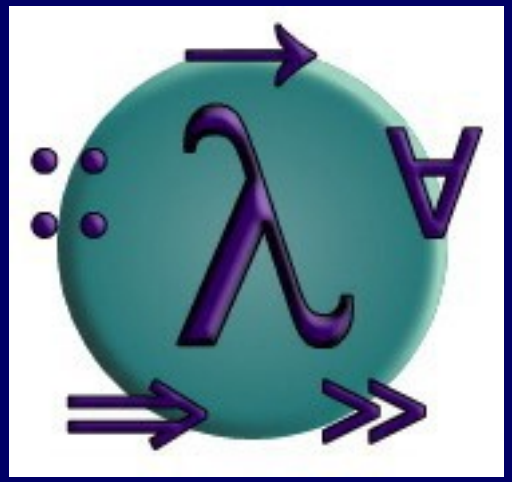
PROGRAMMING IN HASKELL

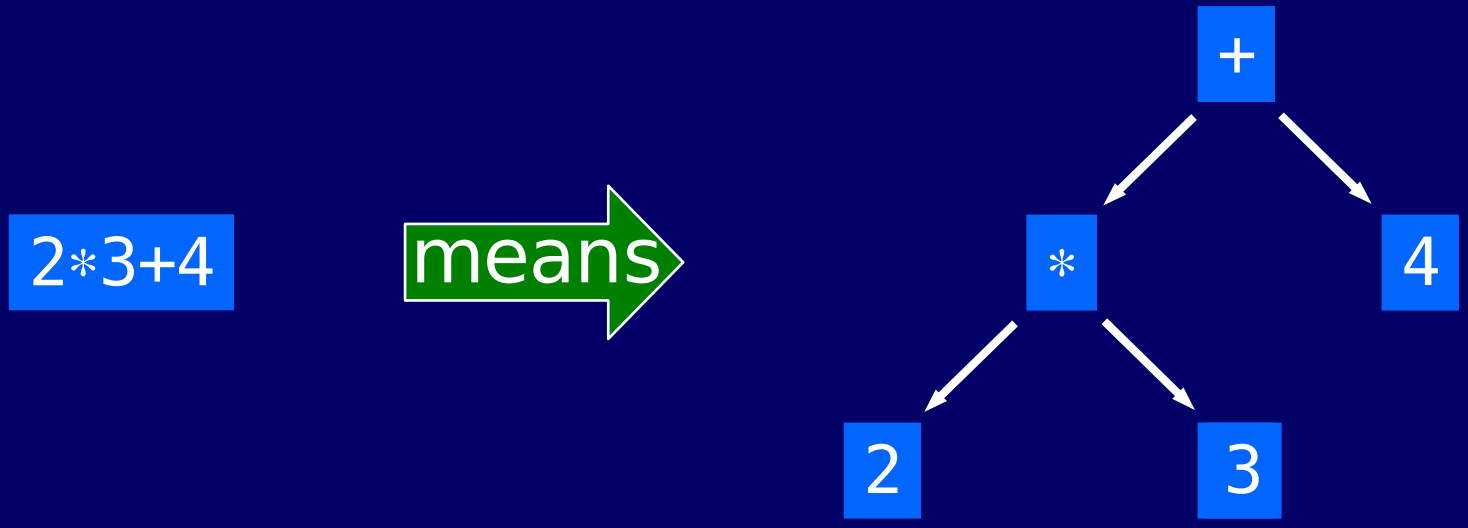


Chapter 7 - Functional Parsers

Original Author: Graham Hutton
(http://www.cs.nott.ac.uk/~gmh/book.html)

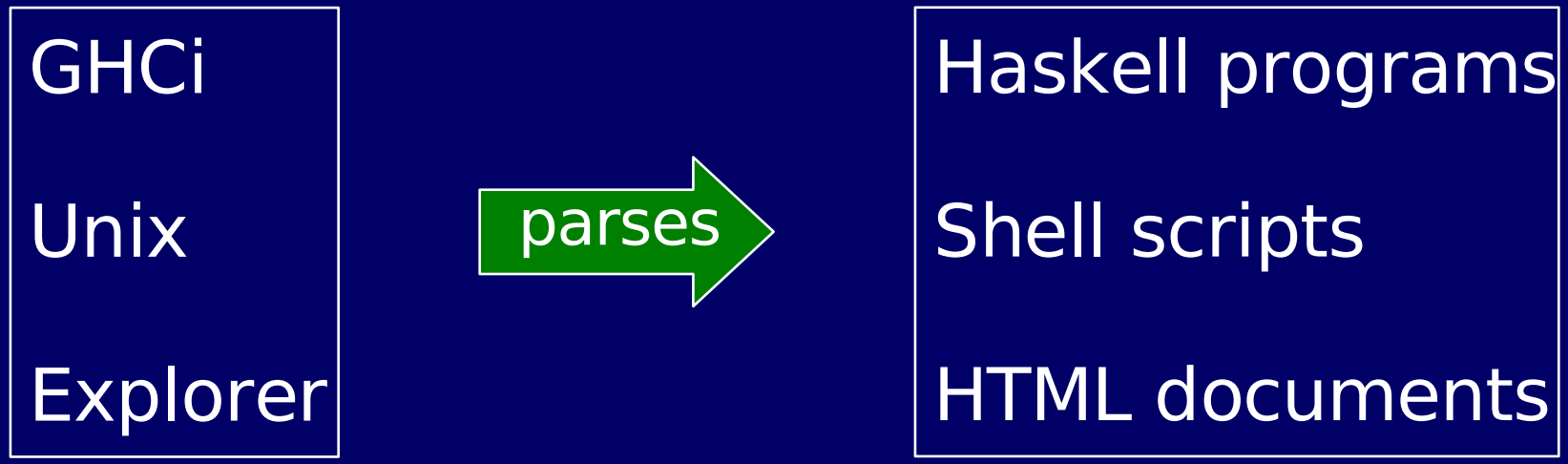
What is a Parser?

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



Where Are They Used?

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



The Parser Type

In a functional language such as Haskell, parsers can naturally be viewed as functions.

A parser is a function that takes a string and returns some form of tree.

```
type Parser = String → 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.

Basic Parsers

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

```
anyChar :: Parser Char

anyChar  = λinp → case inp of

                    []     → []

                    (x:xs) → [(x,xs)]
```

- The parser pzero always fails:

```
pzero :: Parser a
pzero  = λinp → []
```

- The parser return v always succeeds, returning the value v without consuming any input:

```
return  :: a → Parser a
return v = λinp → [(v,inp)]
```

- 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 = λinp → case p inp of

                   []        → parse q inp

                   [(v,out)] → [(v,out)]
```

- The function parse applies a parser to a string:

```
parse :: Parser a → String → [(a,String)]

parse p inp = p inp
```

Examples

The behavior of the five parsing primitives can be illustrated with some simple examples:

```
% ghci
Prelude> :m Text.ParserCombinators.Parsec

> parseTest anyChar ""
parse error at (line 1, column 1):
unexpected end of input

> parseTest anyChar "abc"
'a'
```

```
> parseTest pzero "abc"
parse error at (line 1, column 1): unknown parse error

> parseTest (return 1) "abc"
1

> parseTest (anyChar <|> return 'd') "abc"
'a'

> parseTest (pzero <|> return 'd') "abc"
'd'
```

Note:

- Functional Parsing Combinators are shipped with GHCi in module Text.ParserCombinators.Parsec

- The Parser type is a monad, a mathematical structure that has proved useful for modeling many different kinds of computations.

Sequencing

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

For example:

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

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:

```
> parseTest p "abcdef"
('a','c')

> parseTest p "ab"
parse error
```

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

Derived Primitives

- Parsing a character that satisfies a predicate:

```
satisfy :: (Char → Bool) → Parser Char
satisfy p = do x ← anyChar
               if p x then
                  return x
               else
                  pzero
```

- Parsing a digit and specific characters:

```
digit :: Parser Char
digit  = satisfy isDigit

char  :: Char → Parser Char
char x = satisfy (x ==)
```

- Applying a parser zero or more times:

```
many  :: Parser a → Parser [a]
many p = many1 p <|> return []
```

- Applying a parser one or more times:

```
many1  :: Parser a -> Parser [a]
many1 p = do v  ← p
             vs ← many p
             return (v:vs)
```

- Parsing a specific string of characters:

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

Example

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

```
p :: Parser String
p  = do char '['
        d  ← digit
        ds ← many (do char ','
                      digit)
        char ']'
        return (d:ds)
```

For example:

```
> parseTest p "[1,2,3,4]"
"1234"

> parseTest p "[1,2,3,4"
parse error in ...
```

Note:

- More sophisticated parsing libraries can indicate and/or recover from errors in the input string.

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:

```
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*:

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

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

Note:

- The symbol ε denotes the empty string.

It is now easy to translate the grammar into a parser that 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
```

```
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
```