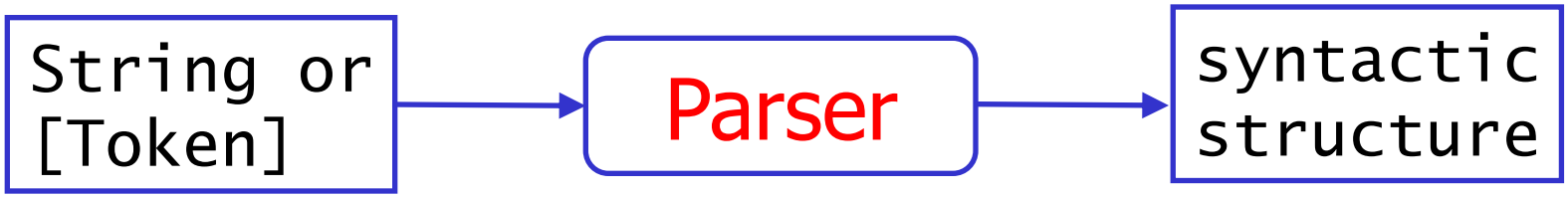
CSCE 314
Programming Languages

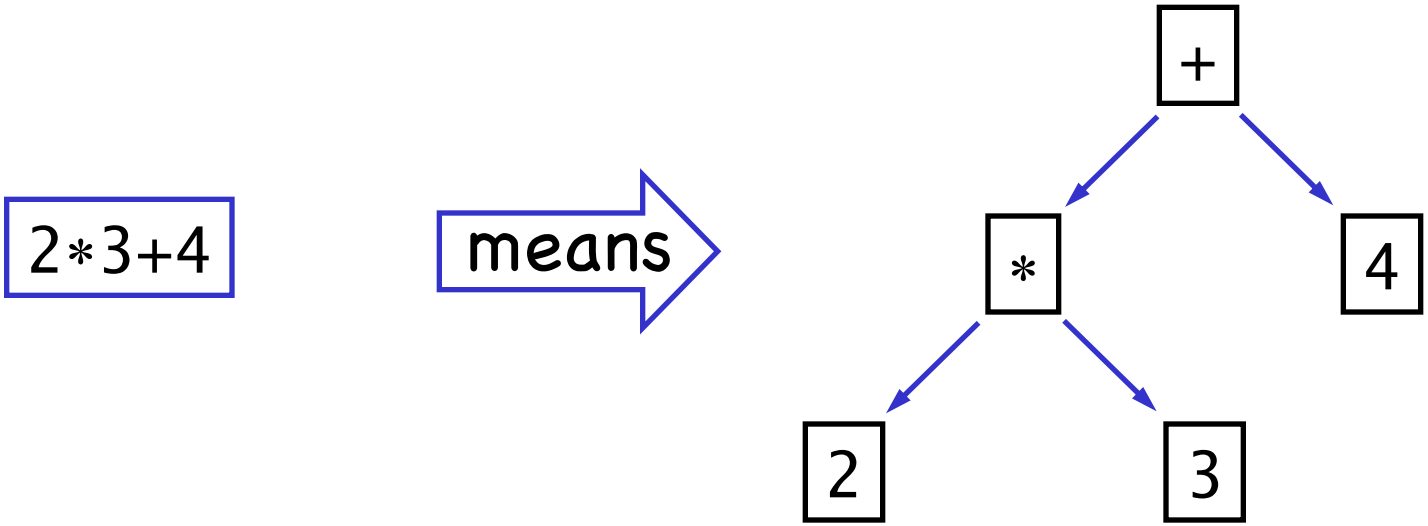
Monadic Parsing

Dr. Hyunyoung Lee

What is a Parser?

A <u>parser</u> is a program that takes a string of characters (or a set of tokens) as input 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>.

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

```
newtype Parser = P (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>:

```
newtype Parser = 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 <u>any type</u>:

```
newtype Parser a = P (String → [(a, String)])
```

Note:

For simplicity, we will only consider parsers that either fail and return the <u>empty list</u> as results, or succeed and return a <u>singleton list</u>.

Basic Parsers (Building Blocks)

We need a function that applies a Parser to an input string:

```
parse :: Parser a -> String -> [(a,String)]
parse (P p) inp = p inp -- apply parser p to inp
```

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

The parser <u>item</u> fails if the input is empty, and consumes the first character otherwise.

Example:

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

```
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:

<if-stmt> :: if (<expr>) then <stmt>

first parse <u>if</u>, then <u>(</u>, then <u><expr></u>, then <u>)</u>, ...

To combine parsers in sequence, we make the Parser type into a monad:

```
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

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
p >>= f = P (\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 (item >>= (\_ -> item)) "abc"
[('b',"c")]
```

Sequencing Parsers (using do)

A sequence of parsers can be combined as a single composite parser using the keyword <u>do</u>.

Example:

```
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."

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

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

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

If any parser in a sequence of parsers <u>fails</u>, 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"
[]
```

>>= or do

Using >>=

```
p1 >>= \v1 ->
p2 >>= \_  ->
P3 >>= \v3 ->
. . .
pn >>= \vn ->
return (f v1 v3 . . . vn)
```

Using do notation

```
do v1 <- p1
   p2
   v3 <- p3
. . .
   vn <- pn
return (f v1 v3 . . . vn)
```

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

Example

Using >>=

```
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:[])
```

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

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")]
```

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 <u>satisfies</u> a predicate:

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

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    =  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:

```
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 = 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)
```