f)(3-1))
\]

\[
 \Rightarrow (3*(\text{fix } f)(3-1))
\]

\[
 \Rightarrow 6
\]

The Monad Type Class

A Monad is any instance of the polymorphic Monad type class that implements the minimal operators: \((>>=)\) and return. Think of a Monad as a container for a value that will be used in a sequence of computations.

\[
\text{class Monad } m \text{ where}
\]

\[
\begin{align*}
\text{return} & : a \rightarrow m a \text{ -- return x: instantiate an "monad of a" (ma)} \\
(\gg=) & : m a \rightarrow (a \rightarrow m b) \rightarrow m b \text{ -- m } \gg= f: \text{xform ma to mb using } f \\
(\gg) & : m a \rightarrow m b \rightarrow m b \text{ -- p } \gg q: \text{xform ma & mb into an mb} \\
\text{fail} & : \text{String } \rightarrow m a \text{ -- construct an failure monad of a}
\end{align*}
\]

-- Minimal complete definition: \((>>=)\), return

\[
\begin{align*}
p \gg q & = p \gg (\_ \rightarrow q) \text{ -- same as } \gg=, \text{ except function is omitted} \\
\text{fail s} & = \text{error s}
\end{align*}
\]
Example: The List Monad

- We can use the list data type [] as the parameter m

```haskell
instance Monad [] where  -- m a :: [a]
  (x:xs) >>= f = f x ++ (xs >>= f) -- iterate f over the entire list
  []      >>= f = []          -- failure is just the empty list

  return x    = [x]  -- construct a singleton list with x
  fail s        = []
```

- The following are all equivalent:

  ```haskell
  xprod xs ys = xs >>= (\x -> ys >>= (\y -> return (x,y)))
  xprod xs ys = do { x <- xs; y <- ys; return (x,y) }
  xprod xs ys = do x <- xs        -- foreach x in xs
                  y <- ys        -- foreach y in ys
                  return (x,y)   -- [(x,y)]
  xprod xs ys = [(x,y) | x <- xs, y <- ys]
  ```

“Do” Notation Applied to Lists

- map f xs = [ f x | x <- xs]
  or
  map f xs = do { x <- xs; return (f x) }

- filter p xs = [ x | x <- xs, p x]
  or
  filter p xs = do { x <- xs; if (p x) then return x else [] }

- msort :: (Ord a) => [a] -> [a]
  msort [] = []
  msort (p:rest) = msort left ++ [p] ++ msort right
  where
    left  = do { x <- rest; if x <= p then return x else [] }  -- rest >>= (\x -> if x <= p then [x] else [])
    right = do { x <- rest; if x > p then return x else [] }  -- rest >>= (\x -> if x > p then [x] else [])
```

Monday, April 27, 2009
The Maybe Type

The Maybe datatype captures the concept that a computation may return "nothing" or "just" a value of type 'a'.

data Maybe a = Nothing | Just a
  deriving (Eq, Ord, Read, Show)

If you prefer, you can think of the Maybe type existentially (aka Sartre) as a computation results in "being" or "nothingness"

data Existential = Nothingness | Being a

The maybe function in the Haskell Prelude operates on the Maybe datatype

maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing = n
maybe n f (Just x) = f x

The Maybe Monad

The Maybe monad is useful when we are using other monads and a computation may return nothing or just a value and we need to return a monad as a result. We will see how this is used in Monadic Parsing

instance Monad Maybe where
  Just x >>\ k = k x
  Nothing >>= k = Nothing
  return x = Just x
  fail s = Nothing
The Matrix Monad

Define Reality in terms of the “Matrix”

```haskell
data Reality a = Matrix a | Zion a deriving (Show)

instance Monad Reality where
  Matrix x >>= jackIn = jackIn x
  Zion x >>= phoneHome = phoneHome x
  return x         = Zion x
  fail s             = Matrix s
```

Mr. Smith is a fixed point combinator. Neo is a “supercombinator” that terminates Mr. Smith :-) 

```haskell
mrSmith f = f (mrSmith f)
neo tf f = tf (mrSmith f)
```

The Either Type

The Either type is similar to the Maybe type in that it allows us to have a “choice” of two different kinds of things. A “Left” thing or a “Right” thing. Another way to think of Either is that either a computation succeeds (Left) or it fails (Right)

```haskell
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)

either :: (a -> c) -> (b -> c) -> Either a b -> c
either l r (Left x)  = l x
either l r (Right y) = r y
```
Syntactic Parsing

Syntactic Parsing is a common computational process where we process a string of characters from left-to-right and convert the characters into some other structure on which some other computation can be performed.

For example: “1+2*3” is a just a string of characters that we often want to interpret as an arithmetic expression and compute a value.

Before we can evaluate this string of characters, we need to syntactically “parse” it and convert an expression tree with binary values at the nodes and then perform a postorder evaluation to reduce the expression tree to a result value.

Parsing Terminology

In parsing, we normally ignore “whitespace” unless it has syntactic significance.

A string that is to be parsed is said to contain “tokens”. E.g., “1+2*3” has 5 tokens:
1 is a constant numeral token
+ is a binary operator token for addition
2 is a constant numeral token
* is a binary operator token for multiplication
3 is a constant numeral token

A token may consist of more than one character. E.g., “This is a sequence of tokens” has 6 tokens all separate by whitespace
In Haskell, we define a polymorphic parser type and a set of "building block" parsing functions that can be combined/sequenced to effect the parsing of a arithmetic expression, a program, or something else.

The Parser type needs to be a function type, since a parser is a function that maps a String to some other type, e.g.,

\[
\text{type Parser = String \to Tree}
\]

To be more general we want to be able to parse a String into any type \(a\), not just a Tree structure. We also want to have the flexibility to return a list of results of the parse, not just one:

\[
\text{type Parser a = String \to [(a, String)]}
\]

Parsing Functions

We can write a generic parse function that given a specific parser function, it applies that function to the input:

\[
\text{parse :: Parser a \to String \to [(a, String)]}
\]

\[
\text{parse pf input = pf input}
\]

We then need a set of basic parsing functions:

\[
\text{return :: a \to Parser}
\]

\[
\text{return v = \inp \to [(v, \inp)]}
\]

\[
\text{failure :: Parser a}
\]

\[
\text{failure = \inp \to []}
\]
Item Parsing Function

Given an input string, we can write a simple “item” parser that parses one character at a time from the string.

```haskell
item :: Parser Char
item = \inp -> case inp of
    [] -> [] -- empty string
    (x:xs) -> [(x,xs)]

parse item "this is a string" => [('t','his a string')]
```