

计算概论A—实验班

函数式程序设计

Functional Programming

胡振江，张 伟

北京大学 计算机学院

2022年09~12月

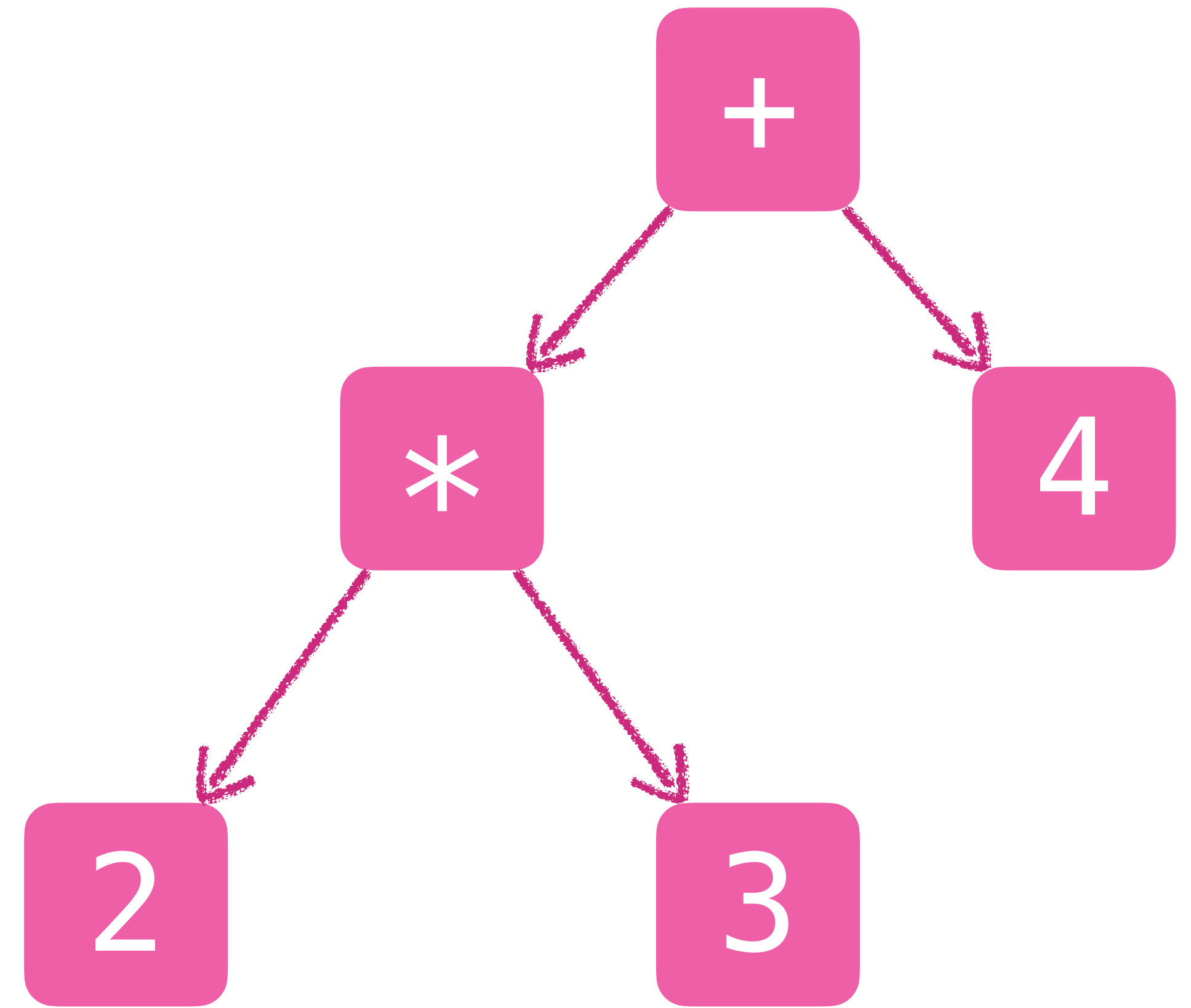
# 第13章： Monadic Parser

# What is a **Parser** (解析器)?

- ❖ A parser: a program that analyses a piece of text to determine its syntactic structure.

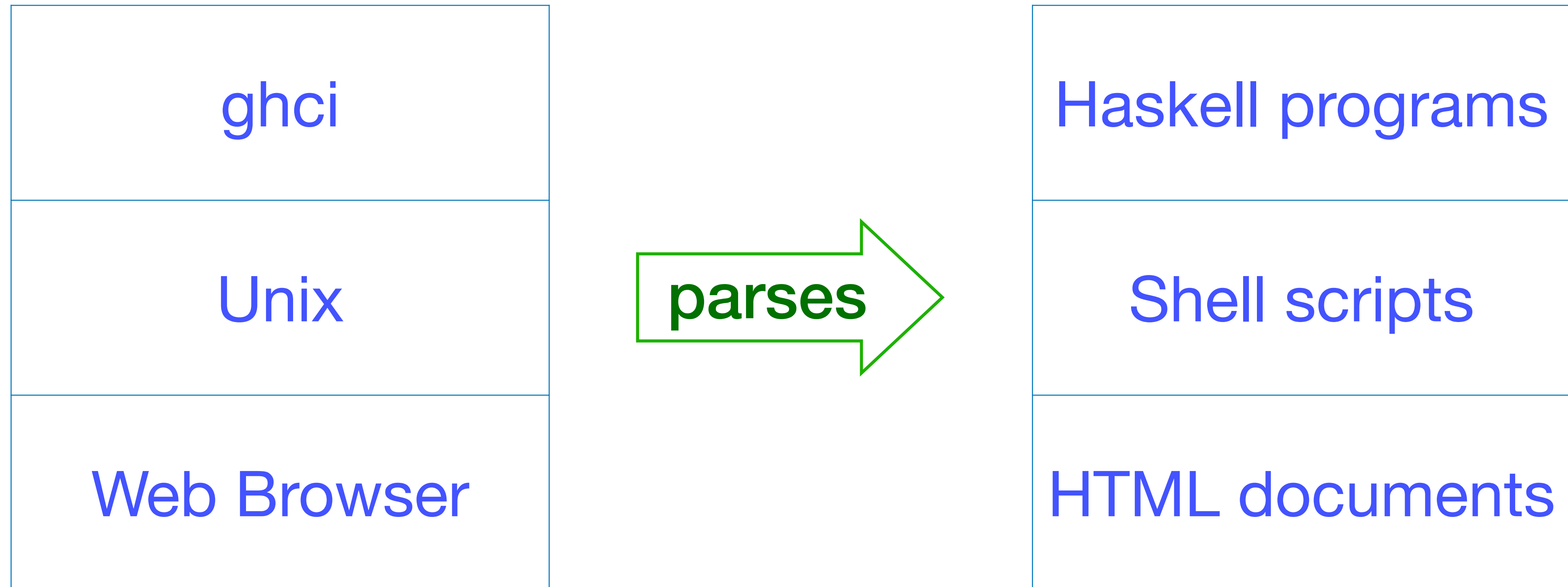
2 \* 3 + 4

means



# Where Are They Used?

- ❖ Almost every real life program uses some form of parser to pre-process its input.



# Parsers as Functions

- ✿ 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.

# Parsers as Functions

- ✿ 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)]
```

# Parsers as Functions

- ✿ Finally, a parser might not always produce a tree, so we generalize to a value of any type.

```
type Parser a = String -> [(a, String)]
```

- \* For simplicity, we will only consider parsers that either fail and return the empty list of results, or succeed and return a singleton list.

# The final type for parsers

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

```
parse :: Parser a -> String -> [(a,String)]
```

```
parse (P f) program = f program
```



# The `item` parser

- ❖ The `item` parser fails if the input is empty, and consumes the first character otherwise:

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

```
ghci> parse item ""
[]
```

```
ghci> parse item "abc"
[('a', "bc")]
```

# Sequencing Parsers

```
instance Functor Parser where
  -- fmap :: (a -> b) -> Parser a -> Parser b
  fmap g p = P $ \program -> case parse p program of
    []          -> []
    [(v,out)]  -> [(g v, out)]
```

```
ghci> parse (toUpper <$> item) "abc"
[('A', "bc")]
ghci> parse (toUpper <$> item) ""
[]
```

# Sequencing Parsers

```
instance Applicative Parser where
  -- pure :: a -> Parser a
  pure v = P $ \program -> [(v,program)]

  -- <*> :: Parser (a -> b) -> Parser a -> Parser b
  pg <*> px = P $ \program -> case parse pg program of
    [] -> []
    [(g,out)] -> parse (g <$> px) out
```

```
ghci> parse (pure 1) "abc"
[(1,"abc")]
```

```
ghci> three = g <$> item <*> item <*> item where g x y z = (x,z)
```

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

# Sequencing Parsers

```
instance Monad Parser where
```

```
  -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
```

```
  p >>= f = P $ \program -> case parse p program of
```

```
    [] -> []
```

```
    [(v,out)] -> parse (f v) out
```

```
ghci> parse (return 1) "abc"
```

```
[(1,"abc")]
```

```
ghci> three = do {x <- item; item; z <- item; return (x,z)}
```

```
ghci> parse three "abcdef"
```

```
[(('a','c'),"def")]
```

# Making Choices

A type class defined in `Control.Applicative`

```
-- A monoid on applicative functors.  
class Applicative f => Alternative f where
```

```
-- An associative binary operation
```

```
(<|>) :: f a -> f a -> f a
```

```
-- The identity of '<|>'
```

```
empty :: f a
```

```
-- | Zero or more.
```

```
many :: f a -> f [a]
```

```
many v = some v <|> pure []
```

```
-- | One or more.
```

```
some :: f a -> f [a]
```

```
some v = (:) <$> v <*> many v
```

▶  $x \langle | \rangle (y \langle | \rangle z) = (x \langle | \rangle y) \langle | \rangle z$

▶  $\text{empty} \langle | \rangle x = x$

▶  $x \langle | \rangle \text{empty} = x$

# Making Choices

Declare Maybe as an instance of Alternative

```
instance Alternative Maybe where
  -- empty :: Maybe a
  empty = Nothing

  -- (<|>) :: Maybe a -> Maybe a -> Maybe a
  Nothing <|> r = r
  l <|> _ = l
```

```
ghci> import Control.Applicative
ghci> some Nothing
Nothing
ghci> many Nothing
Just []
```

# Making Choices

```
instance Alternative Parser where
```

```
  -- empty :: Parser a
```

```
  empty = P $ \program -> []
```

```
  -- (<|>) :: Parser a -> Parser a -> Parser a
```

```
  p <|> q = P $ \program -> case parse p program of  
    [] -> parse q program  
    rst -> rst
```

```
ghci> parse empty "abc"
```

```
[]
```

```
ghci> parse (item <|> return 'd') "abc"
```

```
[('a', "bc")]
```

```
ghci> parse (empty <|> return 'd') "abc"
```

```
[('d', "abc")]
```

# 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 empty
```



# Derived Primitives

- ✿ Parsers for single digits, lower-case letters, upper-case letters, arbitrary letters, alphanumeric characters, and specific characters.

```
digit :: Parser Char
digit = sat isDigit
```

```
lower :: Parser Char
lower = sat isLower
```

```
upper :: Parser Char
upper = sat isUpper
```

```
letter :: Parser Char
letter = sat isAlpha
```

```
alphanum :: Parser Char
alphanum = sat isAlphaNum
```

```
char :: Char -> Parser Char
char x = sat (x ==)
```

# 课堂练习

❖ 定义一个parser,

```
string :: String -> Parser String
```

分析输入的文字是否具有一个给定的前缀

```
ghci> parse (string "abc") "abcdef"
[("abc","def")]
ghci> parse (string "abc") "ab1234"
[]
ghci> parse (string "") "ab1234"
[("", "ab1234")]
```

# 课堂练习

❖ 定义一个parser,

```
string :: String -> Parser String
```

分析输入的文字是否具有一个给定的前缀

```
string :: String -> Parser String
string [] = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)
```

# The `ident` Parser

```
ident :: Parser String
ident = do x <- lower
          xs <- many alphanum
          return (x:xs)
```

```
ghci> parse ident "abc def"
[("abc"," def")]
ghci> parse ident "12 def"
[]
```

# The `nat` Parser

```
nat :: Parser Int
nat = do xs <- some digit
        return (read xs)
```

```
ghci> parse nat "123abc"
[(123,"abc")]
ghci> parse nat "abc123"
[]
```

# The `space` Parser

```
space :: Parser ()  
space = do many (sat isSpace)  
         return ()
```

```
ghci> parse space "   abc"  
[((), "abc")]
```

# The `int` Parser

```
int :: Parser Int
int = do char '-'
         n <- nat
         return $ - n
<|> nat
```

```
ghci> parse int "123abc"
[(123,"abc")]
ghci> parse int "-123abc"
[(-123,"abc")]
ghci> parse int "abc123"
[]
```

# Handling Spacing: **token**

```
token :: Parser a -> Parser a
```

```
token p = do space
```

```
    v <- p
```

```
    space
```

```
    return v
```

```
identifier :: Parser String  
identifier = token ident
```

```
natural :: Parser Int  
natural = token nat
```

```
integer :: Parser Int  
integer = token int
```

```
symbol :: String -> Parser String  
symbol xs = token $ string xs
```



# The `nats` Parser

```
nats :: Parser [Int]
nats = do symbol "["
          n <- natural
          ns <- many $ do {symbol ","; natural}
          symbol "]"
          return (n:ns)
```

```
ghci> parse nats "[1, 2, 3 ]"
[( [1,2,3], "" )]
ghci> parse nats "[1, 2, 3, ]"
[]
```

# 应用：算术表达式的句法解释及评估

- ♣ 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 **+**.

# 应用：算术表达式的句法解释及评估

- Formally, the syntax of such expressions is defined by the following context free grammar:

```
expr ::= term '+' expr | term
term  ::= factor '*' term | factor
factor ::= digit | '(' expr ')
digit ::= '0' | '1' | ... | '9'
```

```
expr ::= term ('+' expr | ε)
term  ::= factor ('*' term | ε)
factor ::= digit | '(' expr ')
digit ::= '0' | '1' | ... | '9'
```

\* The symbol  $\epsilon$  denotes the empty string.

# 应用：算术表达式的句法解释及评估

- ✿ It is now easy to translate the grammar into a parser that evaluates expressions, by simply rewriting the grammar rules using the parsing primitives.

```
expr ::= term ('+' expr | ε)
```

```
expr :: Parser Int
expr = do t <- term
        do symbol "+"
           e <- expr
           return (t + e)
        <|> return t
```

# 应用：算术表达式的句法解释及评估

- ✿ It is now easy to translate the grammar into a parser that evaluates expressions, by simply rewriting the grammar rules using the parsing primitives.

```
term ::= factor ('*' term | ε)
```

```
term :: Parser Int
term = do f <- factor
        do symbol "*"
            t <- term
            return (f * t)
        <|> return f
```

# 应用：算术表达式的句法解释及评估

- ✿ It is now easy to translate the grammar into a parser that evaluates expressions, by simply rewriting the grammar rules using the parsing primitives.

```
factor ::= digit | '(' expr ')'
```

```
factor :: Parser Int
factor = do symbol "("
            e <- expr
            symbol ")"
            return e
          <|> natural
```

# 应用：算术表达式的句法解释及评估

✿ Finally, if we define

```
eval :: String -> Int
eval xs = fst $ head $ parse expr xs
```

then we try out some examples:

```
ghci> eval "2 * ( 3 + 4 )"
14
ghci> eval "2 * 3 + 4"
10
```

# 作业



13-1 Extend the expression parser to allow the use of **subtraction** and **division**, based upon the following extensions to the grammar:

```
expr ::= term ( '+' expr | '-' expr |  $\epsilon$  )
```

```
term ::= factor ( '*' term | '/' term |  $\epsilon$  )
```

# 第13章： Monadic Parser

就到这里吧