Chapter 8 - Functional Parsers
What is a Parser?

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

$2 \times 3 + 4$ means $2 \times 3 + 4$
Where Are They Used?

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

- Hugs
- Unix
- Explorer

parses

- Haskell programs
- Shell scripts
- 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:

\[
\text{type Parser } a = \text{String } \rightarrow [(a,\text{String})]
\]

**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 \texttt{item} fails if the input is empty, and consumes the first character otherwise:

\begin{verbatim}
item :: Parser Char
item = λinp → case inp of
    []    → []
    (x:xs) → [(x,xs)]
\end{verbatim}
The parser **failure** always fails:

\[
\text{failure} :: \text{Parser } a \\
\text{failure} = \lambda \text{inp} \to []
\]

The parser **return** \(v\) always succeeds, returning the value \(v\) without consuming any input:

\[
\text{return} :: a \to \text{Parser } a \\
\text{return } v = \lambda \text{inp} \to [(v,\text{inp})]
\]
The parser \( p +++ q \) behaves as the parser \( p \) if it succeeds, and as the parser \( q \) otherwise:

\[
(++) :: \text{Parser } a \rightarrow \text{Parser } a \rightarrow \text{Parser } a
\]

\[
p +++ q = \lambda \text{inp} \rightarrow \text{case } p \text{ inp of}
\]

\[
[] \rightarrow \text{parse } q \text{ inp}
\]

\[
[(v,\text{out})] \rightarrow [(v,\text{out})]
\]

The function \text{parse} applies a parser to a string:

\[
\text{parse} :: \text{Parser } a \rightarrow \text{String} \rightarrow [(a,\text{String})]
\]

\[
\text{parse } p \text{ inp} = p \text{ inp}
\]
Examples

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

```
% hugs 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 **Parsing** is available on the web from the Programming in Haskell home page.

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

- 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:

```haskell
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 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:

```haskell
> 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:

\[
\text{sat} :: (\text{Char} \rightarrow \text{Bool}) \rightarrow \text{Parser Char}
\]
\[
\text{sat } p = \text{do } x \leftarrow \text{item}
\]
\[
\quad \text{if } p \ x \ \text{then}
\]
\[
\quad \quad \text{return } x
\]
\[
\quad \text{else}
\]
\[
\quad \text{failure}
\]
 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 **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:

\[
p :: \text{Parser}\ \text{String} \\
p = \text{do char ']['} \\
\quad d \leftarrow \text{digit} \\
\quad ds \leftarrow \text{many (do char ',')} \\
\quad \quad \quad \text{digit} \\
\quad char ']' \\
\quad \text{return (d:ds)}
\]
For example:

\[
\begin{array}{l}
> \text{parse p } "[1,2,3,4]"\\
[("1234","")]\\
\\
> \text{parse p } "[1,2,3,4"\\
[]
\end{array}
\]

Note:

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

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

\[
term \rightarrow \text{factor} ('\star' \ text \mid \varepsilon)
\]

Note:

- The symbol $\varepsilon$ 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:

```haskell
expr :: Parser Int
expr  = do t ← term
          do char '+'
             e ← expr
             return (t + e)
+++ return t
```
factor :: Parser Int
factor = do d ← digit
         return (digitToInt d)
         +++ do char '('
              e ← expr
              char ')'
              return e

term :: Parser Int
term = do f ← factor
        do char '*'
           t ← term
           return (f * t)
           +++ return f
Finally, if we define

\[
\text{eval} :: \text{String} \rightarrow \text{Int} \\
\text{eval } \text{xs} = \text{fst} \ (\text{head} \ (\text{parse expr } \text{xs}))
\]

then we try out some examples:
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:

\[
\begin{align*}
\text{expr} & \rightarrow \text{term} (\ '+' \ \text{expr} \mid '-' \ \text{expr} \mid \varepsilon) \\
\text{term} & \rightarrow \text{factor} (\ '*' \ \text{term} \mid '/' \ \text{term} \mid \varepsilon)
\end{align*}
\]