Monadic Parsing

1

Schedule

			State.hs	
Tue Nov 12 / Fr Nov 15	Monadic Parsing	Ch. 13	Lecture_Week12 Parser.hs, Lecture12.hs	PS8, Data.hs, State.hs, Parser.hs, Lexer.hs, Infer.hs, Ps8.hs **Checkpoint #1: attend office hours this week (or earlier)**
Tue Nov 19 / Fri Nov 22	More parsing and Parsec		Lecture_Week13	**5-8 min presentation in class on Friday**
Tue Nov 26	Property Testing; QuickCheck		Quiz 5 on Tue	PS8 due on Tuesday
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**

2

Outline

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

3

Recursive-Descent Parsing

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

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

- To parse a nonterminal, call (i.e., descend) into corresponding routine

4

Recursive-Descent Parsing

- Production right-hand-side forms body of procedure. Parser tries one production, then next, and so on
  - E.g., *expr* ::= *term* + *expr* | *term*

  Parser first tries to parse *term + expr*; if fail, it tries *term*
- Parsing right-hand-side means calling procedures for nonterminals and consuming terminals, in turn
  - E.g., *term* + *expr* first calls *term*, then (if success) consumes + then calls *expr*

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


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 trying to match +. It backtracks to the beginning and tries *term*

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

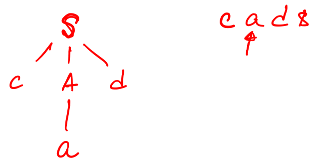

Backtracking Example

- Consider grammar

  S ::= c *A* d

  A ::= a b | a

and input string cad$



c a d $

- Is there backtracking with this grammar?

  expr ::= *term* + *expr* | *term*

  term ::= *factor* * *term* | *factor*

  factor ::= (*expr*) | identifier | integer

a * b * c * d $

- What we can and what we cannot parse with recursive-decent?

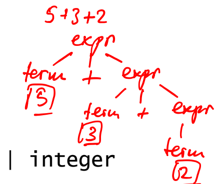
5+3+2

- This grammar can be parsed:

  expr ::= *term* + *expr* | *term*

  term ::= *factor* * *term* | *factor*

  factor ::= (*expr*) | identifier | integer



- How about this "better" disambiguation of original expr grammar:

  expr ::= *expr* + *term* | *term*

  term ::= *term* * *factor* | *factor*

  factor ::= (*expr*) | identifier | integer

LEFT RECURSION IS BAD.

- For some grammars recursive-descent does not backtrack
- We can rewrite our grammar like this:

  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 does not consume and backtrack; `symbol` does a “lookahead” of at most one token and decides. Thus, a lot more efficient than previous parser!

- 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
```


Evaluation

- Look at simplified version of grammar:

  expr ::= integer *ttail*
  ttail ::= + integer *ttail* | - integer *ttail* | ε

- Let us write a parser for this grammar. It parses into strings:

```
expr :: Parser String
expr =
 do i <- integer
    tt <- ttail
    return ((show i)++tt)
```

```
ttail :: Parser String
ttail =
 do
```


- Look at simplified version of grammar:

  expr ::= integer *ttail*
  ttail ::= + integer *ttail* | - integer *ttail* | ε

- Now let’s look at parse tree of expression 5 - 3 - 2

expr ::= integer *ttail*
ttail ::= + integer *ttail* | - integer *ttail* | ε

- We want to build a parser that also evaluates expression. First try:

```
expr :: Parser Int
expr =
 do i <- integer
    tt <- ttail
    return (i + tt)
```

```
ttail :: Parser Int
ttail = do symbol "+"
           i <- integer
           tt <- ttail
           return (i + tt)
        <|> do symbol "-"
               i <- integer
               tt <- ttail
               return (i - tt)
        <|> return 0
```


expr ::= integer *ttail*
ttail ::= + integer *ttail* | - integer *ttail* | ε

- It's wrong! Parser evaluates 5 - 3 - 2 incorrectly. Here is how:

- Problem of course is that grammar is right-recursive and tree tilts to the right while we are doing a bottom-up evaluation

expr ::= integer *ttail*
ttail ::= (+ | -) integer *ttail* | ε

- Parse of our running example 5 - 3 - 2 with lists of tuples:

- One way to fix

expr ::= integer *ttail*
ttail ::= (+ | -) integer *ttail* | ε

```
type IntOp = Int -> Int -> Int

expr :: Parser Int
expr = do i <- integer
          -- tt :: [(IntOp,Int)] i.e., flattens tree
          tt <- ttail
          let comb :: Int -> (IntOp, Int) -> Int
              comb x (op,y) = op x y
          -- foldl tt list around i
          return (foldl comb i tt)
```

expr ::= integer *ttail*
ttail ::= (+ | -) integer *ttail* | ε

- Parser `ttail` parses into a more complex structure, a list of tuples:

```
symbol' :: Parser IntOp
symbol' = do symbol "+"
             return (+)
          <|> do symbol "-"
                 return (-)
```

```
ttail :: Parser [(IntOp,Int)]
ttail = do s <- symbol'
           i <- integer
           tt <- ttail
           return ((s,i):tt)
        <|> return []
```


17

- Another way to fix, using a State transformer

expr ::= integer *ttail*
ttail ::= (+ | -) integer *ttail* | ε

```
expr :: Parser Int
expr = do i <- integer
          -- tt :: Parser (S.State Int ())
          -- encodes transform of rest-of-tree
          tt <- ttail
          return (S.runState tt i)
```


18

expr ::= integer *ttail*
ttail ::= (+ | -) integer *ttail* | ε

- Parse of our running example 5 - 3 - 2 with state transformer:

expr
Integer 5 — ttail [\pr -> pr-3-2]
Integer 3 — ttail [\pr -> pr-2]
Integer 2 — ttail [\pr -> pr]
ε

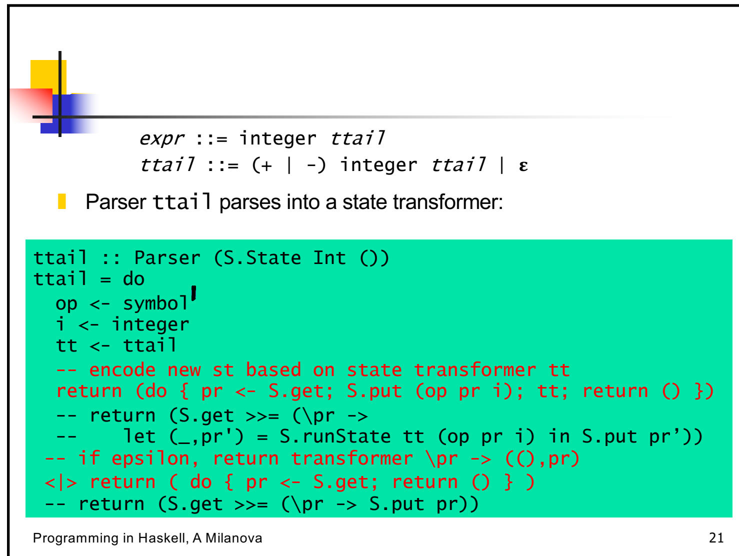
19

- Another way to fix, using a State transformer

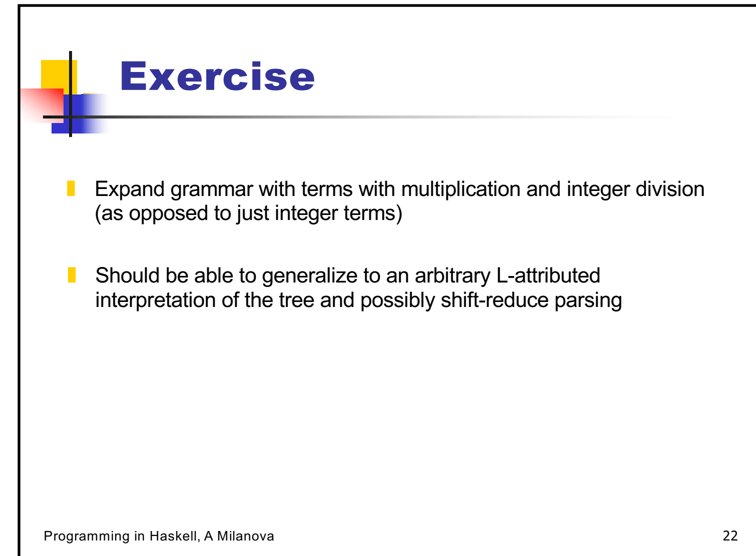
expr ::= integer *ttail*
ttail ::= (+ | -) integer *ttail* | ε

```
expr :: Parser Int
expr = do i <- integer
          -- tt :: Parser (S.State int ())
          -- encodes transform of rest-of-tree
          tt <- ttail
          let (_,i') = S.runState tt i
          return i'
```

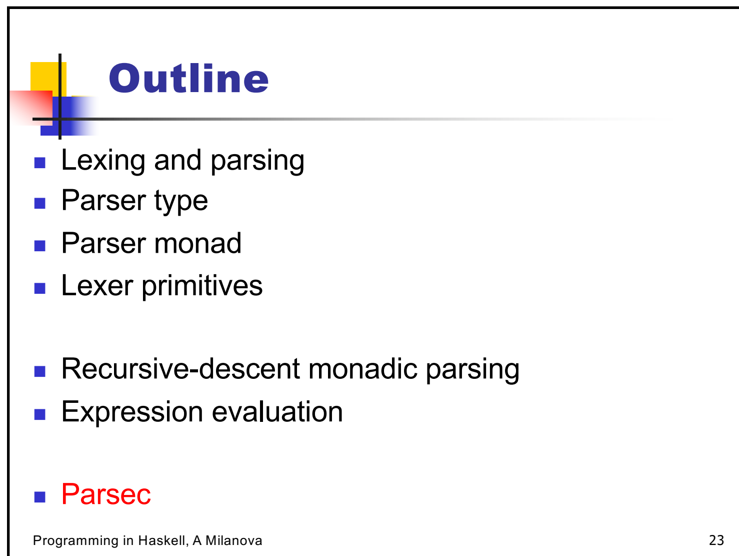

20



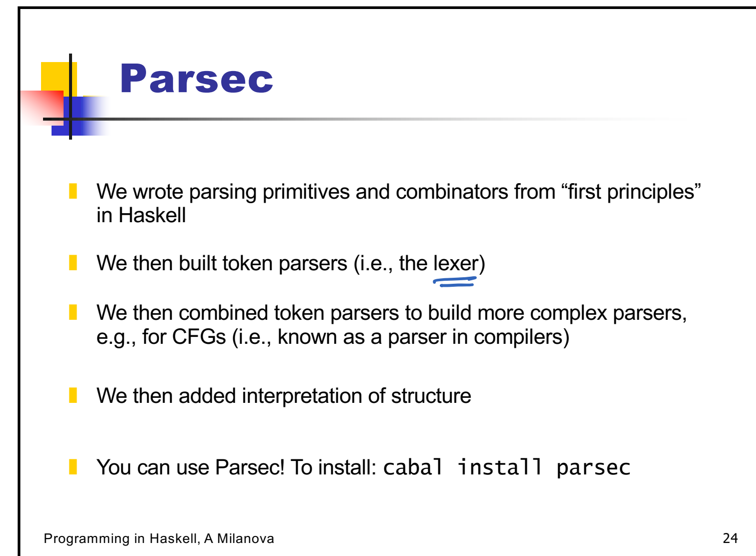
21



22



23



24

Parser Types

Using Text.Parsec

- `ParsecT` is most general:

```
type ParsecT s u m a
-- Input stream s, typically String
-- User state u wrapped in a monad, e.g., Int
-- Monad to wrap output m, e.g., [], Maybe
-- Output type a, e.g., Char
```

- `Parsec` instantiates `ParsecT` with `Identity` monad, i.e., no wrapping:

```
type Parsec s u = ParsecT s u Identity
```


- To run a `Parsec` parser:

```
> :t runParser
… =>
Parsec s u a -> u -> SourceName -> s -> (Either ParseError a)
> runParser digit () "" "9abc"
Right '9'
> runParser digit () "" "abc"
Left (line 1, column 1):
unexpected "a"
expecting digit
```

- An even simpler `parse`, no user state:

```
> parse digit "" "9abc"
Right '9'
> parse digit "" "abc"
Left ... -- Error
```


- Remember our `Parser` a type:

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

- We can define a similar specialized parser in terms of `Parsec`:

```
type Parser a = Parsec String () a

runParser' :: Parser a -> String -> Either ParseError a
runParser' p = runParser p () ""

> runParser' digit "987"
Right '9'
```


Character Primitives

- Character primitives are defined in terms of `tokenPrim` since we don't have access to the stream (it is enclosed into a monad)
- The item primitive:

```
item :: Parser Char
item = tokenPrim showChar nextPos Just
     where
       showChar x = "'" ++ [x] ++ "'"
       nextPos pos x xs = updatePosChar pos x

> runParser' item "abc"
Right 'a' -- just as item we defined last week
```

- The `sat` primitive. Note: there is `satisfy` in Text.Parsec.

```
sat :: (Char -> Bool) -> Parser Char
sat p
   = tokenPrim showChar nextPos testChar
   where
     showChar x = "'" ++ [x] ++ "'"
     nextPos pos x xs = updatePosChar pos x
     testChar x = if p x then Just x else Nothing

> runParser' (sat isDigit) "9abc"
Right '9' -- just as sat we defined last week
```

- We can now derive all lexer functions: `natural`, `integer`, `symbol`, and `identifier`; just like we did last week with the `Parser` a type. Then we can derive more complex parsers

Parsec Lexer

- But we don't have to! We can use the Text.Parsec.Token
- `emptyDef` is one of the default language definitions; another one is `haskellDef`; these are records
- We can redefine some features, e.g., identifier starting character and remaining identifier characters:

```
import Text.Parsec
import Text.Parsec.Pos (updatePosChar, updatePosString)
import qualified Text.Parsec.Token as P
import Text.Parsec.Language (emptyDef)

langDef = emptyDef { P.identStart = lower
                   , P.identLetter = alphaNum
                   }
```


- Then we can instantiate the lexer:

```
-- The lexer
lexer = P.makeTokenParser langDef

identifier :: Parsec String u String
identifier = P.identifier lexer
natural = P.natural lexer
integer = P.integer lexer
symbol = P.symbol lexer
```

- Typically, we just reuse one of the language definitions; if we need a special lexer, we define one from scratch using `tokenPrim`

Parsec Expression Parser

- So, let's build a parser for our LL(1) grammar using those lexer primitives. We'll parse back into a String

expr ::= integer *ttail*
ttail ::= (+ | -) integer *ttail* | ε

```
expr :: Parser String
expr = do i <- integer
          t <- ttail
          return ((show i) ++ t)

ttail :: Parser String
ttail = do …
```

SAME AS BEFORE

Evaluation

- Evaluation takes advantage of user state. Need the Parsec type

```
expr ::= integer ttail
ttail ::= (+ | -) integer ttail | ε
```

```
expr :: Parsec String Integer Integer
expr = do i <- integer
```

```
ttail :: Parsec String Integer Integer
ttail = do op <- symbol
```

```
> runParser expr 0 "" "5-3-2"
```


Exercise

- Let's evaluate expression using this slightly different grammar:

```
expr ::= integer + expr | integer – expr | integer
```

```
expr' :: Parsec String (Integer,IntOp) Integer
expr' = do i <- integer
```


Some Notes

- Important: Parsec's combinator **do not** backtrack by construction (unlike Parser a we wrote which did backtrack)
- Need to force backtracking with try
- Extend the parsers with more productions and handling of eof