Parsing Expressions

Original slides by Koen Lindström Claessen

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
```

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

built-in show function produces ugly results

- 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
```

- How to parse?

Recursive strategy?

- Our usual strategy (divide and conquer):
  - Split the input in parts
  - Recursively process the parts
  - Combine the results of the recursive calls

- But how do we know where to split the string?

  Examples: “(1+2)*3” “1+2*3”

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

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

One or more terms with '+' between them

We're currently ignoring parentheses

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

Expression Grammar

A formal way of expressing the structure of expressions (EBNF):

- expr ::= term “**+**” ... “**+**” term
- term ::= factor “*****” ... “*****” factor
- factor ::= number

Representing parsers

- A parser receives a *string*, and either *fails* or returns a *value* plus the *rest* of the string

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

Parsing Numbers

String -> Maybe (Int,String)

```
number :: Parser Int
```

Main> *number ”23”*
Just (23, ””)
Main> *number ”117junk”*
Just (117, ”junk”)
Main> *number ”apa”*
Nothing
Main> *number ”23+17”*
Just (23, ”+17”)

how to implement?

Parsing Numbers

```
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 :: Int -> String

or

read :: Read a => a -> String

```
import Data.Char
```

at the top of your file

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

```
isPDF :: FilePath -> Bool
isPDF s = case reverse (take 4 (reverse s)) of
                ".pdf" -> True
                _      -> False
```

Note: cases must have same indentation

Parsing Numbers

```
number :: Parser Int
```

a *case* expression

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

Main> *num ”23”*
Just (Num 23, ””)
Main> *num ”apa”*
Nothing
Main> *num ”23+17”*
Just (Num 23, ”+17”)

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+17mumble”*
Just (Add (Num 23) (Num 17), ”mumble”)

Parsing Expressions

start with a term?

is there a + sign?

can we parse *another* expr?

```
expr :: Parser Expr
expr s1 = case term s1 of
           Just (a,s2) -> case s2 of
                            '+':s3 -> case expr s3 of
                                        Just (b,s4) -> Just (Add a b, s4)
                                        Nothing     -> Just (a,s2)
                            _      -> Just (a,s2)
           Nothing     -> Nothing
```

Next, define the term parser

Parsing Terms

```
term :: Parser Expr
term s1 = case factor s1 of
            Just (a,s2) -> case s2 of
                             '*':s3 -> case term s3 of
                                         Just (b,s4) -> Just (Mul a b, s4)
                                         Nothing     -> Just (a,s2)
                             _      -> Just (a,s2)
            Nothing     -> Nothing
```

just **copy** the code from expr and make some **changes**!

NO!!

Parsing Chains

```
chain :: Parser a -> Char -> (a->a->a) -> Parser a
```

```
chain p op f s1 =
  case p s1 of
    Just (a,s2) -> case s2 of
                     c:s3 | c == op -> case chain p op f s3 of
                                         Just (b,s4) -> Just (f a b, s4)
                                         Nothing     -> Just (a,s2)
                     _              -> Just (a,s2)
    Nothing     -> Nothing
```

argument op

recursion

argument f

argument p

a *higher-order function*

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

Factor?

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

Parentheses

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

a factor can be a parenthesized expression again

Expression Grammar

– expr ::= term "**+**" ... "**+**" term

– term ::= factor "*" ... "*" factor

– factor ::= number
    | "(" expr ")"

Two alternatives

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
```

Only succeed if there is no junk left

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 :: Parser a -> Char -> (a->a->a) -> Parser a
chain p op f s1 =
   case p s1 of
     Just (a,s2) -> case s2 of
                      c:s3 | c == op -> case chain p op f s3 of
                                          Just (b,s4) -> Just (f a b, s4)
                                          Nothing     -> Just (a,s2)
                      _                  -> Just (a,s2)
     Nothing     -> Nothing
```

```
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
```

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

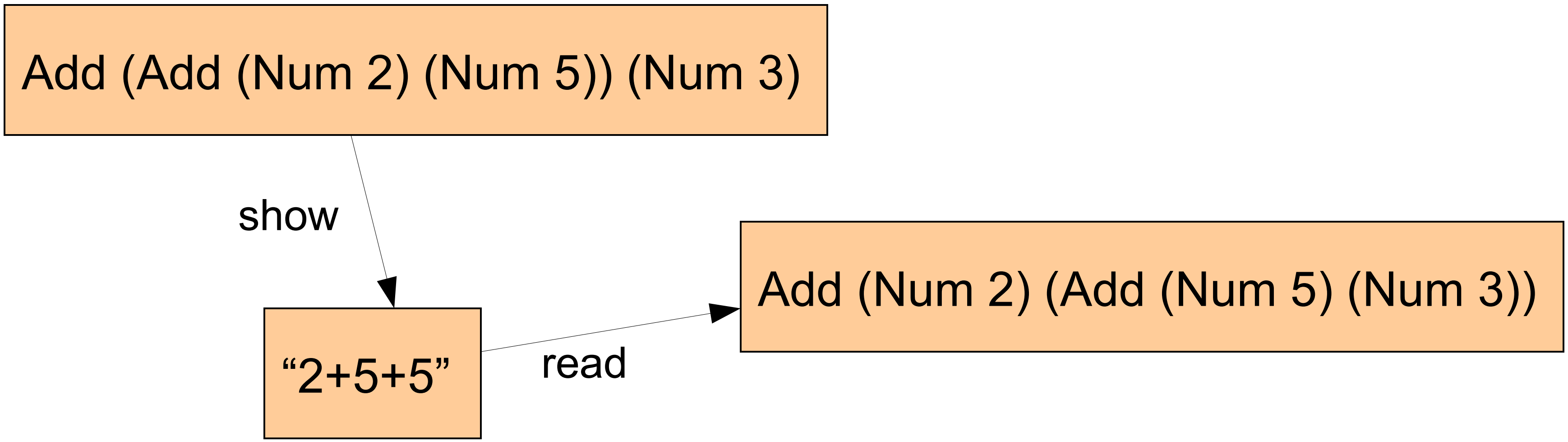
```
neg :: (Int,String) -> (Int,String)
neg (x,s) = (-x,s)
```

Testing again

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

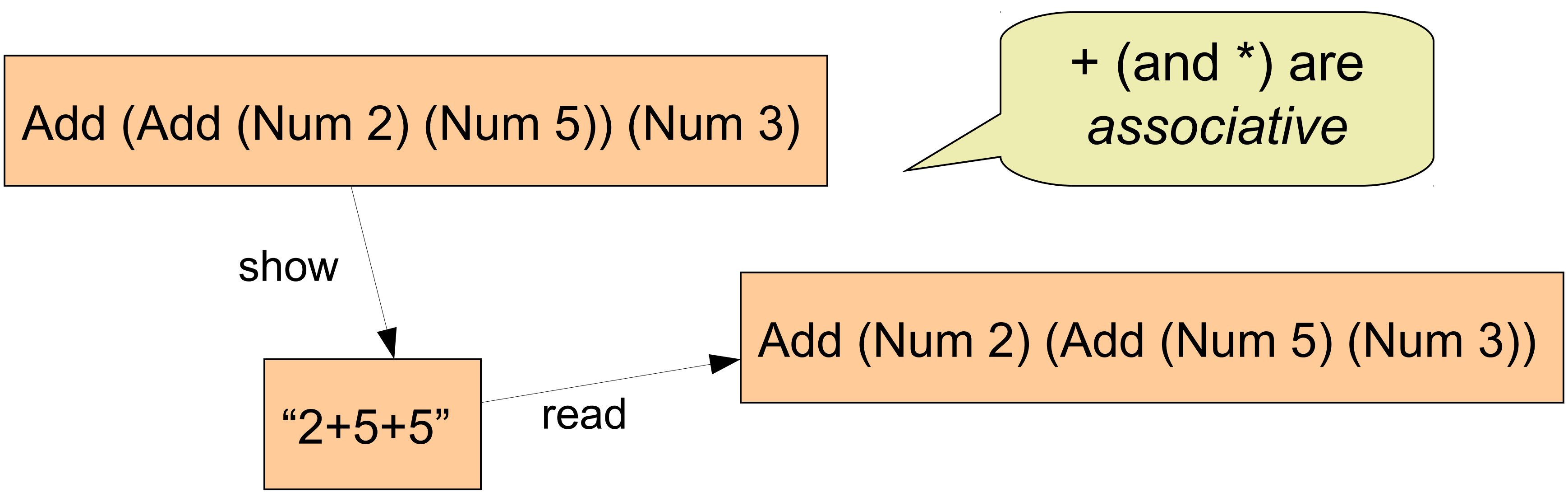
Testing again

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?

Very difficult!

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”