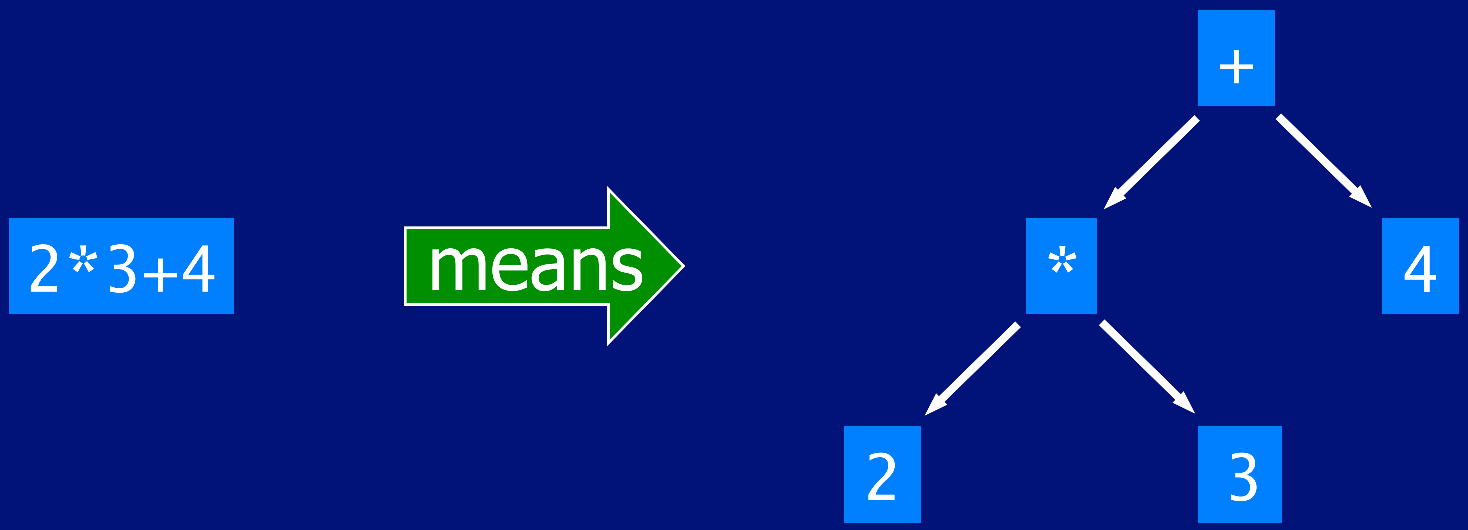
What is a Parser?

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.

GHC		Haskell programs
Unix	parses →	Shell scripts
Explorer		HTML documents

The Parser Type

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.

Basic Parsers

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

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

- The parser failure always fails:

```
failure :: Parser a
failure  = 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 <u>examples</u>:

```
% ghci Parsing

> 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 <u>Parsing</u> is available on the web from the Programming in Haskell home page.

- For technical reasons, the first failure example actually gives an error concerning <u>types</u>, but this does not occur in non-trivial examples.

- The Parser type is a <u>monad</u>, 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 ← item
        item
        y ← item
        return (x,y)
```

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 $\leftarrow$ 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 fails, then the sequence as a whole fails. For example:

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

> parse p "ab"
[]
```

- 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:

```
sat  :: (Char → Bool) → Parser Char
sat p = do x ← item
           if p x then
              return x
            else
              failure
```

- Parsing a digit and specific characters:

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

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

- Applying a parser zero or more times:

```
many  :: Parser a → Parser [a]
many p = many1 p +++ return []
```

- Applying a parser <u>one or more</u> times:

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

- Parsing a specific <u>string</u> 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:

```
> 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.

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:

$$expr \rightarrow term\ \texttt{'+'}\ expr \mid term$$

$$term \rightarrow factor\ \texttt{'*'}\ term \mid factor$$

$$factor \rightarrow digit \mid \texttt{'('}\ expr\ \texttt{')'}$$

$$digit \rightarrow \texttt{'0'} \mid \texttt{'1'} \mid \ldots \mid \texttt{'9'}$$

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

```
expr → term ('+' expr | ε)

term → factor ('*' term | ε)
```

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
```

Exercises

(1) Why does factorising the expression grammar make the resulting parser more efficient?

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

```
expr → term ('+' expr | '-' expr | ε)

term → factor ('*' term | '/' term | ε)
```