CSCE 314
Programming Languages
Functional Parsers

Dr. Hyunyoung Lee
What is a Parser?

A parser is a program that takes a text (set of tokens) and determines its **syntactic structure**.

\[ 2 \times 3 + 4 \] means

```
  2
 / \
*   3
 / \
4   
```
The Parser Type

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

```
type Parser = 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:

```
type Parser = String ➔ (Tree, String)
```

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

```
type Parser = String ➔ [(Tree, String)]
```
Furthermore, a parser might not always produce a tree, so we generalize to a value of any type:

\[
\text{type } \text{Parser } a = \text{String } \rightarrow [(a,\text{String})]
\]

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

\[
\text{type } \text{TokenParser } b~a = [b] \rightarrow [(a,[b])]\]

Note:
For simplicity, we will only consider parsers that either fail and return the empty list of 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
*Main> item "parse this"
[('p','arse this')]"
The parser `return v` *always succeeds*, returning the value `v` without consuming any input:

```
return :: a -> Parser a
return v = \inp -> [(v,inp)]
```

The parser `failure` *always fails*:

```
failure :: Parser a
failure = \inp -> []
```

Example:

```
*Main> Main.return 7 "parse this"
[(7,"parse this")]
*Main> failure "parse this"
[]
```
We can make it more explicit by letting the function `parse` apply a parser to a string:

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

```
parse p inp = p inp -- essentially id function
```

Example:

```
*Main> parse item "parse this"
[('p' ,"arse this")]
```
Choice

What if we have to backtrack? First try to parse p, then q? The parser \texttt{p+++q} behaves as the parser p if it succeeds, and as the parser q otherwise.

\begin{verbatim}
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = \inp -> case p inp of
    []        -> parse q inp
    [(v,out)] -> [(v,out)]
\end{verbatim}

Example:

*Main> parse failure "abc"
[]
*Main> parse (failure +++ item) "abc"
[('a','bc')]
Examples

> parse item ""
[]
> parse item "abc"
[('a','bc')]
> parse failure "abc"
[]
> parse (return 1) "abc"
[(1,'abc')]
> parse (item +++ return 'd') "abc"
[('a','bc')]
> parse (failure +++ return 'd') "abc"
[('d','abc')]
Note:

The library file Parsing is available on the course home page.

The Parser type is a monad, a mathematical structure that has proved useful for modeling many different kinds of computations.
Sequencing

Commonly, we want to sequence parsers, e.g., the following grammar:

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

First parse if, then (, then <expr>, ...

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

For example:

\[
p :: \text{Parser } (\text{Char}, \text{Char})
\]

\[
p = \text{do } x \leftarrow \text{item}
\]

\[
\text{item}
\]

\[
y \leftarrow \text{item}
\]

\[
\text{return } (x,y)
\]

Meaning: “The value of \(x\) is generated by the item parser.”
Note:

- Each parser must begin in precisely the same column. That is, the **layout rule** applies.

- The values returned by intermediate parsers are **discarded** by default, but if required can be named using the ← operator.

- The value returned by the **last** parser is the value returned by the sequence as a whole.
If any parser in a sequence of parsers fails, then the sequence as a whole fails. For example:

```haskell
> parse p "abcdef"
[('a', 'c'), "def"]
```

```haskell
> parse p "ab"
[]
```

The do notation is not specific to the `Parser` type, but can be used with any monadic type.
The “Monadic” Way

Parser sequencing operator

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

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

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

Example

\[> \text{parse } ((\text{failure }++\text{ item}) \gg= (\_ \rightarrow \text{item})) \text{ "abc"}
\]
\[\[['b','c']]\]
Sequencing

Typical parser structure

\[
p_1 \gg= \backslash v_1 \rightarrow \\
p_2 \gg= \backslash v_2 \rightarrow \\
\ldots \\
p_n \gg= \backslash v_n \rightarrow \\
\text{return (f v_1 v_2 \ldots v_n)}
\]

Using do notation

\[
do \\
\begin{align*}
    v_1 & \leftarrow p_1 \\
    v_2 & \leftarrow p_2 \\
    \ldots \\
    v_n & \leftarrow p_n
\end{align*}
\]

\[
\text{return (f v_1 v_2 \ldots v_n)}
\]

If some \( v_i \) is not needed, \( v_i \leftarrow p_i \) can be written as \( p_i \gg= \_ \rightarrow \ldots \).
Example

Typical parser structure

\[
\text{rev3} = \\
\text{item} \implies \text{ \textbackslash v1 \rightarrow} \\
\text{item} \implies \text{ \textbackslash v2 \rightarrow} \\
\text{item} \implies \text{ \_ \rightarrow} \\
\text{item} \implies \text{ \textbackslash v3 \rightarrow} \\
\text{return \$} \\
\quad \text{reverse (v1:v2:v3:[[])}}
\]

Using do notation

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

\[
> \text{rev3 "abcdef"} \\
[(["dba","ef")]]
\]

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

```plaintext
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 failure
```

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:

```
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

many :: Parser a -> Parser [a]
many p = many1 p +++ return []

many1 :: Parser a -> Parser [a]
many1 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 = many (sat isSpace) >>
    return ()

token :: Parser a -> Parser a
token p = space >>
    p >>= \v ->
    space >>
    return v

identifier :: Parser String
identifier = token ident

ident :: Parser String
ident = sat isLower >>= \x ->
    many (sat isAlphaNum) >>= \xs ->
    return (x:xs)
```
Arithmetic Expressions

Consider a simple form of expressions built up from single digits using the operations of addition + and multiplication *, together with parentheses.

We also assume that:

* and + associate to the right.

* has higher priority than +.
Formally, the syntax of such expressions is defined by the following context free grammar:

\[
\begin{align*}
\text{expr} & \rightarrow \text{term} \; '+' \; \text{expr} \mid \text{term} \\
\text{term} & \rightarrow \text{factor} \; '*' \; \text{term} \mid \text{factor} \\
\text{factor} & \rightarrow \text{digit} \mid '(' \; \text{expr} \; ')' \\
\text{digit} & \rightarrow '0' \mid '1' \mid \ldots \mid '9'
\end{align*}
\]
However, for reasons of efficiency, it is important to factorize the rules for $expr$ and $term$:

\[
expr \rightarrow term \ ('+' \ expr \ | \ \varepsilon) \\

term \rightarrow factor \ ('\ast' \ term \ | \ \varepsilon)
\]

Note: The symbol $\varepsilon$ denotes the empty string.
It is now easy to translate the grammar into a parser that evaluates expressions, by simply rewriting the grammar rules using the parsing primitives.

That is, we have:

```haskell
expr :: Parser Int
expr = do t ← term
    do char '+'
        e ← expr
        return (t + e)
    +++ return t
```
term :: Parser Int
term = do f ← factor
       do char '*'
           t ← term
           return (f * t)
       +++ return f

factor :: Parser Int
factor = do d ← digit
          return (digitToInt d)
          +++ do char '('
             e ← expr
             char ')' return e
Finally, if we define

```haskell
eval :: String → Int
eval xs = fst (head (parse expr xs))
```

then we try out some examples:

```haskell
> eval "2*3+4"
10
> eval "2*(3+4)"
14
> eval "2+5-"
7
> eval "+5-"
*** Exception: Prelude.head: empty list