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.

\[2 \times 3 + 4\] means:

```
(2 \times (3 + 4))
```
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.
We need a function that applies a Parser to an input string:

\[
\text{parse} :: \text{Parser } a \rightarrow \text{String} \rightarrow [(a, \text{String})]
\]

\[
\text{parse } (P \ p) \ \text{inp} = p \ \text{inp} -- \text{apply parser } p \text{ to inp}
\]

**Example:**

\[
> \text{parse item } \text{"Howdy all"}\n\]

\[
[(\text{`H'}, \ \text{"owdy all"})]\n\]

\[
> \text{parse item } \text{""}\n\]

\[
[]\n\]
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:

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

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 )
```
The “Monadic” Way

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

\[(>>=) :: \text{Parser } a \to (a \to \text{Parser } b) \to \text{Parser } b\]

\[p >>= f = \text{P } (\inp \to \text{case parse } p \inp \text{ of}
  \[\] \to []
  [(v, out)] \to \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

\[> \text{parse } (\text{item >>= (\_ } \to \text{ item})) \text{ "abc" }\]
\[[('b','c')]\]
Sequencing Parsers (do)

Now a sequence of parsers can be combined as a single composite parser using the keyword \texttt{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.”

```haskell
> parse three "abcd"
[(['a','c'],"d")]
```

The parser \texttt{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:

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

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

> parse three "ab"
  []
\textbf{>>= or \textit{do}}

\textbf{Using \textit{>>=}}

\begin{align*}
\text{p1} & \triangleright= \ \lambda v_1 \rightarrow \\
\text{p2} & \triangleright= \ \lambda v_2 \rightarrow \\
\ldots \\
\text{pn} & \triangleright= \ \lambda v_n \rightarrow \\
\text{return (f v_1 v_2 \ldots v_n)}
\end{align*}

\textbf{Using \textit{do} notation}

\begin{align*}
\text{do } v_1 & \leftarrow \text{p1} \\
\text{v2} & \leftarrow \text{p2} \\
\ldots \\
\text{vn} & \leftarrow \text{pn} \\
\text{return (f v_1 v_2 \ldots v_n)}
\end{align*}

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

Using $\gg=$

\[
\text{rev3} = \\
\text{item } \gg= \ \_ \rightarrow \\
\text{item } \gg= \ \_ \rightarrow \\
\text{item } \gg= \ \_ \rightarrow \\
\text{return } \$
\]

\[
\text{reverse (v1:v2:v3:[])}
\]

Using \text{do} notation

\[
\text{rev3} = \\
\text{do } v1 \leftarrow \text{item} \\
v2 \leftarrow \text{item} \\
\text{item} \\
v3 \leftarrow \text{item} \\
\text{return } \$
\]

\[
\text{reverse (v1:v2:v3:[])}
\]

\[
> \text{parse rev3 “abcdef”} \\
[(“dba”,“ef”)]
\]

\[
> \text{parse (rev3 } \gg= (\_ \rightarrow \text{item})) “abcde” \\
[(‘e’,“”)]
\]

\[
> \text{parse (rev3 } \gg= (\_ \rightarrow \text{item})) “abcd” \\
[]
\]
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 <|> q behaves as the parser p if it succeeds, and as the parser q otherwise.

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

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

Example:
> parse empty "abc"
[]
> parse (item <|> return ‘d’) "abc"
[(‘a’,"bc")]

Example:
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:

```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:

\[
\text{string :: String} \to \text{Parser String} \\
\text{string []} = \text{return []} \\
\text{string (x:xs)} = \text{do char x} \\
\quad \text{string xs} \\
\quad \text{return (x:xs)}
\]

Entire parse fails if any of the recursive calls fail

> \text{parse (string "if [")} "if (a<b) return;" []
> \text{parse (string "if ")} "if (a<b) return;" ["if (","a<b) return;")]
many applies

the same

parser many times

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

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

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)