Parsing Expressions

Slides by Koen Lindström Claessen & David Sands

Expressions

- Such as
  - 5*2+12
  - 17+3*(4*3+75)
- Can be modelled as a datatype

```
data Expr
   = Num Int
   | Add Expr Expr
   | Mul Expr Expr
```

Showing and Reading

- We have seen how to write

```
showExpr :: Expr -> String
```

built-in show function produces ugly results

```
Main> showExpr (Add (Num 2) (Num 4))
"2+4"
Main> showExpr (Mul (Add (Num 2) (Num 3)) (Num 4)
(2+3)*4
```

- This lecture: How to write

```
readExpr :: String -> Expr
```

built-in read function does not match showExpr

Parsing

- Transforming a ”flat” string into something with a richer structure is called *parsing*
  - expressions
  - programming languages
  - natural language (swedish, english, dutch)
  - ...
- Very common problem in computer science
  - Many different solutions

Parser libraries

- Haskell has many nice libraries that make it easy to write parsers
  - E.g. *parsec* included in the Haskell Platform: http://hackage.haskell.org/package/parsec
- In this lecture we will do it from scratch

Expressions

```
data Expr
   = Num Int
   | Add Expr Expr
   | Mul Expr Expr
```

- Let us start with a simpler problem
- How to parse

```
data Expr
   = Num Int
```

but we keep in mind that we want to parse real expressions...

Parsing Numbers

```
number :: String -> Int
```

```
Main> number "23"
23
Main> number "apa"
?
Main> number "23+17"
?
```

Parsing Numbers

- Parsing a string to a number, there
  cases:
  - (1) the string is a number, e.g. ”23”
  - (2) the string is not a number at all, e.g. ”apa”
  - (3) the string *starts* with a number, e.g. ”17+24”

Case (1) and (3) are similar...

how to model these?

A Parser

String -> Maybe (a, String)

```
type Parser a = String -> Maybe (a, String)
```

A Parser for things is a function from Strings to Maybe a thing and a String

Parsing Numbers

- Parsing a string to a number, there are three cases:

  (1) the string is a number, e.g. ”23”

  `Just(23,””)`

  (2) the string is not a number at all, e.g. ”apa”

  `Nothing`

  (3) the string *starts* with a number, e.g. ”17+24”

  `Just(17,”+24”)`

Parsing Numbers

```
number :: Parser Int
```

```
Main> number "23"
Just (23, "")
Main> number "apa"
Nothing
Main> number "23+17"
Just (23, "+17")
```

how to implement?

Case expressions

- We have seen many examples of pattern matching in function definitions

```
rank (Card r _) = r
```

Sometimes we just want to match on a local value given by an expression
Use case expressions for this

```
addPDF :: FilePath -> FilePath
addPDF s = case reverse (take 4 (reverse s)) of
                    ".pdf" -> s
                    _      -> s ++ ".pdf"
```

Note: cases must have same indentation

Parsing Numbers

```
import Data.Char(isDigit)

number :: Parser Int
number (c:s)
  | isDigit c = Just (numb,rest)
  | otherwise = Nothing
    where
    numb = read (takeWhile isDigit (c:s))
    rest = dropWhile isDigit (c:s)
```

```
read :: String -> Int
```

or more generally

```
read :: Read a => String -> a
```

Parsing Numbers

```
number :: Parser Int
```

```
num :: Parser Expr
num s = case number s of
                Just (n, s') -> Just (Num n, s')
                Nothing      -> Nothing
```

a *case expression*

```
Main> num "23"
Just (Num 23, "")
Main> num "apa"
Nothing
Main> num "23+17"
Just (Num 23, "+17")
```

The structure of expression strings

- An expression must be of the form
  "$t_1 + t_2 + \ldots + t_m$"
- Each term t_i must be of the form
  "$f_1 * f_2 * \ldots * f_n$"
- Each factor f_i must be a number

One or more terms with '+' between them

We're currently ignoring parentheses

- We need four different parsers, one for each category: expression, term, factor, number

Parsing strategy

Solves the problem of where to split the string

Each parser will eat as much of the input as “makes sense” to it, and leave the rest untouched

- Parse “1*2+3asd” as an expression
  - result: Add (Mul (Num 1) (Num 2)) (Num 3)
  - rest: “asd”
- Parse “1*2+3asd” as a term
  - result: Mul (Num 1) (Num 2)
  - rest: “+3asd”
- Parse “1*2+3asd” as a factor
  - result: Num 1
  - rest: “*2+3asd”

Parsing example

- Parse “1+2” as an expression
  - Should have the form “$t_1 + t_2 + \ldots + t_m$”, so we start by looking for a term
- Parse “1+2” as a term
  - Should have the form “$f_1 * f_2 * \ldots * f_n$”, so we start by looking for a factor
- Parse “1+2” as a factor
  - Should be a number

… continue on the next slide

Parsing example

- Parse “1+2” as a number
  - Return the number and the rest of the string: (1,“+2”)
- The factor parser returns (Num 1, “+2”)
- The term parser returns (Num 1, “+2”)
- The expression parser now has hold of the first term.
  - Since the rest of the string starts with “+”, it goes on to look for another term.
  - Now the rest of the string is “”, so there are no more terms, and it can return (Add (Num 1) (Num 2), “”)

The structure of expression strings

- An expression must be of the form
  "$t_1 + t_2 + \ldots + t_m$"
- Each term t_i must be of the form
  "$f_1 * f_2 * \ldots * f_n$"
- Each factor f_i must be a number

Expressions

```
data Expr
   = Num Int
   | Add Expr Expr
```

- Expressions are now of the form
  - ”23”
  - ”3+23”
  - ”17+3+23+14+0”

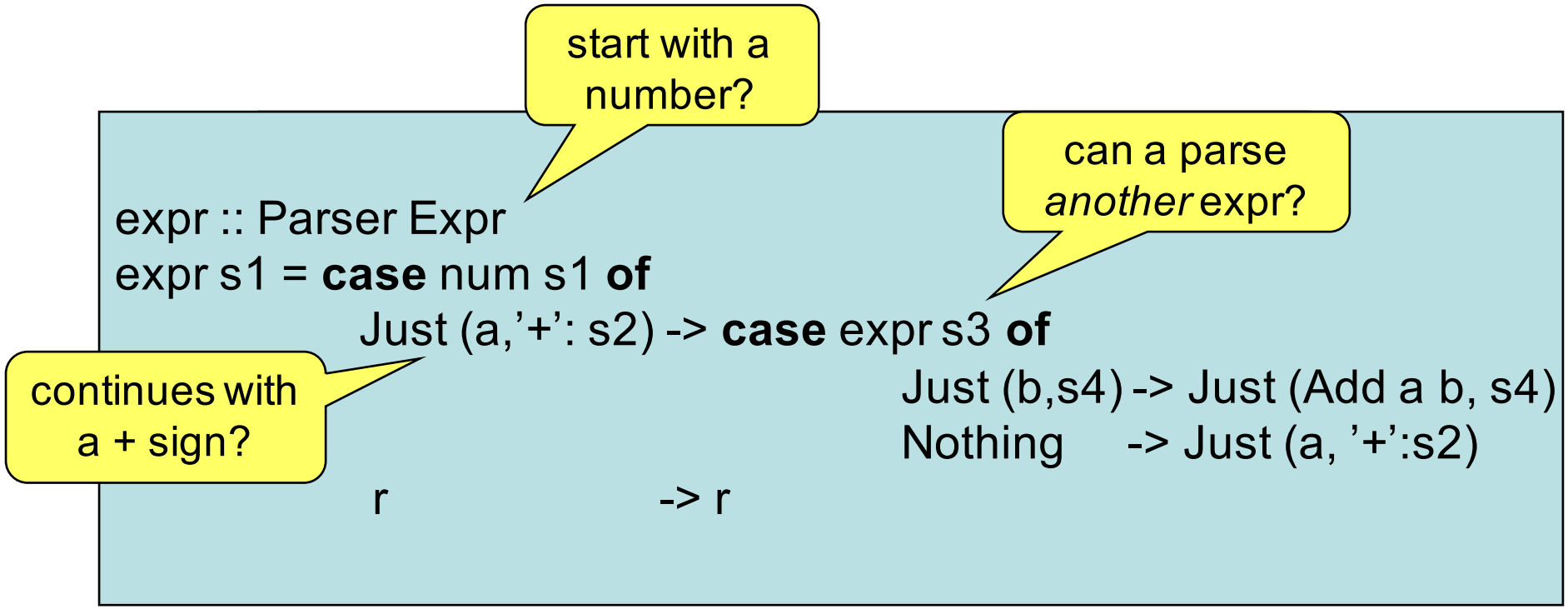
a *chain* of numbers with ”+”

Parsing Expressions

```
expr :: Parser Expr
```

```
Main> expr "23"
Just (Num 23, "")
Main> expr "apa"
Nothing
Main> expr "23+17"
Just (Add (Num 23) (Num 17), "")
Main> expr "23+17)"
Just (Add (Num 23) (Num 17), ")")
```

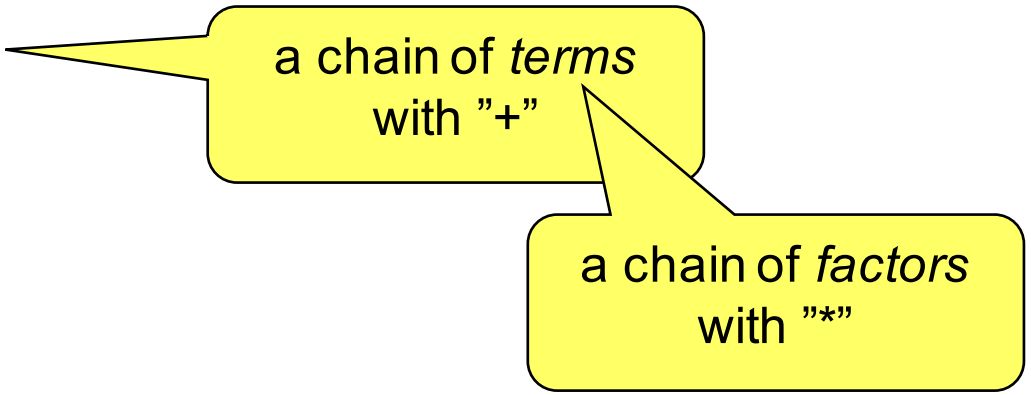
Parsing Expressions



Expressions

```
data Expr
   = Num Int
   | Add Expr Expr
   | Mul Expr Expr
```

- Expressions are now of the form
  - ”23”
  - ”3+23*4”
  - ”17*3+23*5*7+14”



Grammar for Expressions

- Parse Expressions according to the following BNF grammar:

<expr> ::= <term> | <term> "+" <expr>

<term> ::= <factor> | <factor> "*" <term>

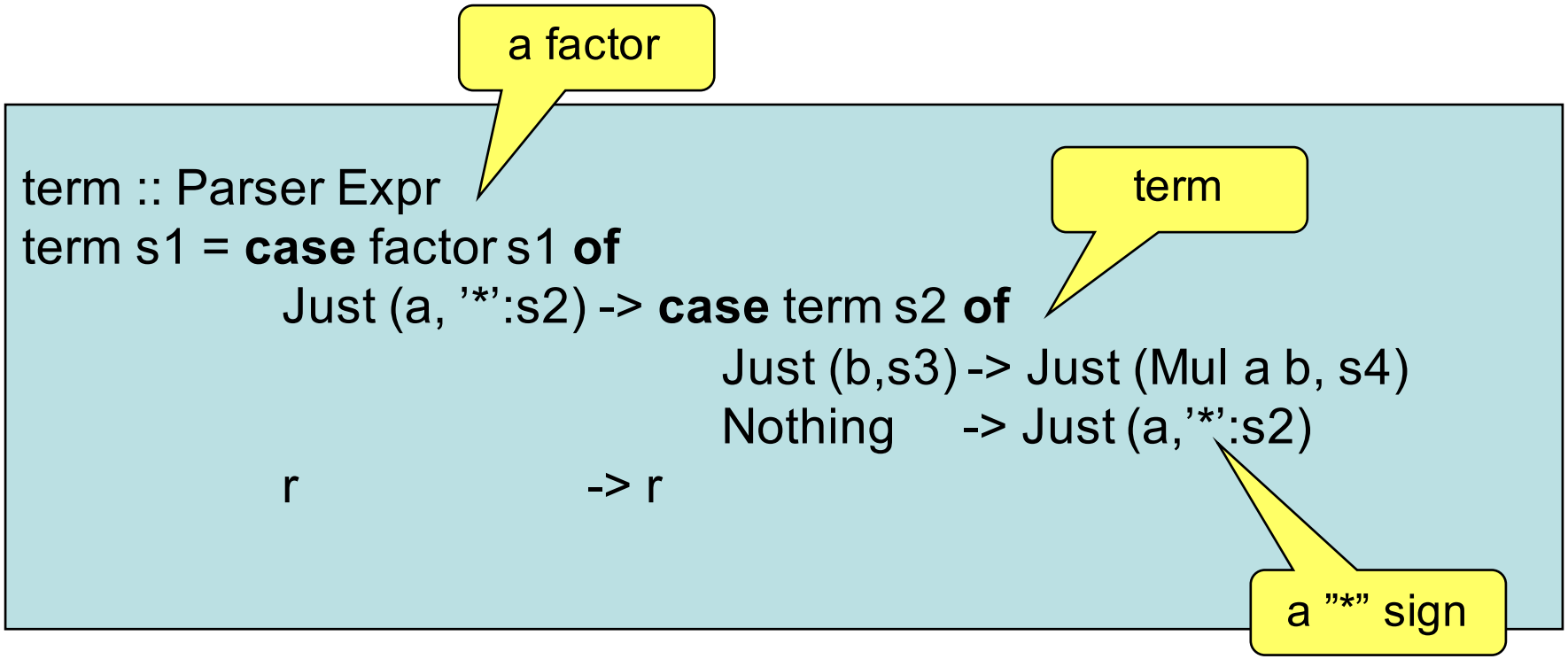
<factor> ::= "(" <expr> ")" | <number>

Parsing Expressions

```
expr :: Parser Expr
expr s1 = case term s1 of
              Just (a,'+':s2) -> case expr s2 of
                                   Just (b,s3) -> Just (Add a b, s3)
                                   Nothing     -> Just (a, '+':s2)
              r               -> r
```

```
term :: Parser Expr
term = ?
```

Parsing Terms



Horrible cut-and-paste programming!

Better: abstract over the differences between term and expr and make a more general function

Parsing Chains

```
chain p op f s =
      case p s of
            Just (n,c:s') | c == op ->
                  case chain p op f s' of
                        Just (m,s'') -> Just (f n m,s'')
                        Nothing      -> Just (n,c:s')
            r -> r
```

```
expr, term :: Parser Expr
expr  = chain term '+' Add
term  = chain factor '*' Mul
```

Factor?

```
factor :: Parser Expr
factor = num
```

Parentheses

- So far no parentheses
- Expressions look like
  - 23
  - 23+5*17
  - 23+5*(17+23*5+3)

a factor can be a parenthesized expression again

Factor?

```
factor :: Parser Expr
factor ('(':s) =
   case expr s of
      Just (a, ')':s1) -> Just (a, s1)
      _                -> Nothing

factor s = num s
```

Reading an Expr

```
Main> readExpr ”23”
Just (Num 23)
Main> readExpr ”apa”
Nothing
Main> readExpr ”23+17”
Just (Add (Num 23) (Num 17))
```

```
readExpr :: String -> Maybe Expr
readExpr s = case expr s of
                    Just (a,””) -> Just a
                    _           -> Nothing
```

Alternative number parsing

```
number :: Parser Int
number (c:s) | isDigit c = Just (n,s')
                    where n = read $ takeWhile isDigit (c:s)
                          s' = dropWhile isDigit s

number _                 = Nothing
```

Summary

- Parsing becomes easier when
  - Failing results are explicit
  - A parser also produces the *rest* of the string
- Case expressions
  - To look at an intermediate result
- Higher-order functions
  - Avoid copy-and-paste programming

The Code (1)

```
readExpr :: String -> Maybe Expr
readExpr s = case expr s of
                    Just (a,"") -> Just a
                    _           -> Nothing

expr, term :: Parser Expr
expr  = chain term '+' Add
term  = chain factor '*' Mul

factor :: Parser Expr
factor ('(':s) =
   case expr s of
      Just (a, ')':s1) -> Just (a, s1)
      _                -> Nothing
factor s = num s
```

The Code (2)

```
chain p op f s =
    case p s of
        Just (n,c:s2) | c == op ->
            case chain p op f s2 of
                Just (m,s3) -> Just (f n m,s3)
                Nothing     -> Just (n,c:s2)
        r -> r
```

```
number :: Parser Int
number (c:s) | isDigit c = Just (digits 0 (c:s))
number _                 = Nothing

digits :: Int -> String -> (Int,String)
digits n (c:s) | isDigit c = digits (10*n + digitToInt c) s
digits n s                 = (n,s)
```

Testing readExpr

```
prop_ShowRead :: Expr -> Bool
prop_ShowRead a =
    readExpr (show a) == Just a
```

```
Main> quickCheck prop_ShowRead
Falsifiable, after 3 tests:
-2*7+3
```

negative numbers?

Fixing the Number Parser

```
number :: Parser Int
number (c:s) | isDigit c = Just (digits 0 (c:s))
number ('-':s)           = fmap neg (number s)
number _                 = Nothing
```

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f (Just x) = Just (f x)
fmap f Nothing = Nothing
```

```
neg :: (Int,String) -> (Int,String)
neg (x,s) = (-x,s)
```

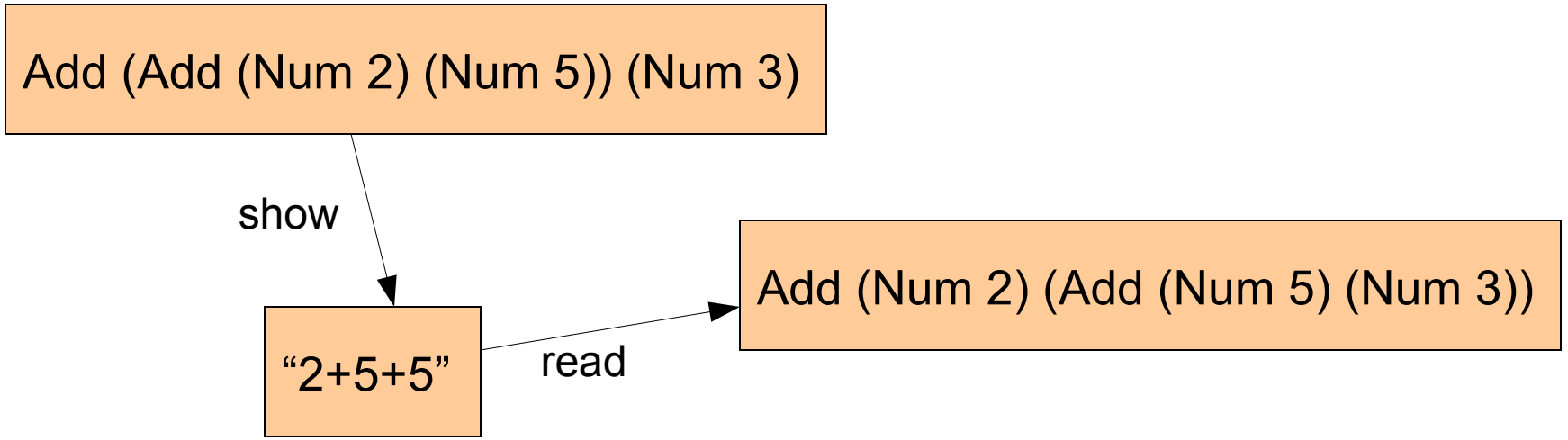
This function is actually overloaded. Works for many types besides Maybe.

Testing again

Main> *quickCheck prop_ShowRead*
Falsifiable, after 5 tests:
2+5+3

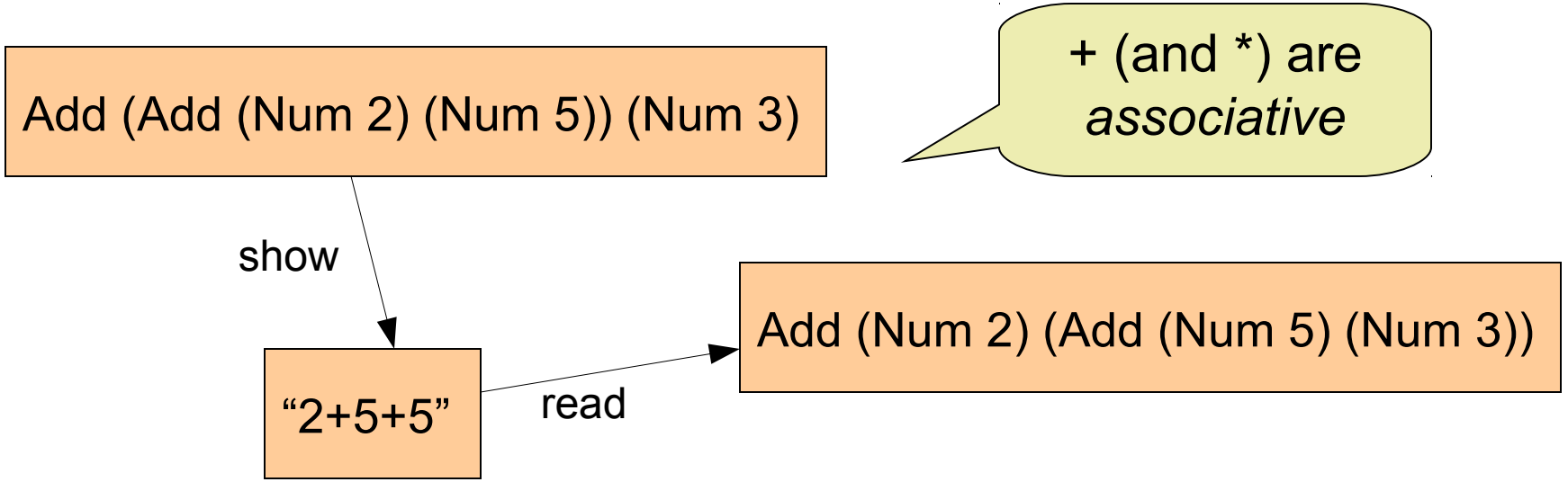
Testing again

Main> *quickCheck prop_ShowRead*
Falsifiable, after 5 tests:
2+5+3



Testing again

Main> *quickCheck prop_ShowRead*
Falsifiable, after 5 tests:
2+5+3



Fixing the Property (1)

The result does not have to be *exactly* the same, as long as the *value* does not change.

```
prop_ShowReadEval :: Expr -> Bool
prop_ShowReadEval a =
    fmap eval (readExpr (show a)) == Just (eval a)
```

Main> *quickCheck prop_ShowReadEval*
OK, passed 100 tests.

Fixing the Property (2)

The result does not have to be *exactly* the same, only after rearranging associative operators

```
prop_ShowReadAssoc :: Expr -> Bool
prop_ShowReadAssoc a =
    readExpr (show a) == Just (assoc a)
```

non-trivial recursion and pattern matching

```
assoc :: Expr -> Expr
assoc (Add (Add a b) c) = assoc (Add a (Add b c))
assoc (Add a b)          = Add (assoc a) (assoc b)
assoc (Mul (Mul a b) c)  = assoc (Mul a (Mul b c))
assoc (Mul a b)          = Mul (assoc a) (assoc b)
assoc a                  = a
```

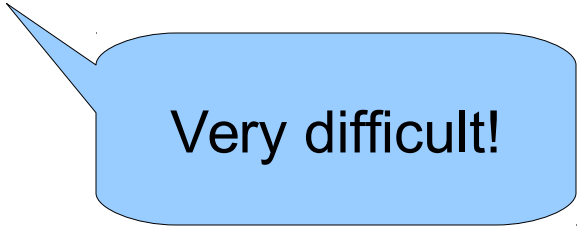
(study this definition and what this function does)

```
Main> quickCheck prop_ShowReadAssoc
OK, passed 100 tests.
```

Properties about Parsing

- We have checked that readExpr correctly processes anything produced by showExpr

- Is there any other property we should check?
  - What can still go wrong?
  - How to test this?



Summary

- Testing a parser:
  - Take any expression,
  - convert to a String (show),
  - convert back to an expression (read),
  - check if they are the same
- Some structural information gets lost
  - associativity!
  - use “eval”
  - use “assoc”