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.

String or [Token] → Parser → syntactic structure

2*3+4 means

```
2 * 3
+ 4
```

```
+ 

* 

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:

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

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

```
type TokenParser b a = [b] → [(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:

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

Example:

```
*Main> item "parse this"
[('p','arse this')]```
The parser \texttt{return } v \textit{ always succeeds}, returning the value \( v \) without consuming any input:

\begin{verbatim}
return :: a -> Parser a
return v = \inp -> [(v, inp)]
\end{verbatim}

The parser \texttt{failure} \textit{always fails}:

\begin{verbatim}
failure :: Parser a
failure = \inp -> []
\end{verbatim}

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:

```
parse :: Parser a → String → [(a,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 \( p +++ q \) behaves as the parser p if it succeeds, and as the parser q otherwise.

\[
(++) :: \text{Parser } a \rightarrow \text{Parser } a \rightarrow \text{Parser } a
p +++ q = \inp \rightarrow \text{case } p \inp \text{ of}
\]
\[
[] \rightarrow \text{parse } q \inp
\]
\[
[(v, out)] \rightarrow [(v, out)]
\]

Example:

*MMain> parse failure "abc"
[]
*MMain> 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:

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

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

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

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

\[
p = \text{do } x \leftarrow \text{item} \\
\quad \text{item} \\
\quad y \leftarrow \text{item} \\
\quad \text{return } (x,y)
\]

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")]

> 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

\[
(\gg=) \colon \text{Parser } a \to (a \to \text{Parser } b) \to \text{Parser } b
\]

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

\[
[] \to []
\]

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

\[p \gg= f\]

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

Example

\[
> \text{parse } ((\text{failure } +++ \text{ item}) \gg= (\_ \to \text{item})) \text{ "abc"}
\]

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

Typical parser structure

\[ \begin{align*}
  p_1 & \gg= \nu_1 \rightarrow \\
  p_2 & \gg= \nu_2 \rightarrow \\
  \quad & \vdots \\
  p_n & \gg= \nu_n \rightarrow \\
  \text{return} \ (f \ \nu_1 \ \nu_2 \ \ldots \ \nu_n)
\end{align*} \]

Using do notation

\[ \begin{align*}
  \text{do} \ \nu_1 & \leftarrow p_1 \\
  & \quad \nu_2 \leftarrow p_2 \\
  & \quad \vdots \\
  & \quad \nu_n \leftarrow p_n \\
  \text{return} \ (f \ \nu_1 \ \nu_2 \ \ldots \ \nu_n)
\end{align*} \]

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

Typical parser structure

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

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

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

\[
\text{many} :: \text{Parser} \ a \rightarrow \text{Parser} \ [a] \\
\text{many} \ p = \text{many1} \ p +++ \text{return} \ [] \\
\text{many1} :: \text{Parser} \ a \rightarrow \text{Parser} \ [a] \\
\text{many1} \ p = \ do \ v <- p \\
vs <- \text{many} \ p \\
\text{return} \ (v:vs)
\]

Examples

\[
> \text{parse} \ (\text{many} \ \text{digit}) \ "123ab" \\
[\("123","ab"\)] \\
> \text{parse} \ (\text{many} \ \text{digit}) \ "ab123ab" \\
[\("",\text{"ab123ab"\})] \\
> \text{parse} \ (\text{many} \ \text{alphanum}) \ "ab123ab" \\
[\("\text{ab123ab",""\})]
\]
Example

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

\[
p :: \text{Parser String}
p = \text{do} \ \text{char} \ ']' \\
\quad d \gets \text{digit} \\
\quad ds \gets \text{many} \ (\text{do} \ \text{char} \ ',',\ 'digit) \\
\text{char} \ ']' \\
\text{return} \ (d:ds)
\]

> 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 = 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:

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

\[
\textit{expr} \rightarrow \textit{term} \ ('+\' \ \textit{expr} \ | \ \varepsilon )
\]

\[
\textit{term} \rightarrow \textit{factor} \ ('*\' \ \textit{term} \ | \ \varepsilon )
\]

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
```
factor :: Parser Int
factor = do d ← digit
            return (digitToInt d)
+++ do char '('
            e ← expr
            char ')
            return e

term :: Parser Int
term = do f ← factor
        do char '*'
            t ← term
            return (f * t)
+++ return f
Finally, if we define

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

then we try out some examples:

```
> eval "2*3+4"
10

> eval "2*(3+4)"
14

> eval "2+5-"
7

> eval "+5-"
*** Exception: Prelude.head: empty list
```