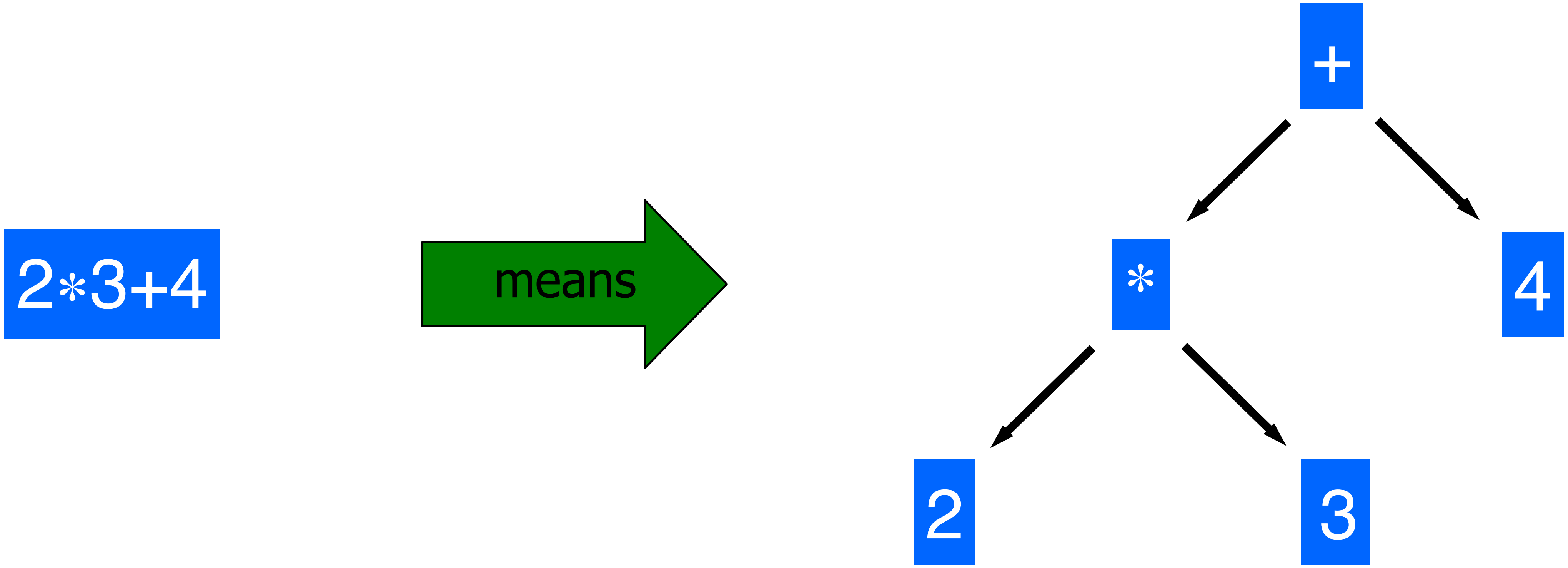
PROGRAMMING IN HASKELL

Chapter 8 - Functional Parsers

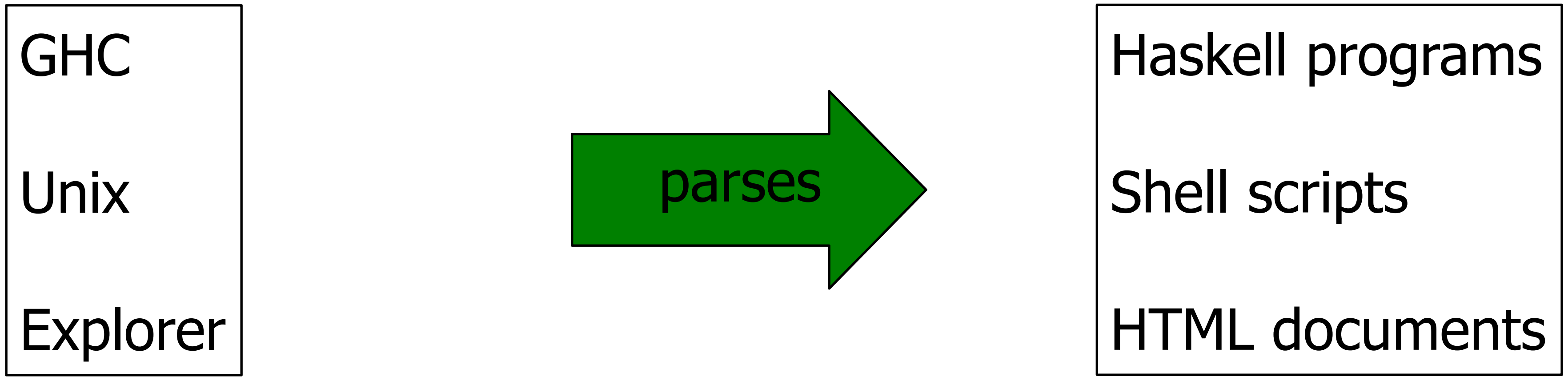
What is a Parser?

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.

```
data Parser = P (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 <u>unused input</u>:

```
data Parser = P (String → (Tree,String))
```

A string might be parsable in many ways, including none, so we generalize to a <u>list of results</u>:

```
data Parser = P (String → [(Tree,String)])
```

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

```
data Parser a = P (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 <u>singleton list</u>.

Basic Parsers

- The parser <u>item</u> 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)])
```

- The parser <u>failure</u> always fails:

```
failure :: Parser a
failure  = P (λinp → [])
```

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

```
return  :: a → Parser a
return v = P (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 = P (λinp → case parse 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 p) inp = p inp
```

Examples

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

```
% 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 Parsing is available on the web from Moodle.
- 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 <u>do</u>.

For example:

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

Sequencing

The do-notation allows you to take out what you parsed from the Parser structure!

Later in the course we will explain how this works.

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

item :: Parser Char

x :: Char

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 $\leftarrow$ 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:

```
> 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 <u>satisfies</u> 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 <u>expressions</u> 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 <u>grammar</u>:

$$\mathit{expr} \rightarrow \mathit{term}\ \text{'+'}\ \mathit{expr} \mid \mathit{term}$$

$$\mathit{term} \rightarrow \mathit{factor}\ \text{'*'}\ \mathit{term} \mid \mathit{factor}$$

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

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

However, for reasons of efficiency, it is important to <u>factorise</u> 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
```

A rule of thumb for the do-notation

The do-notation can be confusing for Haskell beginners. A key difficulty is in understanding the typing of the arrow:

```
do ...
   x ← e
   ...
```

For some type A,
if e :: Parser A then x :: A

term :: Parser Int

```
expr :: Parser Int
expr  = do t ← term
           ar '+'
                expr
           return (t + e)
        +++ return t
```

t :: Int

```
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:

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

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