CSCE 314
Programming Languages

Monadic Parsing

Dr. Hyunyoung Lee
What is a Parser?

A parser is a program that takes a string of characters (or a set of tokens) as input and determines its syntactic structure.

```
2*3+4  means  2 * 3 + 4
```

```
+   4
    / 
   /   
  2    3
   `/   `
    *   *
```

String or [Token] --> Parser --> syntactic structure
The Parser Type

In a functional language such as Haskell, parsers can naturally be viewed as functions.

newtype Parser = P (String -> Tree)

A parser is a function that takes a string and returns some form of tree.

However, a parser might not require all of its input string, so we also return any unused input:

newtype Parser = P (String -> (Tree, String))
A string might be parsable in many ways, including none, so we generalize to a list of results:

```
newtype Parser a = P (String → [(Tree, String)])
```

The empty list denotes failure, a singleton list denotes success.

Furthermore, a parser might not always produce a tree, so we generalize to a value of any type:

```
newtype Parser a = P (String → [(a, String)])
```

Note: For simplicity, we will only consider parsers that either fail and return the empty list as results, or succeed and return a singleton list.
Basic Parsers (Building Blocks)

We need a function that applies a Parser to an input string:

```
parse :: Parser a -> String -> [(a, String)]
p parse (P p) inp = p inp -- apply parser p to inp
```

```
item :: Parser Char
     -- P (String -> [(Char, String)])
item = P (\inp -> case inp of
            [] -> []
            (x:xs) -> [(x,xs)])
```

The parser `item` fails if the input is empty, and consumes the first character otherwise.

Example:
```
> parse item "Howdy all"
[('H', "owdy all")]
> parse item ""
[]
```
item = P (\inp -> case inp of
              [] -> []
              (x:xs) -> [(x, xs)])

Quiz: What is the output of the following expression?

> parse item "a"

> parse item "ab"
Sequencing Parsers

Often, we need to combine parsers in sequence, e.g., the following grammar:
\[
\text{<if-stmt> :: if (<expr>) then <stmt>}
\]

first parse if, then (, then <expr>, then ), ...

To combine parsers in sequence, we make the Parser type into a monad:

\[
\text{instance Monad Parser where}
\]
\[
-- (>>=) :: Parser a \rightarrow (a \rightarrow Parser b) \rightarrow Parser b
\]
\[
p >>= f = \text{P} (\lambda inp \rightarrow \text{case parse p inp of}
\]
\[
[\{} \rightarrow \{\}
\]
\[
[(v,\text{out})] \rightarrow \text{parse (f v) out}
\]

The “Monadic” Way

A sequence of parsers can be combined as a single composite parser

\[
(p >>= f) = \text{Parse} a \rightarrow (a \rightarrow \text{Parse} b) \rightarrow \text{Parser} b
\]

\[
p >>= f = \text{Parse} \lambda \text{inp} \rightarrow \text{case parse} p \text{ inp of}
\]

\[
[\_] \rightarrow [\_]
\]

\[
[(v, \text{out})] \rightarrow \text{parse} (f v) \text{ out}
\]

\[
p >>= f
\]

- fails if \( p \) fails
- otherwise applies \( f \) to the result of \( p \)
- this results in a new parser, which is then applied

Example

> parse (item >>= (\_ -> item)) "abc"

[('b','c')]
Sequencing Parsers (using do)

A sequence of parsers can be combined as a single composite parser using the keyword do.

Example:

```haskell
three :: Parser (Char,Char)
three = do x <- item
          item
          z <- item
          return (x,z)
```

Meaning: “The value of x is generated by the item parser.”

The parser `return v` always succeeds, returning the value v without consuming any input:

```haskell
return :: a -> Parser a
return v = P (\inp -> [(v,inp)])
```
If any parser in a sequence of parsers fails, then the sequence as a whole fails. For example:

```haskell
three :: Parser (Char, Char)
three = do x <- item
           item
           z <- item
           return (x, z)
```

> parse three "abcdef"
[[(‘a’, ‘c’), "def"]]

> parse three "ab"
[]
\[ \gg= \text{or do} \]

**Using \[\gg=\]**

\[
\begin{align*}
p1 & \gg= v1 -> \\
p2 & \gg= _ -> \\
P3 & \gg= v3 -> \\
& \ldots \\
pn & \gg= vn -> \\
\text{return } (f v1 v3 \ldots vn)
\end{align*}
\]

**Using do notation**

\[
\begin{align*}
do v1 & \leftarrow p1 \\
p2 & \\
v3 & \leftarrow p3 \\
& \ldots \\
vn & \leftarrow pn \\
\text{return } (f v1 v3 \ldots vn)
\end{align*}
\]

If some \(v_i\) is not needed, \(v_i \leftarrow p_i\) can be written as \(p_i\), which corresponds to \(p_i \gg= _ -> \ldots\).
Example

Using >>==

```javascript
rev3 =
  item >>= \v1 ->
  item >>= \v2 ->
  item >>= \_  ->
  item >>= \v3 ->
return $
  reverse (v1:v2:v3:[])
```

Using do notation

```javascript
rev3 =
  do v1 <- item
  v2 <- item
  item
  v3 <- item
return $
  reverse (v1:v2:v3:[])
```

> parse rev3 “abcdef”
[(“dba”,“ef”)]
> parse (rev3 >>= (\_  -> item)) “abcde”
[(‘e’,”“)]
> parse (rev3 >>= (\_  -> item)) “abcd”
[]
Key benefit: The result of first parse is available for the subsequent parsers

```
parse (item >>= (\x ->
    item >>= (\y ->
        return (y:[x]))) ) "ab"
```

```
[("ba","" )]
```

Quiz: write the above definition using do.
Making Choices

What if we have to backtrack? First try to parse p, then q? The parser \( p \mid\mid q \) behaves as the parser p if it succeeds, and as the parser q otherwise.

empty :: Parser a
empty = P (\inp -> []) -- always fails

(\mid\mid) :: Parser a -> Parser a -> Parser a
p \mid\mid q = P (\inp -> case parse p inp of
  [] -> parse q inp
  [(v,out)] -> [(v,out)])

Example:

> parse empty "abc"
[]

> parse (item \mid\mid return 'd') "abc"
[(a,'bc')]
Examples

> parse item ""
[]

> parse item "abc"
[('a','bc')]

> parse empty "abc"
[]

> parse (return 1) "abc"
[(1,"abc")]

> parse (item <|> return 'd') "abc"
[('a','bc')]

> parse (empty <|> return 'd') "abc"
[('d','abc')]

> parse ((empty <|> item) >>= (\_ -> item)) "abc"
[('b','c')]
Derived Primitives

Parsing a character that satisfies a predicate:

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

Examples

> parse (sat (==’a’)) “abc”
[('a','bc')]
> parse (sat (==’b’)) “abc”
[]
> parse (sat isLower) “abc”
[('a','bc')]
> parse (sat isUpper) “abc”
[]
Derived Parsers from sat

digit, letter, alphanum :: Parser Char
digit = sat isDigit
letter = sat isAlpha
alphanum = sat isAlphaNum

lower, upper :: Parser Char
lower = sat isLower
upper = sat isUpper

char :: Char -> Parser Char
char x = sat (== x)
To accept a particular string

Use sequencing recursively:

```
string :: String -> Parser String
string [] = return []
string (x:xs) = do char x
  string xs
  return (x:xs)
```

Entire parse fails if any of the recursive calls fail

```
> parse (string "if ["] "if (a<b) return;"
[]
> parse (string "if (") "if (a<b) return;
["if (","a<b) return;">]
```
many applies the same parser many times

\[
\begin{align*}
\text{many} & : \text{Parser } a \rightarrow \text{Parser } [a] \\
\text{many } p & = \text{some } p \triangleright \triangleright \text{return } [] \\
\text{some} & : \text{Parser } a \rightarrow \text{Parser } [a] \\
\text{some } p & = \text{do } v \leftarrow p \\
& \quad \quad \text{vs } \leftarrow \text{many } p \\
& \quad \quad \text{return } [v:vs]
\end{align*}
\]

Examples

\[
\begin{align*}
> \text{parse (many digit) "123ab"} \\
& \quad [("123","ab")]
> \text{parse (many digit) "ab123ab"} \\
& \quad [('',"ab123ab")]
> \text{parse (many alphanum) "ab123ab"} \\
& \quad [("ab123ab",'')]
\end{align*}
\]
Example

We can now define a parser that consumes a list of one or more digits of correct format from a string:

```haskell
p :: Parser String
p  = do char '['
    d  <- digit
    ds <- many (do char ','
                  digit)
    char ']'
    return (d:ds)
```

```
> parse p "[1,2,3,4]"
["1234","]"
> parse p "[1,2,3,4"
[]
```

Note: More sophisticated parsing libraries can indicate and/or recover from errors in the input string.
Example: Parsing a token

```haskell
space :: Parser ()
space = do many (sat isSpace)
         return ()

token :: Parser a -> Parser a
token p = do space
         v <- p
         space
         return v

identifier :: Parser String
identifier = token ident

ident :: Parser String
ident = do x <- sat isLower
         xs <- many (sat isAlphaNum)
         return (x:xs)
```