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

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 are three 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?

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

Parsing Numbers

```
number :: Parser Int
```

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

how to implement?

Parsing Numbers

a helper function

with an extra argument

```
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)
```

```
import Data.Char
```

at the top of your file

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”)

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*3”*
Just (Add (Num 23) (Num 17), ”*3”)

Parsing Expressions

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

start with a number?

can a parse *another* expr?

continues with a + sign?

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”

a chain of *terms* with ”+”

a chain of *factors* with ”*”

Grammar for Expressions

- Parse Expressions according to the following BNF grammar:

```
<expr>      ::= <term> | <term> "+" <expr>
<term>      ::= <factor> | <factor> "*" <term>
<factor>    ::= "(" <expr> ")" | <number>
```

If you have not seen BNF before:

- don’t get confused, this is not Haskell
- ask for a refund on your Batchelor degree

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

a factor

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

term

a ”*” sign

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)
```

Refactoring the Parser: First Attempt

Many operations in our Parser can be made **more general**

- more reuse, less clutter

Here we refactor the definition into

- **Basic building blocks** for parsers (dependent on the type of our Parser)
- **Combinators**: building blocks for making parsers from other parsers (independent of the type of Parser)

The Big Picture

Refactor/generalise

`ReadExpr.hs`

- "Brute force" parser.
- Big ugly case expressions.
- Minimal reuse.

Refactor

`RefactoredParser`

- Few basic bulding blocks (datatype dependent)
- Parser "Combinators"

`RefactoredReadExpr`

- A few lines of code

Alternative approach

`Parsing.hs`

- Parser as an instance of Monad

`ReadExprMonadic`

- A few lines of code

Next lecture

Basic parsers (1)

```
sat :: (Char -> Bool) -> Parser Char
sat p (c:cs) | p c = Just (c,cs)
sat _  _           = Nothing


char :: Char -> Parser Char
char c = sat (==c)
```

As before

Parses a specific character

Basic parsers (1)

```
succeed :: a -> Parser a
succeed a s = Just (a,s)


fail :: Parser a
fail s = Nothing
```

Always succeeds in producing an a without consuming any of the input string

Always fails

Not so useful on their own – but will be handy in combination with other parsers…

Basic parsers (2)

```
(+++) :: Parser a -> Parser a -> Parser a
(p +++ q) s =
   listToMaybe [x | Just x <- [p s, q s]]
```

the successful parses

try parsing both with p and with q

return the first successful parse

Example

```
Main> (number +++ succeed 42) "123xxx"

Just (123,"xxx")

Main> (number +++ succeed 42) "xxx"

  Just (42,"xxx")

Main> map (sat isDigit +++ char '{')
         ["{hello", "8{hello", "hello"]

[Just ('[',"hello"),Just ('8',"[hello"),Nothing]
```

Basic parsers (2)

```
pmap :: (a -> b) -> Parser a -> Parser b
pmap f p s = case p s of
               Nothing -> Nothing
               Just (a,s') -> Just (f a ,s')
```

```
Main> pmap digitToInt (sat isDigit) "1+2"
Just (1,"+2)")
```

Parse one thing after another

Several ways to parse one thing then another, e.g.

- parse first thing, discard result then parse second thing (function **(>->)**)
- parse first thing, parse and discard a second thing, return result of the first **(<-<)**
- parse the first thing and then parse a second thing in a way which depends on the value of the first (function **(>*>)**)
- parse a sequence of as many things as possible (functions **zeroOrMore, oneOrMore**)

Parse one thing after another

```
(>->) :: Parser a -> Parser b -> Parser b

(p >-> q) s = case p s of
                          Nothing       -> Nothing
                          Just (_, s’) -> q s’
```

throws away the result of first parse

```
Main> (char '('  >-> sat isDigit) "(1+2)"
Just ('1',"+2)")
```

Parse one thing after another

```
>*> :: Parser a -> (a -> Parser b) -> Parser b

(p >*> f) s = case p s of
                          Nothing -> Nothing
                          Just (a,rest) -> f a rest
```

```
Main> (sat isDigit >*> \a -> sat (>a))  "12xxx"
Just ('2',"xxx")
Main> (sat isDigit >*> \a -> sat (>a))  "10xxx"
Nothing
```

Parse one thing after another

```
>*> :: Parser a -> (a -> Parser b) -> Parser b

(p >*> f) s = case p s of
                          Nothing -> Nothing
                          Just (a,rest) -> f a rest
```

```
Main> (sat isDigit >*> \a -> sat (>a))  "12xxx"
Just ('2',"xxx")
Main> (sat isDigit >*> \a -> sat (>a))  "10xxx"
Nothing
```

Derived Parsers

```
(>->)  :: Parser a -> Parser b -> Parser b
p >-> q = p >*> \_ -> q

<-< :: Parser a -> Parser b -> Parser a
(p <-< q) s = p >*> \a -> q >-> succeed a
```

(as before) throws away the result of first parser

throws away the result of second parser

```
Main> (sat isDigit <-< char '>' )  "2>xxx"
Just ('2',"xxx")
```

Parsing sequences to lists

```
(<:>) :: Parser a -> Parser [a] -> Parser [a]
p <:> q = p >*> \a -> pmap (a:) q

zeroOrMore,oneOrMore :: Parser a -> Parser [a]

zeroOrMore p = oneOrMore p +++ succeed []
oneOrMore p  =  p  <:> zeroOrMore p
```

```
Main> zeroOrMore (sat isDigit) "1234xxxx"
Just ("1234","xxxx")
Main> zeroOrMore (sat isDigit) "x1234xxx"
Just ("","x1234xxx")
Main> (char '@' <:> oneOrMore (char '+')) "@++xxx"
Just ("@++","xxx“)
```

Building Parsers with Parsers

```
number :: Parse Int
number = pmap read (oneOrMore (sat isDigit))
```

read can't fail here since it is only applied to a list of digits!

```
num :: Parse Expr
number = pmap Num number
```

`Int -> Expr`

`Parse Int`

Building Parsers with Parsers

```
expr, term, factor :: Parser Expr

expr = chain term   '+' Add
term = chain factor '*' Mul

factor = (char '(' >-> expr <-<  char ')')
         +++ num
```

(as before)

```
chain :: Parser a -> Char -> (a -> a -> a) -> Parser a
chain p c f =
         pmap (foldr1 f) (p <:> afterFirst)
         where afterFirst = zeroOrMore (char c >-> p)
```

Summary (Refactoring)

- By using higher-order programming we can build **parser combinators** (functions that build parsers from parsers) from which specific parsers can be quickly written.
- Next time: Turning parser combinators into a Monads