Combinator Parsing Activity

Mattox Beckman

Manager	Keeps team on track	
Recorder	Records decisions	
Reporter	Reports to Class	

Purpose

Monadic combinator parsers work very similarly to the LL parsers we covered before, but the monadic interface manages the input stream for us. The resulting parsers are much easier to read and to write. Your goals are:

- Understand the types of the parser combinators.
- Explain the result of executing a parser.
- Explain the <|> combinator.
- Implement `many` and `many1`.

Part 1 — The Types

```haskell
newtype Parser t = Parser (String -> [(t,String)])
run (Parser p) = p

oneOf xx =
  Parser (\inp -> case inp of
                  (s:ss) | s `elem` xx -> [(s,ss)]
                  otherwise            -> [])

sat pred =
  Parser (\inp -> case inp of
                  (s:ss) | pred s    -> [(s,ss)]
                  otherwise          -> [])

p1 = run (oneOf "abc")  "axy"
p2 = run (oneOf "abc")  "xya"
```

The `newtype` is like `data`, but the resulting type has only one constructor, and it is optimized away by the compiler. We use it instead of simply saying `type Parser t = String -> [(t,String)]` because we can't declare a `type` as an instance, but we can declare a `newtype` as an instance.

Problem 1) What will be the values of `p1` and `p2`?

Problem 2) Can you write the function `digit` that parses a digit? Use `sat` to do this. For more of a challenge, have it return an actual integer.

Part 2 — The Type Classes

```
instance Functor Parser where
  fmap f (Parser  p1) =
      Parser (\inp -> [(f t, s) |
                       (t,s) <- p1 inp])

instance Applicative Parser where
  pure a = Parser (\inp -> [(a,inp)])
  (Parser p1) <*> (Parser p2) =
      Parser (\inp -> [(v1 v2, ss2) |
                       (v1,ss1) <- p1 inp,
                       (v2,ss2) <- p2 ss1])

instance Monad Parser where
  (Parser p) >>= f =
      Parser (\inp -> concat [run (f v) inp'
                             | (v,inp') <- p inp])

data Exp = IntExp Integer
         | PlusExp Exp Exp
  deriving Show

p3 = run (IntExp <$> digit)  "123"
p4 = run (PlusExp <$> getIntExp <*> getIntExp)  "123"
p5 = do i1 <- getIntExp
        i2 <- getIntExp
        return (IntExp i1 i2)
```

Problem 3) What is the value of `p3`? Trace through the evaluation and be sure everyone on your team understands how we got that result.

Problem 4) Write the function `getIntExp` that is like `digit` but encapsulates the digit in an `IntExp`.

Problem 5) What is the value of `p4`? Trace through the evaluation and be sure everyone on your team understands how we got that result.

Problem 6) What is the value of `p5`? Trace through the evaluation and be sure everyone on your team understands how we got that result.

Part 3 — Choice, Many, Many1

```haskell
(Parser p1) <|> (Parser p2) =
   Parser (\inp -> take 1 $ p1 inp ++ p2 inp)

string [] = Parser (\inp -> [([],inp)])
string (s:ss) = do v <- char s
                   vv <- string ss
                   return $ v:vv

getPlusExp = do string "+"
                e1 <- getExp
                e2 <- getExp
                return (PlusExp e1 e2)

getExp = getIntExp
     <|> getPlusExp
```

Problem 7) Examine the code for `<|>`. How does it work? Hint: consider the cases that `p1` succeeds, `p1` fails but `p2` succeeds, and both `p1` and `p2` fail.

Problem 8) Write the parsers `many p` and `many1 p` that take zero or more (for `many`) or one or more (for `many1`) repetitions of `p`.

Problem 9) The way we suggested writing `getIntExp` only works for a single digit. Can you make it work for multi-digit integers now?

Part 4 — Precedence

Problem 10) Now add `TimesExp` to the mix. Stratify the grammar so that `TimesExp` has higher precedence than `PlusExp`.