

Monadic Parsing



1

Schedule

Haskell				
Tue Nov 5 / Fri Nov 8	The State Monad	Ch. 12	Quiz 4 on Fri Lecture_Week11	PS7 due Tuesday Lecture11.hs , Lecture11'.hs , State.hs
Tue Nov 12 / Fr Nov 15	Parsing Theory (a bit); Monadic Parsing	Ch. 13		PS8 Checkpoint #1: attend office hours this week (or earlier)
Tue Nov 19 / Fri Nov 22	Parsec		Quiz 5 on Fri	5-8 min presentation in class on Friday PS8 due on Tuesday
Tue Nov 26	Property Testing; QuickCheck			
Tue Dec 3 / Fri Dec 6	TBD		Quiz 6 on Fri	Checkpoint #2: attend office hours this week (or earlier)
Tue Dec 10	Project presentations			Project due 5-8 min presentation in class

Programming in Haskell, A Milanova

2

Quiz 4

Passing state explicitly:

$Subst :: LExp \rightarrow (String, LExp) \rightarrow Int \rightarrow (LExp, Int)$

```

subst (App e1 e2) (v2,m) fresh =
  let (e1',fresh') = subst e1 (v2,m) fresh
      (e2',fresh'') = subst e2 (v2,m) fresh' in
      (App e1' e2', fresh'')
  
```

S (State → (LExp, Int))

vs. a state transformer:

$subst :: LExp \rightarrow (String, LExp) \rightarrow S.State Int LExp$

```

subst (App e1 e2) (v2,m) = do
  e1' <- subst e1 (v2,m)
  e2' <- subst e2 (v2,m)
  return (App e1' e2')
  
```

Programming in Haskell, A Milanova

3

3

Quiz 4

```

subst (Lambda v1 e) (v2,m) fresh =
  if v1 == v2
  then (Lambda v1 e, fresh)
  else let newStr = "_" ++ (show fresh)
        (e',fresh') = subst e (v1, Atom newStr) (fresh + 1)
        (e'',fresh'') = subst e' (v2,m) fresh' in
        (Lambda newStr e'', fresh'')
  
```

```

subst (Lambda v1 e) (v2,m) =
  if v1 == v2
  then (Lambda v1 e)
  else do fresh <- S.get
        let newStr = "_" ++ (show fresh)
            S.put (fresh + 1)
            e' <- subst e (v1, Atom newStr)
            e'' <- subst e' (v2,m)
        return (Lambda newStr e'')
  
```

Programming in Haskell, A Milanova

4

4

Outline

- Lexing and parsing
 - Parser type
 - Parser monad
 - Lexer primitives
-
- Recursive-descent monadic parsing

5

Lexing and Parsing

- Syntax is the form or structure of expressions, statements, and program units of a given language

Syntax of a Java **while** statement:
`while (boolean_expr) statement`

- Semantics is the meaning of expressions, statements and program units of a given language

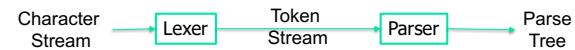
Semantics of **while** (*boolean_expr*) *statement*
Execute *statement* repeatedly (0 or more times) as long as *boolean_expr* evaluates to **true**

6

- A formal language is a set of strings (also called sentences) over a finite alphabet
- A generator is a set of rules that generate the strings in the language
- A recognizer reads input strings and determines whether they belong to the language

7

- Regular languages describe tokens (e.g., identifiers, symbols and constants)
 - Generated by a Regular Expression
 - Recognized by DFA (lexer)
- Context-free languages describe constructs of more complex constructs (e.g., expressions and statements)
 - Generated by a Context-Free Grammar
 - Recognized by a Pushdown Automaton (parser)



8

Example

Consider expression: `x1 + -123 * x2`

Token sequence: `id:x1` `+` `int:-123` `*` `id:x2`

Identifiers: `letter (letter | digit)*`

Symbols: `+, *`

Integers: `(-|ε) digit+`

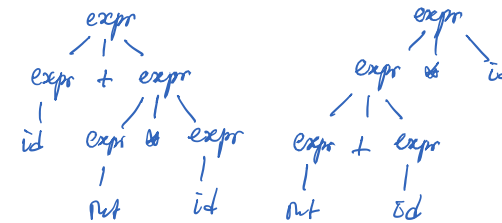
An example grammar and parse tree:

$expr ::= expr + expr \mid expr * expr \mid (expr) \mid id \mid int$
expr is a nonterminal
*+, *, id and int are terminals. These are tokens supplied by lexer.*

9

`id + int * id`

$expr ::= expr + expr \mid expr * expr \mid (expr) \mid id \mid int$



10

Parser Type

- Back to Haskell. Haskell allows us to compose lexers and parsers conveniently and parse into complex structures

- A parser takes an input string and produces (typically) some structured object, e.g., a parse tree:

```
type Parser = String -> StructuredObject
```

- A parser consumes only part of the string input. Therefore, we expand the type into a tuple of the structured object and the remainder of the input string:

```
type Parser = String -> (StructuredObject, String)
```

11

- The Parser type is polymorphic

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

- Also, it must account for parse errors and ambiguous grammars

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

- An empty list means the input string is ill-formed according to the grammar

- A non-singleton list result means there is more than one way to parse the string (into a parse tree) according to grammar

12

runParser p "ana"

- And as a last step, we'll wrap the type in a newtype declaration:

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

- Why do we need to wrap in a newtype?
- Does the type look familiar?
- Very similar to the state transformer:

```
newtype State s a = S (s -> (a,s))
runState (S f) = f
```

Programming in Haskell, A Milanova 13

13

Building a Small Parser

```
newtype Parser a = P (String -> [(a,String)])
runParser (P f) = f -- extract function to run on input
```

- A parser that parses a single character

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

```
> runParser item "abc"
[( 'a', 'bc' )]
> runParser item ""
[]
```

Programming in Haskell, A Milanova 14

14

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

- Let us now modify `item` to parse a single digit:

```
import Data.Char -- isDigit

oneDigit :: Parser Int
oneDigit = P (\s -> case s of
  [] -> []
  (x:xs) -> if isDigit x then [(read [x],xs)]
             else [])
```

```
> runParser oneDigit "abc"
[] -- fail
> runParser oneDigit "12"
[( '1', '2' )]
> runParser oneDigit "1"
[( '1', "" )]
```

Pro 15

15

Exercise

- Write a parser that reads one character and returns the negate function if character is -, id if it is + and fails otherwise

```
import Data.Char -- isDigit

oneOp :: Parser (Int -> Int)
oneOp = P (\s -> case s of
  ('-' : xs) -> [(negate, xs)]
  ('+' : xs) -> [(id, xs)]
  _ -> [])
```

```
> fst (head (runParser oneOp "-abc")) 10
> fst (head (runParser oneOp "+10")) 10
```

Programming in Haskell, A Milanova (based on notes by Stephanie Weirich) 16

16

- Now generalize: satisfy parser returns character if it matches input predicate, fails otherwise

```
import Data.Char -- isDigit, isAlpha, isUpper, etc.

satisfy :: (Char -> Bool) -> Parser Char
satisfy p = P (\s -> do -- monadic bind of []
    (x,xs) <- runParser item s
    guard (p x)
    return (x,xs))

> runParser (satisfy isAlpha) "a"
["a", ""]
> runParser (satisfy isUpper) "a"
[]
```

- We'll get back to satisfy in just a little bit.

17

Parser is a Functor

- How do we combine (i.e., compose) smaller parsers into larger ones that parse into more complex structures?
- We'll instantiate Parser as a Monad and make use of bind to combine parsers. In Haskell, we need to first instantiate Parser into a Functor and an Applicative Functor, and then Monad

```
instance Functor Parser where
-- fmap :: (a -> b) -> Parser a -> Parser b
fmap f p = P (\s -> [ (f a, s') | (a, s') <- runParser p s ])
```

18

Parser is an Applicative Functor

```
instance Applicative Parser where
-- pure :: a -> Parser a
pure x = P (\s -> [(x,s)])

-- <*> :: Parser (a->b) -> Parser a -> Parser b
pab <*> pa =
    P (\s -> ...
```

- Note: We do not need Monads to parse context free grammars, we can parse with applicative functors

19

Parser is a Monad

- We'll use monads because
 - (1) the do notation is a super convenient way to encode grammars
 - (2) we've done so much with monads already!

```
instance Monad Parser where
-- return :: a -> Parser a
return x = P (\s -> [(x,s)])

-- >>= :: Parser a -> (a -> Parser b) -> Parser b
p >>= f =
    P (\s -> [ (b,s'') | (a,s') <- runParser p s
                    , (b,s'') <- runParser (f a) s'])
```

20

- Download **Parser.hs** and **Lecture12.hs** and code as we move on
- satisfy becomes a bit easier now that we can use the do notation

```
import Data.Char -- isDigit, isAlpha, isUpper, etc.

sat :: (Char -> Bool) -> Parser Char
sat p = do c <- item -- parse character with item
         if p c then P (λs -> [(c,s)]) else P (λs -> [])
           return c           empty

> runParser (sat isAlpha) "a"
[(('a', "")]
> runParser (sat isUpper) "a"
[]
```

21

Basic Primitives

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

lower :: Parser Char
lower = sat isLower

upper :: Parser Char
upper = sat isUpper

alphanum :: Parser Char
alphanum = sat isAlphaNum

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

22

```
-- parses three characters, skipping the middle
-- e.g., runParser three "abcde" yields [(('a','c'),"de")]
three :: Parser (Char,Char)
three = do x <- item
          item
          z <- item
          return (x,z)

-- matches a string
-- runParser (string "ana") "ana123" yields [(("ana","123")]
-- runParser (string "ana") "a123" yields []
string :: String -> Parser String
string [] = return []
string (x:xs) = do char x
                  string xs
                  return (x:xs)
```

23

- Our goal is to encode grammars
- A typical regular grammar:
 - letter (letter | digit)* -- identifier
 - (-|ε) digit+ -- integer
- A typical context-free grammar:
 - expr ::= term + expr | term
 - term ::= factor * term | factor
 - factor ::= identifier | integer
- We know how to sequence. We need to alternate as well!
- Also, we need a way to encode Kleene star and Kleene plus

24

Alternating

- <|> runs parser p and if parse succeeds, <|> returns result without evaluating q. If p fails, <|> runs q.

```
instance Alternative Parser where
  -- empty :: Parser a
  empty = P (\s -> [])

  -- <|> :: Parser a -> Parser a -> Parser a
  p <|> q =
    P (\s -> case runParser p s of
      [] -> runParser q s
      xs -> xs)
```

Programming in Haskell, A Milanova

25

25

many $\times \sim x^*$
some $\times \sim x^+$

```
> runParser empty "abc"
[]
> runParser (item <|> return 'd') "abc"
[('a', 'bc')]
> runParser (empty <|> return 'd') "abc"
[('d', 'abc')]
> runParser (many digit) "123abc" -- Kleene star
[('123', 'abc')]
> runParser (many digit) "abc" -- Kleene star
[('', 'abc')]
> runParser (some digit) "123abc" -- Kleene plus
[('123', 'abc')]
> runParser (some digit) "abc" -- Kleene plus
[]
```

Programming in Haskell, A Milanova

26

26

Tokens

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

nat :: Parser Int -- natural num constant, almost
nat = do xs <- some digit -- digit+
         return (read xs)

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

Programming in Haskell, A Milanova

27

27

- We want to parse $x1 + -123 * x2$
Into token sequence identifier,+,integer,*,identifier
- But space shouldn't matter, $x1 + -123 * x2$
Is same token sequence identifier,+,integer,*,identifier

```
space :: Parser () -- returns nothing
space = do many (sat isSpace)
         return ()
```

```
token :: Parser a -> Parser a
token p = do space
            res <- p
            return res
```

Programming in Haskell, A Milanova

28

28

Lexer Primitives

- Lexer primitives parse tokens

```
identifier :: Parser String -- identifier
identifier = token ident -- filters out space

natural :: Parser Int -- natural number
natural = token nat

integer :: Parser Int
integer = token int

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

29

```
> runParser identifier "1234"
> runParser identifier "a1234"
> runParser natural "1234"
> runParser natural "-1234"
> runParser integer "1234abc"
> runParser integer "-1234abc"
> runParser (symbol "+") "+abc"
```

30

Exercise

```
-- parses a list of integers
-- runParser nats " [1, 2, 3 ] " yields [( "[1,2,3]", " ") ]
-- runParser nats " [1, ] " yields []
nats :: Parser String
nats = ...
```


31

Outline

- Lexing and parsing
- Parser type
- Parser monad
- Lexer primitives

- Recursive-descent monadic parsing


32



- So far we built parsers (lexers) for regular grammars
- How about context-free grammars?
- An expression grammar:
 $expr ::= expr + expr \mid expr * expr \mid (expr) \mid identifier \mid integer$
- What's the problem with this grammar?

Programming in Haskell, A Milanova 33

33



- One way to disambiguate:
 $expr ::= term + expr \mid term$
 $term ::= factor * term \mid factor$
 $factor ::= (expr) \mid identifier \mid integer$
- And easily build a monadic parser! E.g., the `expr` Parser:


```

expr :: Parser String
expr = do t <- term
         symbol "+"
         e <- expr
         return (t ++ " + " ++ e)
         <|> term
  
```

- What are some issues with this grammar?

Programming in Haskell, A Milanova 34

34



- Finish up coding the parsers, then add the start production with end-of-input:

```

start :: Parser a -> Parser a
start p = do res <- p
           symbol "$"
           return res
  
```


- We have

```

> runParser (start expr) "x1 + -123 * x2$"
[("x1 + -123 + x2", "")]
  
```

Programming in Haskell, A Milanova 35

35



Recursive-Descent Parsing

- Monadic parsers are recursive-descent parsers
- In recursive-descent there is a procedure for each nonterminal, in our example it is `expr`, `term`, and `factor`
- E.g., `expr` parser corresponds to `expr` nonterminal:

```

expr :: Parser String
expr = do t <- term
         symbol "+"
         e <- expr
         return (t ++ " + " ++ e)
         <|> term
  
```

Programming in Haskell, A Milanova 36

36

Recursive-Descent Parsing

- To parse a nonterminal, call (descend) corresponding procedure.
- Right-hand-side of nonterminal forms body of procedure: parser tries one production, then next, and so on
 - E.g., $expr ::= term + expr \mid term$ first tries $expr ::= term + expr$; if it fails, it tries $expr ::= term$
- Parsing right-hand-side means calling procedures and consuming terminals in turn
 - E.g., $term + expr$ first calls $term$, then consumes $+$ then calls $expr$

```
expr :: Parser String
expr = do t <- term
         symbol "+"
         e <- expr
         return (t ++ " " ++ e)
<|> term
```

Programming in Haskell, A Milan

37

37

Recursive-Descent Parsing

- Parsing an alternative may fail in which case recursive-descent backtracks (i.e., puts consumed input back) and tries next alternative
 - E.g., $x1*x2*x3$ Parser tries $term + expr$ but fails on $+$. It backtracks to the beginning then tries $term$

```
expr :: Parser String
expr = do t <- term
         symbol "+"
         e <- expr
         return (t ++ " " ++ e)
<|> term
```

Programming in Haskell, A Milanova

38

38

Backtracking Example

- Consider grammar
 - $S ::= c A d$
 - $A ::= a b \mid a$and input string cad
- What happens with our grammar:
 - $expr ::= term + expr \mid term$
 - $term ::= factor * term \mid factor$
 - $factor ::= (expr) \mid identifier \mid integer$
- Why can't we use this "better" disambiguation:
 - $expr ::= expr + term \mid term$
 - $term ::= term * factor \mid factor$
 - $factor ::= (expr) \mid identifier \mid integer$

39

39

- For some grammars recursive-descent does not backtrack

- We can rewrite our grammar one more time:


```
expr ::= term term_tail
term_tail ::= + term term_tail | ε
term ::= factor factor_tail
factor_tail ::= * factor factor_tail | ε
factor ::= ( expr ) | identifier | integer
```

- A classic LL(1) grammar meaning parser can predict the production by looking at just one token of lookahead
- The monadic parser will consume and backtrack from at most one token, thus, a lot more efficient

Programming in Haskell, A Milanova

40

40



■ We can rewrite our grammar one more time:
expr ::= term term_tail
term_tail ::= + term term_tail | ε
becomes
expr ::= term term_tail
term_tail ::= + expr | ε
becomes
expr ::= term (+ expr | ε)

```
expr :: Parser String
expr = do t <- term
        do symbol "+"
           e <- expr
           return (t ++ " + " ++ e)
<|> return t
```

Programming in Haskell, Ch. 11, slide 41