Lecture 26

Lectured by Prof. Caldwell and scribed by Sunil Kothari

November 25, 2008

1 Review

We will start from the beginning. The goal is to apply parsers to a string and that will generate tree in the end.

The book defines parsers as functions from $String \to Tree$.
Then it is generalized to $String \to (Tree, String)$.
Then the author says that since a string can parsed in multiple ways so let's account for those. The resultant type of parsers is $[(String \to [tree, String)]$.
The one nice feature is that empty list will indicate failure. Finally, the type of parser is $Parser\ \alpha = String \to [(\alpha, String)]$.

We hope to use sequencing operations from the Monad.

These are 3 different ways of defining a data type.
This is something new.

$$newType\ Parser\ a = MkP(String \to [(a, String)])$$

The only difference here is that there is another version

$$type\ String = [Char]$$

Note that $String$ is a synonym of $[Char]$.

What happens in the data declaration ?

$$data\ Parser\ a = MkP(String \to [(a, String)])$$

Typically, *data* is used for inductive data types so that you can do case and pattern matching.

$$typeParsera = String \to [(a, String)]$$

Putting a constructor (MkP here) tells the system to recognize the type as a Parser. It's more or less an efficiency thing . Note that there's no recursion in newType thing.

The newtype thing is not really there in the compiler.

Anyways we are adopting the newtype thing for Parser.

```
newtype Parser a = MkP (String -> [(a,String)])
```

apply is given as:

```
apply:: Parser a -> String -> [(a,String)]
apply (MkP f) i = f i
```

applyParser is given as:

```
applyParser :: Parser a -> String -> a
applyParser p = fst.head.apply p
```

Remember that (>>=) the type of bind operator is:

```
Hugs> :t (>>=)
(>>=) :: Monad a => a b -> (b -> a c) -> a c
Hugs>
```

Now, we can define an instance of Monad for Parser as given in Bird's book:

```
instance Monad Parser
   -- (>>=) :: Parser a -> ( a -> Parser b) -> Parser b
  p >>= q = MkP f
     where f s = [(y,s'')|(x,s') <- apply p s, (y,s'') <- apply (q x) s']

   -- return :: a -> Parser a
  return v = MkP( i -> [(v,i)])
```

Note that list comprehension is like do notation for lists.
Interestingly, Hutton's book defines the bind operator as follows:

```
 instance Monad where
  -- (* Hutton's thing*)
   p >>= f MkP (\i -> case (apply p i) of
                        [] -> []
                        [(v,out)] -> apply (f v) out

  -- out is the remaining part of the input string
```

Next, we look at *item*, which parses a string character by character

```
 item:: Parser Char
 item = MkP(ı -> case i of
                   [] -> []
                   (x:xs) -> [(x,xs)])
```

Here's an example of a parser which reads first two characters and returns the remainder of the string

```
Parser> apply (do {x <-item; y <- item; return (y,x)}) "xyzzy"
[(('y','x'),"zzy")] :: [((Char,Char),String)]

Parser> apply (do {x <-item; y <- item; return (y,x)}) "x"
[] :: [((Char,Char),String)]

Parser> apply (do {x <-item; y <- item; return (y,x)}) "xyzzzy"
[(('y','x'),"zzzy")] :: [((Char,Char),String)]
Parser>
```

zero is a parser that always fails – returns an empty list.

```
 zero :: Parser a
 zero = MkP(\i -> [])

 Parser> apply zero "xyzzy"
[] :: [(a,String)]
```

Next, we define a parser which reads the first three characters of a string and returns a pair of the first and the third character, and also returns the remaining string.

```
 p :: Parser (Char,Char)

 p =  do x <- item
         item
         y <- item
         return (x,y)

Parser> apply p "xyzzy"
[(('x','z'),"zy")] :: [((Char,Char),String)]
```

This business of instantiating monads and figuring out is a fascinating work of functional programming people. The downside is that it is complicated. But it gets easy when you use it.

As of now, we have simple parsers. Now we make a parser which uses a predicate

```
sat::(Char -> Bool) -> Parser Char
sat p = do x <- item
           if p x then return x else zero
```

Here’s some examples:

```
Parser> apply (sat (==’x’)) "wxyzzy"
[] :: [(Char,String)]
Parser> apply (sat (==’x’)) "xyzzy"
[(’x’,"yzzy")] :: [(Char,String)]
```

This is interesting – the type contains a function *flip*, which is normally not the case.

```
:t (==’x’)
flip (==) ’x’ : Char -> Bool
```

We can do more

```
Parser> apply (sat (¸-> c ‘elem‘ "xyzzy")) "wxyzzy"
[] :: [(Char,String)]

Parser> apply (sat (¸-> c ‘elem‘ "xyzzy")) "xyzzy"
[(’x’,"yzzy")] :: [(Char,String)]

Parser> apply (sat (¸-> c ‘elem‘ "xy")) "xyzzy"
[(’x’,"yzzy")] :: [(Char,String)]
```

Then we have a parser which parses only digits and another which parses returns the character corresponding a particular digit.

```
  digit  :: Parser Char
  digit  = sat isDigit

  digit’ :: Parser Int
  digit’ = do  d <- digit; return (ord d - ord ’0’)

Parser> apply digit "1234"
[(’1’,"234")] :: [(Char,String)]

Parser> apply digit "abcd"
[] :: [(Char,String)]
```

```
Parser> apply digit' "abcd"
[] :: [(Int,String)]

Parser> apply digit' "1234"
[(1,"234")] :: [(Int,String)]
Parser>
```

More sophisticated parsers can now be defined

```
 lower  :: Parser Char
 lower  =  sat isLower
```

```
Parser> apply lowers "sUpper"
[("s","Upper")] :: [([Char],String)]
Parser> apply lowers "ssUpper"
[("ss","Upper")] :: [([Char],String)]
```

```
 upper  :: Parser Char
 upper  = sat isUpper

 char      :: Char -> Parser Char
 char x = sat (==x)
```

sat (!==) is a parser which matches the first character with !.

```
string []  = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)
```

If it's an actual string - eat a character x and then recursively call string on xs

```
Parser> apply (string "xy")  "xyzzy"
[("xy","zzy")] :: [([Char],String)]

Parser> apply (string "wxy")  "xyzzy"
[] :: [([Char],String)]
Parser>
```

The two parsers can also be combined using the +++ (choice) operator.

```
 p `plus` q = Mkp f
     where fx = apply p s ++ apply q s
```

Hutton's choice operator

```
p +++ q  = MkP (\i -> case apply p i of
                          [] -> apply q i
                          m -> m )
```

Hutton has simplified it so as to have deterministic parsers.

```
Parser> apply (char 'x' +++ char 'y') "xyzzy"
[('x',"yzzy")] :: [(Char,String)]

Parser> apply (char 'x' +++ char 'y') "yxzzy"
[('y',"xzzy")] :: [(Char,String)]

Parser> apply (char 'x' `plus` char 'y') "yxzzy"
[('y',"xzzy")] :: [(Char,String)]
```

Also, consider *lowers*:

```
lowers :: Parser String
lowers = do {c <- lower; cs <- lowers; return (c : cs)} +++ return ""
```

Here's some examples:

```
Parser> apply lowers "xyzzy"
[("xyzzy","")] :: [([Char],String)]

Parser> apply lowers "xyzzyABCD"
[("xyzzy","ABCD")] :: [([Char],String)]

Parser> apply lowers "xXyzzyABCD"
[("x","XyzzyABCD")] :: [([Char],String)]
```

Let's look futher down. Read is a type class.

```
Parser> :t read
read :: Read a => String -> a

Parser> read "123" :: Int
123 :: Int
```

digit is defined as

```
digit  :: Parser Char
digit  = sat isDigit
```

The *nat* is a parser that reads one or many digits.

```
nat :: Parser Int
nat = do xs <- many1 digit
         return (read xs)


Parser> apply nat "123"
[(123,"")] :: [(Int,String)]
Parser> apply nat "123abc"
[(123,"abc")] :: [(Int,String)]
```

*many*is a parser to read zero or more times.

```
:t many lower
many lower :: Parser [Char]
```

A standard thing is to tokenize *i.e.* eat as many space as possible.

```
space :: Parser ()
space = do many (char ' ')
           return ()
```

Here's an example.

```
Parser> apply space "     xyzzy"
[((),"xyzzy")] :: [((),String)]
```

The following parser parses a list of natural numbers (including any spaces).

```
 natural :: Parser Int
 natural = token nat

 symbol :: String -> Parser String
 symbol xs = token (string xs)

  natlist :: Parser [Int]
  natlist = do symbol "["
               n <- natural
               ns <- many (do symbol ","
                                natural)
               symbol "]"
               return (n:ns)
```

```
Parser> apply natlist "[1,   2,3,                5]"
[([1,2,3,5],"")] :: [([Int],String)]
Parser> apply natlist "[ ]"
[] :: [([Int],String)]
Parser> apply natlist "[1,   2,3,                5]zbcd "
[([1,2,3,5],"zbcd ")] :: [([Int],String)]
Parser>
```