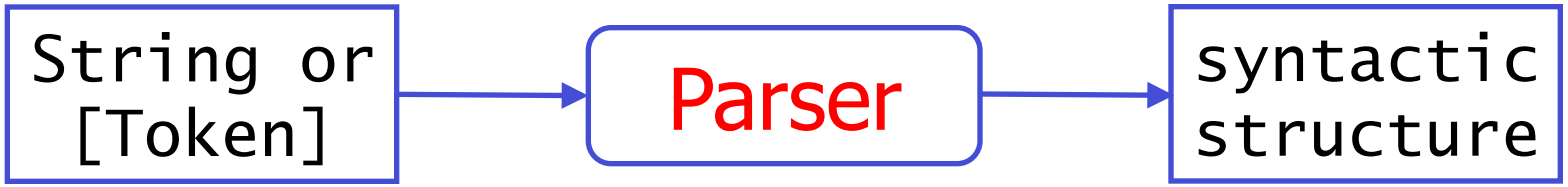
CSCE 314
Programming Languages

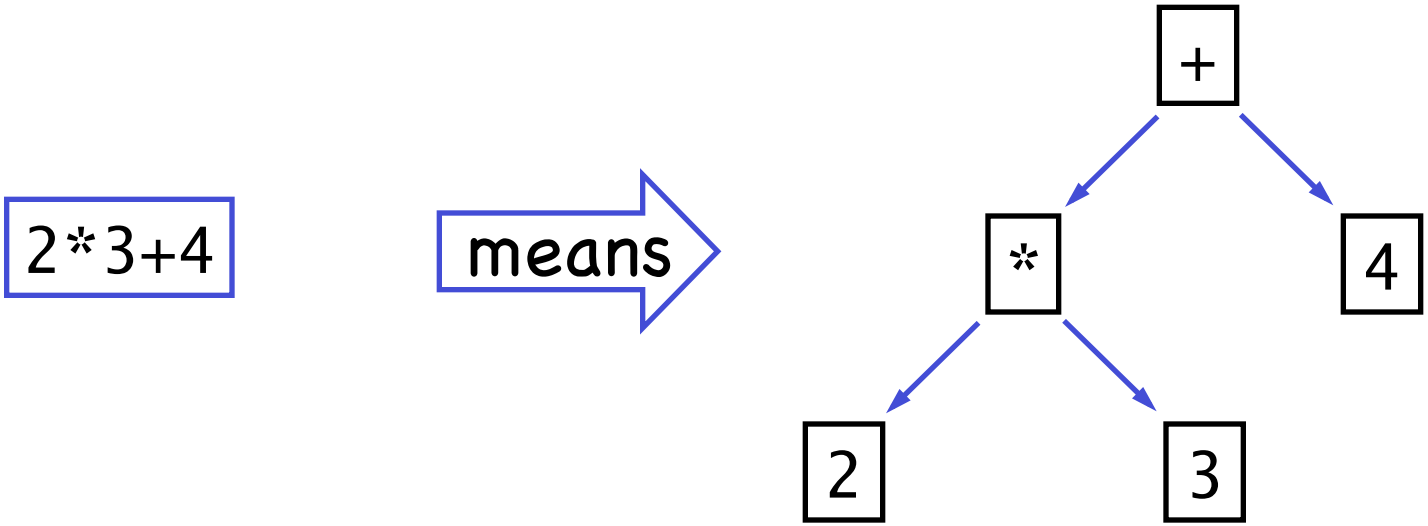
Functional Parsers

Dr. Hyunyoung Lee

What is a Parser?

A <u>parser</u> is a program that takes a text (set of tokens) and determines its <u>syntactic structure</u>.





The Parser Type

In a functional language such as Haskell, parsers can naturally be viewed as <u>functions</u>.

```
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 <u>unused input</u>:

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

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

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

Furthermore, a parser might not always produce a tree, so we generalize to a value of <u>any type</u>:

```
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 <u>singleton list</u>.

Basic Parsers (Building Blocks)

The parser <u>item</u> 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 <u>return v</u> *always succeeds*, returning the value v without consuming any input:

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

The parser <u>failure</u> *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 <u>parse</u> 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.

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

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:

<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 <u>do</u>.

For example:

```
p :: Parser (Char,Char)
p  = do x ← item
        item
        y ← item
        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 <u>layout rule</u> applies.

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

- The value returned by the <u>last</u> parser is the value returned by the sequence as a whole.

- If any parser in a sequence of parsers <u>fails</u>, then the sequence as a whole fails. For example:

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

> parse p "ab"
[]
```

```
p :: Parser (Char,Char)
p  = do x ← item
        item
        y ← item
        return (x,y)
```

- The do notation is not specific to the Parser type, but can be used with <u>any</u> monadic type.

The "Monadic" Way

Parser sequencing operator

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

```
> parse ((failure +++ item) >>= (\_ -> item)) "abc"
[('b',"c")]
```

Sequencing

Typical parser structure

```
p1 >>= \v1 ->
p2 >>= \v2 ->
. . .
pn >>= \vn ->
return (f v1 v2 . . . vn)
```

Using do notation

```
do v1 <- p1
   v2 <- p2
. . .
   vn <- pn
return (f v1 v2 . . . vn)
```

If some `vi` is not needed, `vi <- pi` can be written as `pi`, which corresponds to `pi >>= \_ -> ...`

Example

Typical parser structure

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

Using do notation

```
rev3 =
  do v1 <- item
     v2 <- item
     item
     v3 <- item
     return $
       reverse (v1:v2:v3:[])
```

```
> rev3 “abcdef”
[(“dba”,”ef”)]
```

```
> (rev3 >>= (\_ -> item)) “abcde”
[(‘e’,””)]
> (rev3 >>= (\_ -> 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”,””)]
```

Derived Primitives

Parsing a character that <u>satisfies</u> 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

```
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:

```
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 <u>errors</u> in the input string.

Example: Parsing a token

```
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 <u>expressions</u> 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 <u>grammar</u>:

$$
\begin{aligned}
\mathit{expr} &\rightarrow \mathit{term}\ \texttt{'+'}\ \mathit{expr} \mid \mathit{term} \\
\mathit{term} &\rightarrow \mathit{factor}\ \texttt{'*'}\ \mathit{term} \mid \mathit{factor} \\
\mathit{factor} &\rightarrow \mathit{digit} \mid \texttt{'('}\ \mathit{expr}\ \texttt{')'} \\
\mathit{digit} &\rightarrow \texttt{'0'} \mid \texttt{'1'} \mid \ldots \mid \texttt{'9'}
\end{aligned}
$$

However, for reasons of efficiency, it is important to <u>factorize</u> the rules for *expr* and *term*:

$$expr \rightarrow term\ (\text{'+'}\ expr \mid \varepsilon)$$

$$term \rightarrow factor\ (\text{'*'}\ term \mid \varepsilon)$$

Note: The symbol ε denotes the empty string.

It is now easy to translate the grammar into a parser that <u>evaluates</u> expressions, by simply rewriting the grammar rules using the parsing primitives.

That is, we have:

```
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

```
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
```