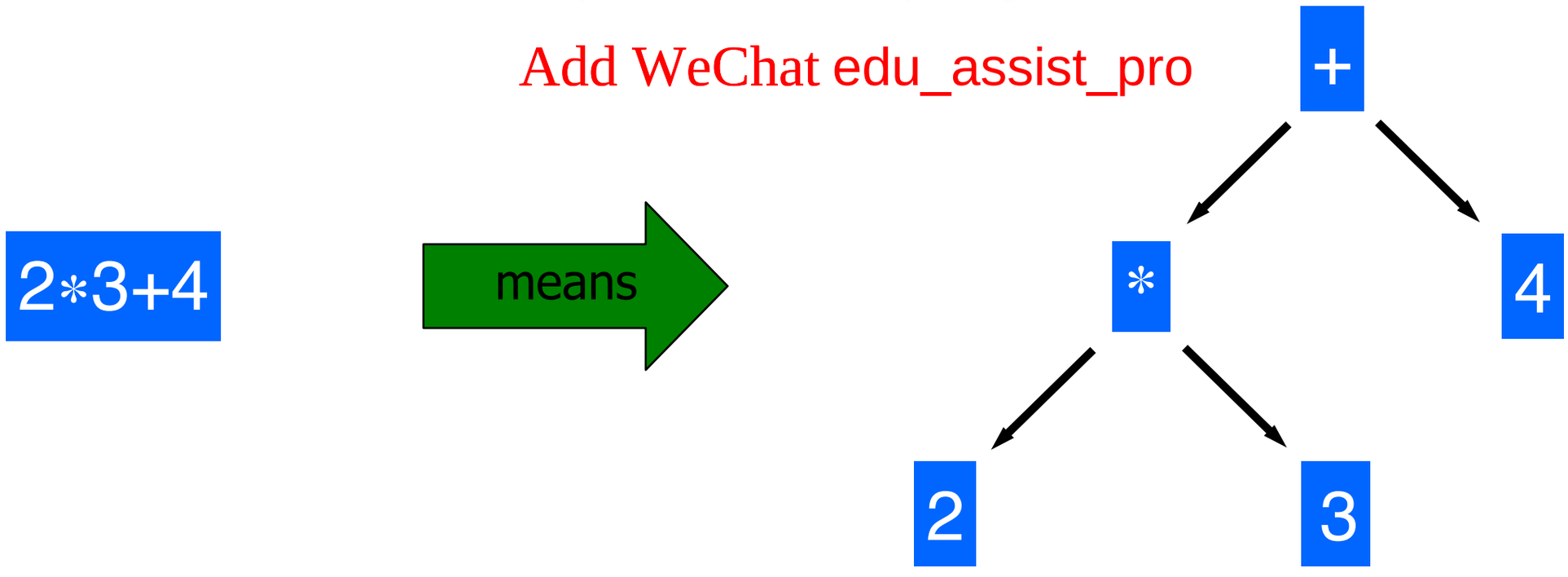
PROGRAMMING IN HASKELL

Chapter 8 - Functional Parsers

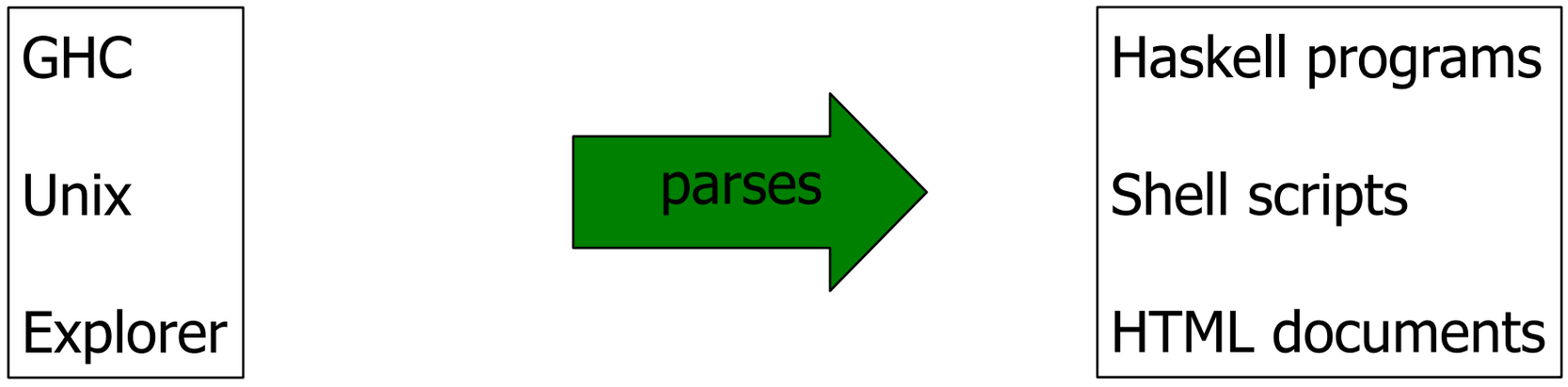
What is a Parser?

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                )
```

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:

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


A string might be parsable in many ways, incl so we generalize to a list of results:

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

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

```
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 singleton list.

Basic Parsers

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


```
item :: Parser Char

item  = P (λinp → ?)
```

- The parser failure always fails:

```
failure :: Parser a
failure  = P (λinp → ?)
```


- The parser return v always succeeds, retu lue v without consuming any input:


```
return  :: a → Parser a
return v = P (λinp → ?)
```

- The parser p +++ q behaves as the parser p if p succeeds, and as the parser q otherwise:

```
(+++)  :: Parser a → Parser a → Parser a
p +++ q = P (λinp
```


- 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')
[('a',"bc")]

> parse (failure +++ return 'd') "abc"
[('d',"abc")]
```

Note:

- The library file Parsing will be 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 mona ure that has proved useful for modeling many different ki

Sequencing

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

For example:


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

Sequencing

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

Later in the course we will explain how this works.


```
p :: Parser (
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 <- operator.
- The value returned by the last parser is 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 = ?
```

- Parsing a digit and specific characters:

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


char  :: Char →
char 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 = ?
```


- Parsing a specific string of


```
string       :: String → Parser String
string = ?
```

Example

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


```
p :: Parser Strin
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
[]
```


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* | *f*

factor → *digit* | '(' *expr* ')‘

digit → '0' | '1' | … | '9'

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

expr → *term* ('+' *expr* | ε)

term → *factor*

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  = ?
```

```
term :: Parser Int
term  = ?
```



```
factor :: Parser Int
factor  = ?
```

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 p f subtraction and division, based upon the following exte

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

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