Adapted from Graham's Lecture slides.

第十三章：Monadic Parser

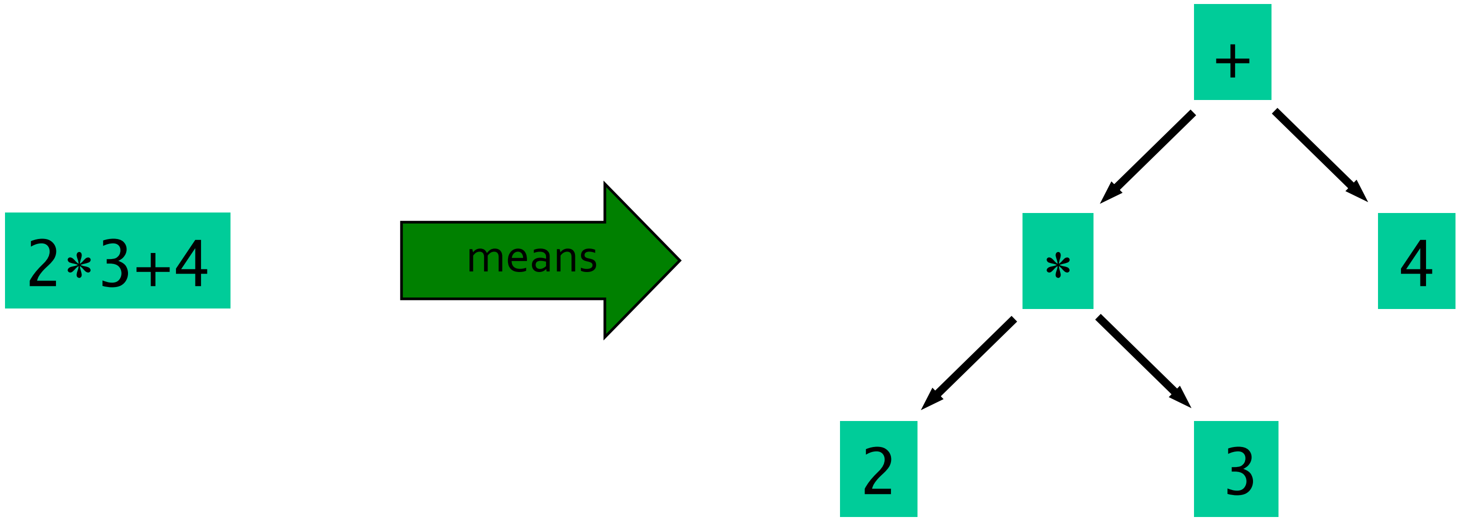
Parser的概念, 作为函数的Parser

构造Parser的DSL

算术表达式的句法分析

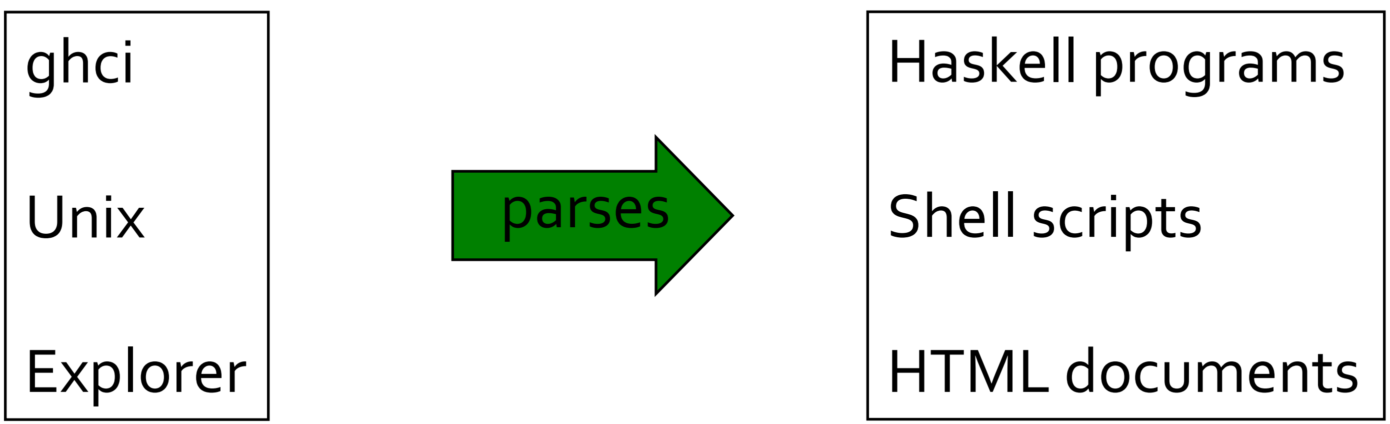
What is a Parser?

A <u>parser</u> is a program that analyses a piece of text to determine its <u>syntactic structure</u>.



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.

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)]
```

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

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

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.

基本定义

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

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

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

```
item :: Parser Char
item  = P (λinp → case inp of
                     []     → []
                     (x:xs) → [(x,xs)])
```

```
> parse item ""
[]

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

Sequencing Parsers

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

```
> parse (fmap toUpper item) "abc"
[('A',"bc")]

> parse (fmap toUpper item) ""
[]
```

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

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

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

> parse three "abcdef"
[((’a’,’c’),"def”)]
```

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


北京大学 PEKING UNIVERSITY

```
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)
```

```
> parse (return 1) "abc"
[(1,"abc”)]

> parse three "abcdef"
[((’a’,’c’),"def”)]
```

```
three :: Parser (Char,Char)
three = do x <- item
           item
           z <- item
           return (x,z)
```

Making Choices

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
```

```
empty <|> x       = x
x <|> empty       = x
x <|> (y <|> z)   = (x <|> y) <|> z
```

Making Choices

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a

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

    many x = some x <|> pure []
    some x = pure (:) <*> x <*> many x
```

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

    -- (<|>) :: Maybe a -> Maybe a -> Maybe a
    Nothing <|> my     = my
    (Just x) <|> _     = Just x
```


PEKING UNIVERSITY

```
instance Alternative Parser where
    -- empty :: Parser a
    empty = P (\inp -> [])

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

```
> parse empty "abc"
[]

> parse (item <|> return 'd') "abc"
[('a',"bc")]

> 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
```

- 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 ==)
```

练习： 定义一个parser:

string :: String -> Parser String

分析输入是不是一个给定的文字序列。

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

> parse (string "abc") "ab1234"
[]
```

- Ident

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

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

- nat

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

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

- space

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

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

- int

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

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

Handling Spacing: token

```
token :: Parser a -> Parser a
token p = do   space
               v <- p
               space
               return v
```

```
identifier = token ident
natural = token nat
integer = token int
symbol xs = token (string xs)
```

- nats

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

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

> parse nats "[1,2,]"
[]
```

应用：算术表达式的句法解释及计算

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:

```
expr   ::= term '+' expr | term

term   ::= factor '*' term | factor

Factor ::= digit | '(' expr ')'

digit  ::= '0' | '1' | … | '9'
```

However, for reasons of efficiency, it is important to factorise the rules for *expr* and *term*:

$$expr \rightarrow term\ ('+'\ expr \mid \varepsilon)$$

$$term \rightarrow factor\ ('*'\ term \mid \varepsilon)$$

Note:

- The symbol ε 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:

```
expr :: Parser Int
expr  = do t ← term
           do char '+'
              e ← expr
              return (t + e)
            <|> return t
```

expr → *term* ('+' *expr* | ε)

```
term :: Parser Int
term  = do f ← factor
           do char '*'
              t ← term
              return (f * t)
           <|> return f
```

term → *factor* ('*' *term* | ε)

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

Factor ::= digit | '(' *expr* ')'

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
```

作业

13-1 Why does factorising the expression grammar make the resulting parser more efficient?

13-2 Extend the expression parser to allow the use of subtraction and division, based upon the following extensions to the grammar:

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

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