Parsing

Zhenjiang Hu

National Institute of Informatics

May 31, June 7, June 14, 2010

Outline I

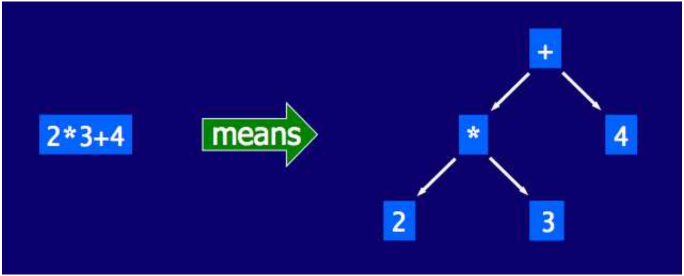
Outline

What is a Parser?

A *parser* is a program that analyses a piece of text to determine its syntactic structure.

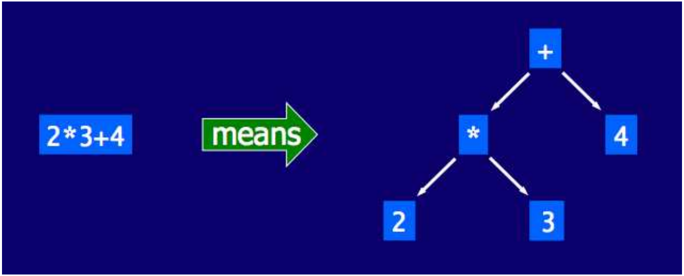
What is a Parser?

A *parser* is a program that analyses a piece of text to determine its syntactic structure.



What is a Parser?

A *parser* is a program that analyses a piece of text to determine its syntactic structure.



Almost every real life program uses some form of parser to pre-process its input.

The Parser Type

Parsers can naturally be viewed as functions: taking a string and returning some form of tree.

type Parser = String → Tree

The Parser Type

Parsers can naturally be viewed as functions: taking a string and returning some form of tree.

type Parser = String $\rightarrow$ Tree

However, a parser might not require all of its input string, so we also return any unused input:

type Parser = String $\rightarrow$ (Tree, String)

The Parser Type

Parsers can naturally be viewed as functions: taking a string and returning some form of tree.

type Parser = String → Tree

However, a parser might not require all of its input string, so we also return any unused input:

type Parser = String → (Tree, String)

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

type Parser = String → [(Tree, String)]

The Parser Type

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

Final Definition

type Parser a = String $\rightarrow$ [(a, String)]

The Parser Type

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

Final Definition

type Parser a $=$ String $\rightarrow$ [(a, String)]

Remark

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

```
item :: Parser Char
item = \inp -> case inp of
                 []     -> []
                 (x:xs) -> [(x,xs)]
```

Basic Parsers

The parser failure always fails:

failure

```
failure :: Parser a
failure = \inp -> []
```

Basic Parsers

The parser return v always succeeds, returning the value v without consuming any input:

return v

```
return   :: a -> Parser a
return v = \inp -> [(v,inp)]
```

Basic Parsers

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

choice

```
(+++)   :: Parser a -> Parser a -> Parser a
p +++ q = \inp -> case p inp of
                    []         -> parse q inp
                    [(v,out)] -> [(v,out)]
```

Basic Parsers

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

choice

```
(+++)  :: Parser a -> Parser a -> Parser a
p +++ q = \inp -> case p inp of
                    []         -> parse q inp
                    [(v,out)] -> [(v,out)]
```

The function parse applies a parser to a string:

choice

```
parse :: Parser a -> String -> [(a,String)]
parse p inp = p inp
```

Examples

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

Load the library

```
> ghci Parsing0
```

Examples

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

Load the library

> ghci Parsing0

test item

*Main> parse item ""

Examples

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

Load the library

> ghci Parsing0

test item

*Main> parse item ""
[]

*Main> parse item “abc”

Examples

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

Load the library

> ghci Parsing0

test item

*Main> parse item ""
[]

*Main> parse item "abc"
[('a',"bc")]

Examples

test failure

*Main> parse failure "abc"

Examples

test failure

*Main> parse failure "abc"
[]

Examples

test failure

*Main> parse failure "abc"
[]

test return

*Main> parse (return 1) "abc"

Examples

test failure

*Main> parse failure "abc"
[]

test return

*Main> parse (return 1) "abc"
[(1,"abc")]

Examples

test choice

*Main> parse (item +++ return 'd') "abc"

Examples

test choice

*Main> parse (item +++ return 'd') "abc"
[('a',"bc")]

Examples

test choice

```
*Main> parse (item +++ return 'd') "abc"
 [('a',"bc")]

*Main> parse (failure +++ return 'd') "abc"
```

Examples

test choice

```
*Main> parse (item +++ return 'd') "abc"
 [('a',"bc")]

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

Outline

1. Basic Parsers
2. Composing Parsers
   - Monad
   - Parser Monad
3. Derived Primitive Parsers
4. Parsing Arithmetic Expressions
5. Handling Lexical Issues
6. Parsing Lambda Expressions

Monad

A monad is a way to structure computations in terms of values and sequences of computations using those values.

- Monadic computations $m\ t$: a value of type $m\ t$ is a computation resulting in a value of type t.
- return: For any value, there is a computation which "does nothing" and produces a computation.

  ```
  return :: a -> m a
  ```

- bind: `x` $>>=$ `f` "runs" the computation x, pass the result to f which computes a new computation.

  ```
  (>>=) :: m a -> (a -> m b) -> m b
  ```

The Monad Laws

Three important laws:

- result is sort of an identity unit of bind

```
return v >>= f          = f v
x        >>= return     = x
```

- bind is associative.

```
(x >>= f) >>= g == x >>= (\v -> f v >>= g)
```

Do Notation

A sequence of computations of the form

```
p1 >>= \a1 ->
p2 >>- \a2 ->
...
pn >>- \an ->
f a1 a2 ... an
```

is denoted using the do notation as follows.

do notation

```
do a1 <- p1
   a2 <- p2
   ...
   an <- pn
   f a1 a2 ... an
```

Parser Monad

Parser *a* denotes a parsing computation resulting in a value of *a*.

Parser Monad

```
newtype Parser a  =  P (String -> [(a,String)])

instance Monad Parser where
  return v = P (\inp -> [(v,inp)])
  p >>= f  = P (\inp -> case parse p inp of
                        []         -> []
                        [(v,out)] -> parse (f v) out)
```

Note that with Haskell we can use `newtype` to redefine the type of `Parser a` with a data constructor P.

Sequencing

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

Example

```
p :: Parser (Char,Char)
p  = do x <- item
        _ <- item
        y <- item
        return (x,y)
```

Outline

Derived Primitives

sat: Parsing a character that satisfies a predicate.

sat

```
sat  :: (Char -> Bool) -> Parser Char
sat p = do x <- item
           if p x then
              return x
            else
              failure
```

Derived Primitives

digit and char: Parsing a digit and specific charaterers.

digit, char

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

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

Derived Primitives

digit and char: Parsing a digit and specific charaterers.

digit, char

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

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

Exercise

Give a definition of `isDigit`.

Derived Primitives

Exercise

Give definitions of lower, upper, letter, alphanum for parsing a lower char, an upper char, a letter, and a letter or a number, respectively.

Derived Primitives

many: Parsing a parser zero or more times.

many

```
many   :: Parser a -> Parser [a]
many p = many1 p +++ return []
```

Derived Primitives

many: Parsing a parser zero or more times.

many

```
many   :: Parser a -> Parser [a]
many p = many1 p +++ return []
```

many1: Parsing a parser one or more times.

many1

```
many1  :: Parser a -> Parser [a]
many1 p = = do v <- p
               vs <- many p
               return (v:vs)
```

Derived Primitives

string: Parsing a specific string of characters.

string

```
string        :: String -> Parser String
string []      = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)
```

Note that in the do notation, we usually write p to denote _ <- p.

Examples

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)
```

Examples

For instance: from a string:

running the parser

```
*Main> parse p "[1,2,3,4]"
[("1234","")]

*Main> parse p "[1,2,3,4"
[]
```

Outline

1. Basic Parsers
2. Composing Parsers
3. Derived Primitive Parsers
4. Parsing Arithmetic Expressions
5. Handling Lexical Issues
6. Parsing Lambda Expressions

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 $+$.

Arithmetic Expressions

Formally, the syntax of such expressions is defined by the following context free grammar:

grammar of the expression

```
expr   := term '+' expr | term

term   := factor '*' term | factor

factor := digit | '(' expr ')'

digit  := '0' | '1' | ... | '9'
```

Arithmetic Expressions

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

grammar of the expression: revised

```
expr   := term ('+' expr | empty)

term   := factor ('*' term | empty)

factor := digit | '(' expr ')'

digit  := '0' | '1' | ... | '9'
```

Note that empty denotes the empty string.

Arithmetic Expressions

It is now easy to translate the grammar into a parser that evaluates expressions (or simply construct an expression tree), by simply rewriting the grammar rules using the parsing primitives. That is, we have:

expr

```
expr :: Parser Int
expr  = do t <- term
           do char '+'
              e <- expr
              return (t + e)
            +++ return t
```

Arithmetic Expressions

term

```
term :: Parser Int
term  = do f <- factor
           do char '*'
              t <- term
              return (f * t)
          +++ return f
```

Arithmetic Expressions

factor

```
factor :: Parser Int
factor  = do d <- digit
             return (digitToInt d)
           +++ do char '('
                  e <- expr
                  char ')'
                  return e
```

Arithmetic Expressions

Finally, if we define

```
eval    :: String -> Int
eval xs = fst (head (parse expr xs))
```

Arithmetic Expressions

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
```

Arithmetic Expressions

Exercise

Extend the expression parser to allow the use of subtraction and division, based upon the following extensions to the grammar:

```
expr ::= term ('+' expr | '-' expr | empty)

term ::= factor ('*' term | '/' term | empty)
```

Outline

1. Basic Parsers
2. Composing Parsers
3. Derived Primitive Parsers
4. Parsing Arithmetic Expressions
5. Handling Lexical Issues
6. Parsing Lambda Expressions

Handling Lexical Issues

Traditionally, a string to be parsed is not supplied directly to a parser, but is first passed through a lexical analysis phase that breaks the string into a sequence of tokens.

In fact, lexers are just simple parsers, and they can be built using our parser primitives and combinators.

White-space

We begin by definition a parser that consumes while-space from the beginning of a string, with a dummy value () returned as result:

space

```
space :: Parser ()
space  =  do many (sat isSpace)
             return ()
  where
    isSpace x = (x == ' ') || (x == '\n')
```

Comment

We write a parser that consumes a comments from "–" to the end of the line, with a dummy value () returned as result:

comment

```
comment :: Parser ()
comment =  do string "--"
              many (sat (\x -> x /= '\n'))
              return ()
```

Junk

This parser is to repeatedly consumes white-spaces and comments until no more remain:

junk

```
junk :: Parser ()
junk = do many (space +++ comment)
          return ()
```

Tokens

The parser token is to ignore spacing and comments and consume a complete token according to a given parser.

token

```
token   :: Parser a -> Parser a
token p =  do junk
              v <- p
              junk
              return v
```

Useful Token Instances

identifier

```
identifier :: Parser String
identifier =  token ident

ident :: Parser String
ident =  do x  <- lower
            xs <- many alphanum
            return (x:xs)
```

Useful Token Instances

natural number

```
natural :: Parser Int
natural =  token nat

nat :: Parser Int
nat =  do xs <- many1 digit
          return (read xs)
```

Useful Token Instances

integer

```
integer :: Parser Int
integer =  token int

int :: Parser Int
int =  do char '-'
          n <- nat
          return (-n)
       +++ nat
```

Useful Token Instances

symbol

```
symbol     :: String -> Parser String
symbol xs =  token (string xs)
```

Useful Token Instances

Examples

```
*Main> parse identifier "   int a[10];"
[("int","a[10];")]

*Main>  parse natural "\n 23   int a[10];"
[(23,"int a[10];")]

*Main>  parse integer "\n -23   int a[10];"
[(-23,"int a[10];")]

*Main> parse (symbol "let")  "let x = ..."
[("let","x = ...")]

*Main> parse natural "\n  -23 inta[10]"
[]
```

Outline

Lambda Calculus

Recall in our previous course on Lambda calculus.

Lambda Expression

$$
\begin{array}{lll}
x \in \mathit{Ide} & & \{\ \text{identifier}\ \} \\
e \in \mathit{Exp} & & \{\ \text{Lambda expression}\ \} \\
 & & \\
e ::= x & & \{\ \text{variable}\ \} \\
\quad\ \mid e_1\ e_2 & & \{\ \text{application}\ \} \\
\quad\ \mid \lambda x.e & & \{\ \text{abstraction}\ \}
\end{array}
$$

For abstraction $\lambda x.e$, we often write it as `x -> e` (``\ x -> e'' in a string format)

Lambda Calculus

Examples

```
c_s = "\\ x -> \\ y -> \\ z -> x z (y z)"
c_k = "\\ x -> \\ y -> x"
c_i = "\\ x -> x"
c_b = "\\ x -> \\ y -> \\ z -> x (y z)"
c_c = "\\ x -> \\ y -> \\ z -> x z y"
c_fix = "(\\ x -> x x) (\\ x -> x)"
```

Lambda Calculus

Lambda Expression in Haskell

```
type Ide = String
data Exp = Var Ide
         | App Exp Exp
         | Lam Ide Exp
```

Refine the Lambda Syntax

Like in arithmetic expression, we have to deal with left associativity of "application" operation, i.e.,

$$e_1 \;\; e_2 \;\; e_3 \;\; \ldots \;\; e_n = (((e_1 \;\; e_2) \;\; e_3) \;\; \ldots) \;\; e_n$$

Refine the Lambda Syntax

Like in arithmetic expression, we have to deal with left associativity of "application" operation, i.e.,

$$e_1 \; e_2 \; e_3 \; \ldots \; e_n = (((e_1 \; e_2) \; e_3) \; \ldots) \; e_n$$

So we refine the syntax as follows:

refined syntax

```
lexp ::= atom lexp | atom

atom ::= var
      |  \x -> lexp
      |  (lexp)
```

Parser Definition

lexp

```
lexp :: Parser Exp
lexp  = chainl1 atom App

chainl1 :: Parser a -> (a->a->a) -> Parser a
chainl1 p f = do x <- p
                 rest x
 where rest x = do y <- p
                   rest (f x y)
                 +++ return x
```

Parser Definition

atom

```
atom    :: Parser Exp
atom     = var +++ lam +++ paren
var     :: Parser Exp
var      = do x <- identifier
              return (Var x)
lam     :: Parser Exp
lam      = do symbol "\\"
              x <- identifier
              symbol "->"
              e <- lexp
              return (Lam x e)
paren   :: Parser Exp
paren    = do symbol "("; e <- lexp; symbol ")"
              return e
```

Testing

Now we can define the following main function for test the parser.

```
parse_lambda :: String -> Exp
parse_lambda inp = fst (head (parse lexp inp))
```

Testing

Now we can define the following main function for test the parser.

```
parse_lambda :: String -> Exp
parse_lambda inp = fst (head (parse lexp inp))
```

Here are some test examples:

Examples

```
*Main> parse_lambda c_s
Lam "x" (Lam "y" (Lam "z" (App (App (Var "x") (Var "z"))
(App (Var "y") (Var "z")))))

*Main> parse_lambda c_fix
App (Lam "x" (App (Var "x") (Var "x"))) (Lam "x" (Var "x"))
```

About the Report

Problem

Extend the parser so that it can handle the lambda expression with local binding.

$x \in Ide$	{ identifier }
$e \in Exp$	{ Lambda expression }
$e ::= x$	{ variable }
$\mid e_1 \; e_2$	{ application }
$\mid \lambda x.e$	{ abstraction }
$\mid$ **let** $x = e$ **in** e	{ let }

Note `Exp` should be extended with a new data constructor for “let”:

```
data Exp ::= ...
          |  Let (Ide, Exp) Exp
```

About the Report

- Solve the problem in the previous slide and summarize your result in a report, which should include
  - an explanation of your solution and several running examples
  - a complete source code
    (you may revise `Parsing_Examples.lhs` based on the library `Parsing.lhs`).
- Submit your report during the class on June 28 (Mon).

Haskell Sources used in Lectures

- `Parsing0.lhs`: this is used before we introduce monad for composing parsers.
- `Parsing.lhs`: this is the parsing library used through the lecture.
- `Parsing_Examples.lhs`: several examples of using the parsing library, including parsers for arithmetic expressions and lambda expressions.