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.

```haskell
type Parser = String -> Tree
```

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

```haskell
type Parser = String -> (Tree, String)
```

A string might be parsable in many ways, including none, so we generalize to a list of results:

```haskell
type Parser = 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
type Parser a = String → [(a, String)]
```

Finally, a parser might take token streams instead of character streams:

```haskell
type TokenParser b a = [b] → [(a, [b])]`
```

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)

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

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

Example:

```haskell
> item "Howdy all"
[("H","owdy all")]
> item ""
[]
```
We can make it more explicit by letting the function `parse` apply a parser to a string:

```haskell
parse :: Parser a -> String -> [(a,String)]
parsing p inp = p inp -- essentially id function
```

Example:

```haskell
> parse item "Howdy all"
[('H','owdy all')]
```
Sequencing Parser

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

\[
<\text{if-stmt}> ::= \text{if } (<\text{expr}> \text{) then } <\text{stmt}>
\]

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 = \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 <|> q behaves as the parser p if it succeeds, and as the parser q otherwise.

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

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

Example:

```haskell
> parse empty "abc"
[]
> parse (item <|> return 'd') "abc"
[('a',"bc")]
```
The "Monadic" Way

Parser sequencing operator

\[
(\gg\gg) \::= \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b
\]

\[
\text{p } \gg\gg \text{ f } = \\text{\inp } \rightarrow \text{case parse } \text{p } \text{inp } \text{of}
\]

\[
\text{[]} \rightarrow \text{[]}
\]

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

\[
p \gg\gg \text{ f}
\]

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

Example

\[
> \text{parse ((empty } <\mid> \text{ item) } \gg\gg (\text{\_ } \rightarrow \text{ item})) \text{ "abc"}
\]

\[
\text{[('b','c')]}\]
Examples

> parse item ""
[
]
> parse item "abc"
[['a','bc']]
> Parse empty "abc"
[
]
> parse (return 1) "abc"
[(1,"abc")]
> parse (item <|> return 'd') "abc"
[['a','bc']]
$\gg\gg=$ or do

Using $\gg\gg=$

\[
p1 \gg\gg= \\_v1 \rightarrow \\
p2 \gg\gg= \_v2 \rightarrow \\
. . . \\
pn \gg\gg= \_vn \rightarrow \\
return (f \ v1 \ v2 \ . . . \ vn)
\]

Using do notation

\[
do \ v1 \leftarrow p1 \\
v2 \leftarrow p2 \\
. . . \\
vn \leftarrow pn \\
return (f \ v1 \ v2 \ . . . \ vn)
\]

If some $v_i$ is not needed, $v_i \leftarrow p_i$ can be written as $p_i$, which corresponds to $p_i \gg\gg= \_ \rightarrow \ldots$
Example

Using >>==

\[
\begin{align*}
\text{rev3} &= \text{item} >>= \backslash \text{v1} \rightarrow \\
& \quad \text{item} >>= \backslash \text{v2} \rightarrow \\
& \quad \text{item} >>= \_ \rightarrow \\
& \quad \text{item} >>= \backslash \text{v3} \rightarrow \\
& \quad \text{return } \$
\end{align*}
\]
\[
\text{reverse } (\text{v1}:\text{v2}:\text{v3}:[])
\]

\[
> \text{rev3} \text{ "abcdef" ["db","ef"]}
\]

Using do notation

\[
\begin{align*}
\text{rev3} &= \text{do } \text{v1} \leftarrow \text{item} \\
& \quad \text{v2} \leftarrow \text{item} \\
& \quad \text{item} \\
& \quad \text{v3} \leftarrow \text{item} \\
& \quad \text{return } \$
\end{align*}
\]
\[
\text{reverse } (\text{v1}:\text{v2}:\text{v3}:[])
\]

\[
> (\text{rev3} >>= (\_ \rightarrow \text{item})) \text{ "abcde" } [\text{("e",")}] \\
> (\text{rev3} >>= (\_ \rightarrow \text{item})) \text{ "abcd" } []
\]
Key benefit: The result of first parse is available for the subsequent parsers

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

```
many :: Parser a -> Parser [a]
many p = some p <|> return []
some :: Parser a -> Parser [a]
some p = do v <- p
         vs <- many p
         return (v:vs)
```

Examples

```
> 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)
```

```haskell
> 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)
```