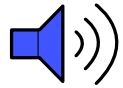
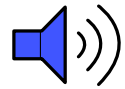
Lecture 12 – Parsec 

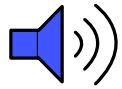
Outline:

- Brief review of BNF grammar and parsing
- Parsing with combinators
- Creating parsers using Parsec combinators

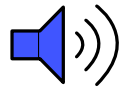
Grammar, syntax, and parsing



- Parsing is the processing of recognizing a string by breaking it down to a set of symbols and analyzing each one against the grammar of the language.
- A grammar describes how to form strings from a language's alphabet that are valid according to the language's syntax.
- A parser checks whether a string belongs to a language and transforms the string into a data structure such as a parse tree.
- A parser is implemented based on a grammar that defines the syntax of a language.

BNF Grammar 

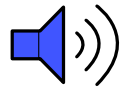
- BNF grammar is a type of context-free grammar that is commonly used to define programming language syntax.
- BNF grammar consists of 4 components: tokens (or terminals), non-terminals, production rules, and a start symbol.
  - Tokens are indivisible unit of syntax such as an identifier, keyword, operators, and literals.
  - Non-terminals such as expressions and statements are defined with tokens and other non-terminals.
  - Production rules are equations that define the derivation of non-terminals.
  - A start symbol is a unique non-terminal that corresponds to the entire program.

An example of BNF grammar 

- Arithmetic expressions

```
1  <expr>     ::= <expr> '+' <term>
2               |  <expr> '-' <term>
3               |  <term>
4
5  <term>     ::= <term> '*' <fact>
6               |  <term> '/' <fact>
7               |  <fact>
8
9  <fact>     ::= '(' <expr> ')'
10              |  <integer>
11
12 <integer>  ::= <digit>
13              |  <integer> <digit>
14
15 <digit>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<expr>, <term>, <fact>, <integer>, <digit> are non-terminals while +, −, /, ∗, (,) and 0 – 9 are tokens. <expr> is the root symbol.

An example of EBNF grammar 

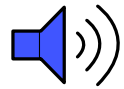
- Arithmetic expressions in EBNF grammar

```
1  <expr>     ::= <term> {('+' | '-') <term>}
2
3  <term>     ::= <fact> {('*' | '/') <fact>}
4
5  <fact>     ::= '(' <expr> ')'
6               |  <integer>
7
8  <integer>  ::= <digit> {<digit>}
9
10 <digit>    ::= [0-9]
```

{<digit>} means zero or more repetitions of <digit>
[0-9] means a digit between 0 and 9.

('+' | '-')

means choosing between '+' and '-'.

An example of parsing 

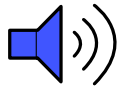
- Parsing an string can return an abstract syntax tree:

```
1 data Exp = Plus Exp Exp
2          | Minus Exp Exp
3          | Times Exp Exp
4          | Div Exp Exp
5          | Const Integer
```

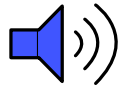
- Parsing "(1 + 2) * 3" results in the following value.

```
1 Times (Plus (Const 1) (Const 2)) (Const 3)
```

- Parsing "(1 + 2) * 3" can also directly evaluate it to 9.

Construct a parser 

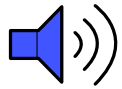
- A parser can be constructed from a parser-generator such as Lex, Yacc, Bison, JavaCC, Jison by providing a BNF phrase grammar (with a separate Lexical grammar).
- A parser can also be written directly using a parser library such as Parsec.
- Parsec has a collection of combinators that can be used to construct a LL(k) parser (Left-to-right, Leftmost derivation), where k is the number tokens of lookahead when parsing a sentence.

Construct a parser 

- A LL parser is a top-down parser that can be implemented with a deterministic push-down automaton.
  Given $\Gamma = N \cup \Sigma$, where N is the set of non-terminals and Σ is the set of tokens including end-of-file eof.

```
1  stack = [s, eof] -- s is the start symbol
2
3  parse stack
4    | stack == [eof] = True
5    | otherwise =
6
7        let x = head stack
8        in if x `elem` sigma  -- x is a token
9              then if readInput == x
10                   then parse $ tail stack
11                   else False
12             else let a = rules x -- rule x ::= a
13                  in parse $ a ++ tail stack
```

  The crucial problem is to select the correct production rule to replace non-terminal x on top of the stack.

Example of parsing 

- Given the BNF grammar below where E is the start symbol.

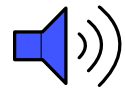
```
E ::= '(' E '+' E ')'
E ::= 'i'
```

- To parse the following input,

```
(i+i)
```

we apply the following sequence of rules

```
Rule  Stack              Stack-action          Input-action
1     [E eof]            pop E, push (E+E)
      [( E + E ) eof]    pop '('               read '('
2     [E + E ) eof]      pop E, push 'i'
      [i + E ) eof]      pop 'i'               read 'i'
      [+ E ) eof]        pop '+'               read '+'
2     [E ) eof]          pop E, push 'i'
      [i ) eof]          pop 'i'               read 'i'
      [) eof]            pop ')'               read ')'
      [eof]
```

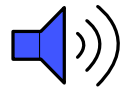
Example of parsing – Count i's 

- Given the BNF grammar below where E is the start symbol.

```
E ::= '(' E '+' E ')'
E ::= 'i'
```

- Think of a parser as a function from input String to an output that contains the result of the parsing and the remaining input not consumed by the parser.

```
type Parser a = String    -- input string
              -> (a,      -- result of parsing
                  String) -- remaining string
-- E :: = 'i'
p2 :: Parser Int              -- count the # of i's
p2 ('i':rest) = (1, rest)     -- successful and return 1
p2 input      = (0, input)    -- unsuccessful
```

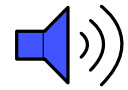
Example of parsing – Count i's 

- Given the BNF grammar below where E is the start symbol.

```
E ::= '(' E '+' E ')'
E ::= 'i'
```

- We quickly run into problems.

```
type Parser a = String   -- input string
              -> (a,     -- result of parsing
                  String) -- remaining string
-- E ::= 'i'
p2 :: Parser Int
p2 ('i':rest) = (1, rest) -- successful and return 1
p2 input      = (0, input) -- unsuccessful

-- E ::= '(' E '+' E ')'
p1 :: Parser Int
p1 ('(' : rest) = let (a, rest') = p1 rest'
            -- or  let (a, rest') = p2 rest'
            -- which one do we call?
p1 input = (0, input)
```

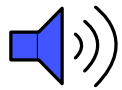
Example of parsing – Count i's 

- Given the BNF grammar below where E is the start symbol.

```
E ::= '(' E '+' E ')' | 'i'
```

- Use $\langle|\rangle$ to encode the choice between two parsers.

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

(<|>) :: Parser a -> Parser a -> Parser a
p = p1 <|> p2  -- if p1 fails to parse, then try p2

-- E ::= 'i'
p2 :: Parser Int
p2 ('i':rest) = (1, rest) -- successful and return 1

-- E ::= '(' E '+' E ')'
p1 :: Parser Int                -- too complicated already
p1 ('(' : rest) = let (a1, rest1) = p rest
                  in case rest1 of
                   ('+' : rest2) ->
                      let (a2, rest3) = p rest2
                      in case rest3 of
                          (')', rest4) -> (a1 + a2, rest4)
```

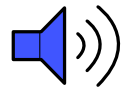
Example of parsing – Count i's 

- Given the BNF grammar below where E is the start symbol.

```
E ::= '(' E '+' E ')' | 'i'
```

- Make Parser a state monad.

```
newtype Parser a = Parser {runParser :: String -> (a, String)}

char :: Char -> Parser Char

p = p1 <|> p2

-- E ::= 'i'
p2 :: Parser Int
p2 = char 'i' >> return 1 -- one i

-- E ::= '(' E1 '+' E2 ')'
p1 :: Parser Int
p1 = do char '('
        a1 <- p  -- # of i's in E1
        char '+'
        a2 <- p  -- # of i's in E2
        char ')'
        return $ a1 + a2
```

Example of parsing – Count i's 

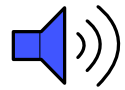
- Given the BNF grammar below where E is the start symbol.

```
E ::= '(' E '+' E ')' | 'i'
```

- Run p: unwrap its function, apply it to input, extract result.

```
newtype Parser a = Parser {runParser :: String -> (a, String)}

char :: Char -> Parser Char

p = p1 <|> p2

p2 = char 'i' >> return 1

p1 = do char '('
        a1 <- p
        char '+'
        a2 <- p
        char ')'
        return $ a1 + a2

run p x = fst $ runParser p x

run p "(i+(i+i))" -- returns 3
```

Example of parsing – Max depth of parenthesis



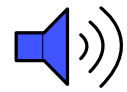
- Given the BNF grammar below where E is the start symbol.

```
E ::= '(' E '+' E ')' | 'i'
```

- p parse input to find the max depth of parenthesis.

```
newtype Parser a = Parser {runParser :: String -> (a, String)}

char :: Char -> Parser Char

p = p1 <|> p2

p2 = char 'i' >> return 0 -- depth is 0

-- E ::= '(' E1 '+' E2 ')'
p1 = do char '('
        a1 <- p  -- depth of parenthesis in E1
        char '+'
        a2 <- p  -- depth of parenthesis in E2
        char ')'
        return $ (max a1 a2) + 1

run p "(i+(i+i))" -- returns 2
```

Example of parsing – Abstract syntax tree

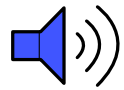


- Given the BNF grammar below where E is the start symbol.

```
E ::= '(' E '+' E ')' | 'i'
```

- p returns a Node value – $p :: Parser\ Node$

```
newtype Parser a = Parser {runParser :: String -> (a, String)}

data Node = Plus Node Node | I

p = p1 <|> p2

p2 = char 'i' >> return I

-- E ::= '(' E1 '+' E2 ')'
p1 = do char '('
        a1 <- p  -- Node for E1
        char '+'
        a2 <- p  -- Node for E2
        char ')'
        return $ Plus a1 a2

run p "(i+(i+i))" -- Plus I (Plus I I)
```

Example of parsing – XML tags 

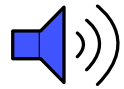
- XML is the start symbol; {x} means zero or more copies of x.

```
    XML ::= OpenTag { XML } EndTag
OpenTag ::= '<' Name '>'
 EndTag ::= '<' '/' Name '>'
```

- parse an XML string with matching open and end tags.

```
data XML = Tag String [XML]

openTag = do {char '<'; name <- word; char '>'; return name}

endTag name = string $ "</" ++ name ++ ">"

xml = do name <- openTag      -- name of open tag
         content <- many xml
         endTag name          -- matching end tag
         return $ Tag name content

-- new combinators:
--   string :: String -> Parser ()
--   many   :: Parser a -> Parser [a]
--   word   :: Parser String
```

Example of parsing – XML tags

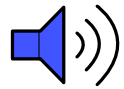


- parse an XML string with matching open and end tags.

```
1  data XML = Tag String [XML]
2
3  openTag = do {char '<'; name <- word; char '>'; return name}
4
5  endTag name = string $ "</" ++ name ++ ">"
6
7  xml = do name <- openTag      -- name of open tag
8           content <- many xml
9           endTag name           -- matching end tag
10          return $ Tag name content
11
12 --    many    :: Parser a -> Parser [a]
13 --    word    :: Parser String
```

- 'string' parses a specific string and return () if successful.

```
1  string :: String -> Parser ()
2
3  string ""     = return ()            -- parse empty string
4  string (c:cs) = char c >> string cs  -- parse char one by one
```

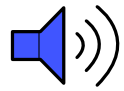
Example of parsing – XML tags 

- parse an XML string with matching open and end tags.

```
1  data XML = Tag String [XML]
2
3  openTag = do {char '<'; name <- word; char '>'; return name}
4
5  endTag name = string $ "</" ++ name ++ ">"
6
7  xml = do name <- openTag      -- name of open tag
8           content <- many xml
9           endTag name           -- matching end tag
10          return $ Tag name content
11
12 --    word    :: Parser String
```

- 'many' runs *p* zero or more times – returns p's output in a list.

```
1  many :: Parser a -> Parser [a]
2  many p = many1 p <|> return []
3
4  many1 p = do  -- many1 runs p one or more times
5     x  <- p
6     xs <- many1 p <|> return []
7     return $ x:xs
```

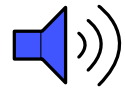
Example of parsing – XML tags 

- parse an XML string with matching open and end tags.

```
1  data XML = Tag String [XML]
2
3  openTag = do {char '<'; name <- word; char '>'; return name}
4
5  endTag name = string $ "</" ++ name ++ ">"
6
7  xml = do name <- openTag      -- name of open tag
8           content <- many xml
9           endTag name           -- matching end tag
10          return $ Tag name content
```

- 'word' parses any string consisting of one or more letters.

```
1  word :: Parser String
2  word = many1 letter
3
4  letter :: Parser Char  -- parse an alphabet
```

Example of parsing – XML tags 

- parse an XML string with matching open and end tags.

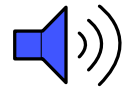
```
1  data XML = Tag String [XML]
2
3  openTag = do {char '<'; name <- word; char '>'; return name}
4
5  endTag name = string $ "</" ++ name ++ ">"
6
7  xml = do name <- openTag      -- name of open tag
8           content <- many xml
9           endTag name           -- matching end tag
10          return $ Tag name content
```

- But this can't even parse[1] "<a>".

```
1  run xml "<a></a>"
2  -- parse error at (line 1, column 5):
3  -- unexpected "/"
4  -- expecting letter
```

Open tag and end tag share <. After <a>, the parser expects another open tag or . After openTag fails, it can't try endTag since < is consumed – no backtracking.

[1] Error handling is not discussed yet.

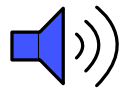
Example of parsing – XML tags 

- A parser without backtracking is LL(1).
  Use *try p* to enable backtracking for p to allow LL(k), $k \geq 2$.

```
try :: Parser a -> Parser a
```

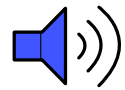
- *try p* has effect only if it runs on the left of some $\langle|\rangle$.

```
data XML = Tag String [XML]

openTag = do {char '<'; name <- word; char '>'; return name}

endTag name = string $ "</" ++ name ++ ">"

xml = do name <- try openTag -- try parse open tag
         content <- many xml
         endTag name          -- matching end tag
         return $ Tag name content

run xml "<a></a>"
-- Tag "a" []

run xml "<a><b><c></c><d></d></b></a>"
-- Tag "a" [Tag "b" [Tag "c" [],Tag "d" []]]
```

Example of parsing – Sentence 

- A sentence is a sequence of words separated by separators such as space and comma and terminated by tokens such as dot and question mark.

```
1  sentence :: Parser [String]
2  sentence = do words <- sepBy1 word separator
3                terminator
4                return words
5
6  separator :: Parser ()
7  separator = skipMany1 (space <|> char ',')
8
9  terminator = oneOf ".?!"
10 space = oneOf " \t"
11
12 --      sepBy1 :: Parser a1 -> Parser a2 -> Parser [a1]
13 --       oneOf :: [Char] -> Parser Char
14 --  skipMany1 :: Parser a -> Parser ()
15
16 run sentence "Coding is easy, but debugging is hard!"
17 -- ["Coding","is","easy","but","debugging","is","hard"]
18
19 run sentence "a, b, , c."
20 -- ["a","b","c"]
```

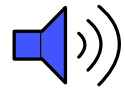
Example of parsing – Sentence 

```haskell
sentence :: Parser [String]
sentence = do words <- sepBy1 word separator
              terminator
              return words

separator :: Parser ()
separator = skipMany1 (space <|> char ',')

terminator = oneOf ".?!"
space = oneOf " \t"

--       oneOf :: [Char] -> Parser Char
--   skipMany1 :: Parser a -> Parser ()
```

- *sepBy*1 *p sep* parses one or more occurrences of *p* separated by *sep* and return a list of values returned by *p*.

```haskell
sepBy1 p sep = do
    x <- p
    xs <- many (sep >> p)
    return $ x:xs

-- zero or more occurrences of p separated by sep
sepBy p sep = sepBy1 p sep <|> return []
```

Example of parsing – Sentence

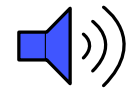


```haskell
sentence :: Parser [String]
sentence = do words <- sepBy1 word separator
              terminator
              return words

separator :: Parser ()
separator = skipMany1 (space <|> char ',')

terminator = oneOf ".?!"
space = oneOf " \t"
```

- *oneOf str* parses one of the char in *str*.
  *skipMany*1 *p* parses one or more occurences of *p* and throws away the results.

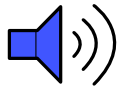
```haskell
oneOf :: [Char] -> Parser Char

skipMany1 :: Parser a -> Parser ()
skipMany1 p = many1 p >> return ()

-- parse zero or more occurrence of p and skip results.
skipMany p = many p >> return ()
```

Example of parsing – Arithmetic expression



- Parse an arithmetic expression and return abstract syntax tree.

```
1  data Exp = Plus Exp Exp
2           | Minus Exp Exp
3           | Times Exp Exp
4           | Div Exp Exp
5           | Const Integer deriving Show
6
7  expr = term `chainl1` addop -- expr ::= term {addop term}
8  term = fact `chainl1` mulop -- term ::= fact {mulop fact}
9  -- fact ::= '(' expr ')' | integer
10 fact = parens expr <|> (Const <$> integer)
11
12 addop = (char '+' >> return Plus)  -- addop ::= '+' | '-'
13     <|> (char '-' >> return Minus)
14 mulop = (char '*' >> return Times) -- mulop ::= '*' | '/'
15     <|> (char '/' >> return Div)
16
17 parens p = do { char '('; x <- p; char ')'; return x }
18 integer = read <$> many1 digit -- integer ::= digit {digit}
19
20 digit :: Parser Char -- digit ::= [0--9]
21 -- chainl1 :: Parser t -> Parser (t -> t -> t) -> Parser t
```

Example of parsing – Arithmetic expression 

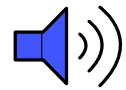
- Parse an arithmetic expression and return abstract syntax tree.

```
1  expr = term `chainl1` addop -- expr ::= term {addop term}
2  term = fact `chainl1` mulop -- term ::= fact {mulop fact}
3  -- fact ::= '(' expr ')' | integer
4  fact = parens expr <|> (Const <$> integer)
5
6  addop = (char '+' >> return Plus)  -- addop ::= '+' | '-'
7      <|> (char '-' >> return Minus)
8  mulop = (char '*' >> return Times) -- mulop ::= '*' | '/'
9      <|> (char '/' >> return Div)
10
11 parens p = do { char '('; x <- p; char ')'; return x }
12 integer = read <$> many1 digit -- integer ::= digit {digit}
13
14 digit :: Parser Char -- digit ::= [0--9]
```

- *chainl*1 *p op* folds *p* one or more times with *op* as an left associative operator.

```
1  chainl1 p op = p >>= rest
2    where rest x = do { f <- op; y <- p; rest $ f x y }
3                   <|> return x
4
5  chainl p op x = chainl1 p op <|> return x
```

Example of parsing – Arithmetic expression



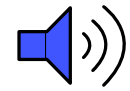
- Parse an arithmetic expression and return abstract syntax tree.

```
1  expr = term `chainl1` addop -- expr ::= term {addop term}
2  term = fact `chainl1` mulop -- term ::= fact {mulop fact}
3  -- fact ::= '(' expr ')' | integer
4  fact = parens expr <|> (Const <$> integer)
5
6  addop = (char '+' >> return Plus)  -- addop ::= '+' | '-'
7      <|> (char '-' >> return Minus)
8  mulop = (char '*' >> return Times) -- mulop ::= '*' | '/'
9      <|> (char '/' >> return Div)
10
11 parens p = do { char '('; x <- p; char ')'; return x }
12 integer = read <$> many1 digit -- integer ::= digit {digit}
13
14 digit :: Parser Char -- digit ::= [0--9]
```

- Test it

```
1  run expr "1+(2-3)*4"
2  --Plus (Const 1) (Times (Minus (Const 2) (Const 3)) (Const 4))
```

Example of parsing – Arithmetic expression



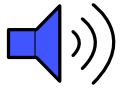
- Parse an arithmetic expression and evaluate it

```
1  expr = term `chainl1` addop -- expr ::= term {addop term}
2  term = fact `chainl1` mulop -- term ::= fact {mulop fact}
3  -- fact ::= '(' expr ')' | integer
4  fact = parens expr <|> integer
5
6  addop = (char '+' >> return (+))  -- addop ::= '+' | '-'
7      <|> (char '-' >> return (-))
8  mulop = (char '*' >> return (*)) -- mulop ::= '*' | '/'
9      <|> (char '/' >> return div)
10
11 parens p = do { char '('; x <- p; char ')'; return x }
12 integer = read <$> many1 digit -- integer ::= digit {digit}
13
14 digit :: Parser Char -- digit ::= [0--9]
```

- Test it

```
1  run expr "1+(2-3)*4"
2  -- evaluates -3
```

Parsec



- Example adopted from Chapter 16 of *Real World Haskell*.

```haskell
import Text.Parsec

type Parser = Parsec [Char] ()

csvFile :: Parser [[String]]
csvFile = endBy line newline

line = sepBy cell (char ',')
cell = many (noneOf ",\n\r")
noneOf :: [Char] -> Parser Char
```

- Runner

```haskell
run :: Show a => Parser a -> String -> IO()
run p input = putStrLn $ case parse p "" input of
        Left error -> show error
        Right x    -> show x

-- parse parser source-name input -> either error or result
parse :: Parser a -> String -> String -> Either ParseError a

run csvFile "10,one\n20,two\n"
-- [["10","one"],["20","two"]]
```