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 4 + 3*2
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:

```haskell
newtype Parser = 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:

```haskell
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)]
parse (P p) inp = p inp -- apply parser p to inp
```

Example:

```
> parse item "Howdy all"
[("H","owdy all")]
> parse item ""
[]
```

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

Often, we need to combine parsers in sequence, e.g., the following grammar:

```
<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:

```haskell
instance Monad Parser where
    -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    p >>= f = P \inp -> case parse p inp of
        []     -> []
        [(v,out)] -> parse (f v) out
```
Sequencing Parser (do)

Now 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 = \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
           y <- item
           return (x, y)
```

```haskell
> parse three "abcdef"
[[(‘a’,’c’),"def"]]
> parse three "ab"
[]
```
Making Choices

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

```
empty :: Parser a
empty = \inp -> [] -- always fails

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

Example:

```
> parse empty "abc"
[]

> parse (item \lor return 'd') "abc"
[("a","bc")]
```
The “Monadic” Way

Parser sequencing operator

\[
(\gg\gg)=) \quad :: \quad \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b \\
p \gg\gg f = \lambda \text{inp} \rightarrow \text{case } \text{parse } p \text{ inp of} \\
\quad \quad [] \rightarrow [] \\
\quad \quad [(v, out)] \rightarrow \text{parse } (f \ v) \text{ out}
\]

\[p \gg\gg f\]

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

Example

\[> \text{parse } ((\text{empty } <\mid \text{ item}) \gg\gg (\_ \rightarrow \text{item})) \text{ "abc"} \\
\quad [(\text{'b',"c"})]\]
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')]
\[ \text{or do} \]

Using \[ \text{\texttt{>>=}} \]

\[
\begin{align*}
\text{p1} & \Rightarrow \text{\texttt{\textbackslash v1}} \rightarrow \\
\text{p2} & \Rightarrow \text{\texttt{\textbackslash v2}} \rightarrow \\
& \quad \ldots \\
\text{pn} & \Rightarrow \text{\texttt{\textbackslash vn}} \rightarrow \\
\text{return (f v1 v2 \ldots vn)}
\end{align*}
\]

Using do notation

\[
\begin{align*}
\text{do } v1 & \leftarrow \text{p1} \\
& \quad v2 \leftarrow \text{p2} \\
& \quad \ldots \\
& \quad vn \leftarrow \text{pn} \\
\text{return (f v1 v2 \ldots vn)}
\end{align*}
\]

If some \( \text{v}_i \) is not needed, \( \text{v}_i \leftarrow \text{p}_i \) can be written as \( \text{p}_i \), which corresponds to \( \text{p}_i \Rightarrow \text{\texttt{\textbackslash _}} \rightarrow \ldots \).
Example

Using $\gg=$

```haskell
rev3 =
  item $\gg=$ \v1 ->
  item $\gg=$ \v2 ->
  item $\gg=$ \_  ->
  item $\gg=$ \v3 ->
  return $
    \text{reverse (v1:v2:v3:[[])}}$
```

Using `do` notation

```haskell
rev3 =
  do v1 <- item
     v2 <- item
     item
     v3 <- item
     return $
       \text{reverse (v1:v2:v3:[[])}}$
```

> rev3 "abcdef"
[[("dba","ef")]]

> (rev3 $\gg=$ (\_  -> item)) "abcde"
[[("e","")]]

> (rev3 $\gg=$ (\_  -> item)) "abcd"
[]
Key benefit: The result of first parse is available for the subsequent parsers

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

[("ba","")]
Derived Primitives

Parsing a character that satisfies a predicate:

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

Examples

```haskell
> 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:

```haskell
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

```haskell
> 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

**Examples**

```haskell
> parse (many digit) "123ab" [("123","ab")]
> parse (many digit) "ab123ab" [("","ab123ab")]
> parse (many alphanum) "ab123ab" [("ab123ab",""')]
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

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)