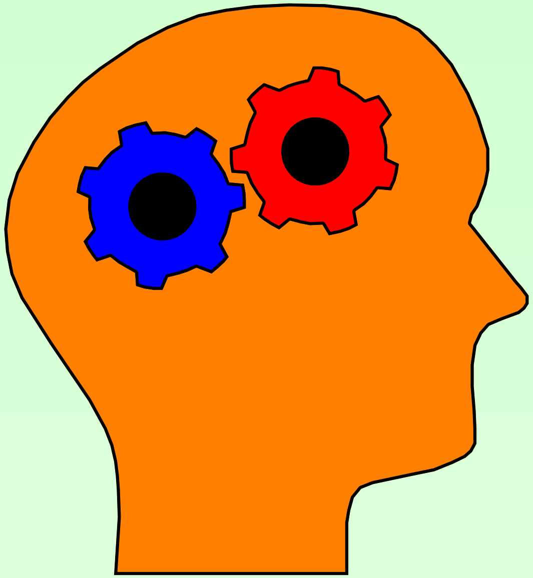
CS2104: Programming Language Concepts

Lecture 7 : Lexical and Syntax Analysis



"Monadic Parsing with Haskell"

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Language Syntax

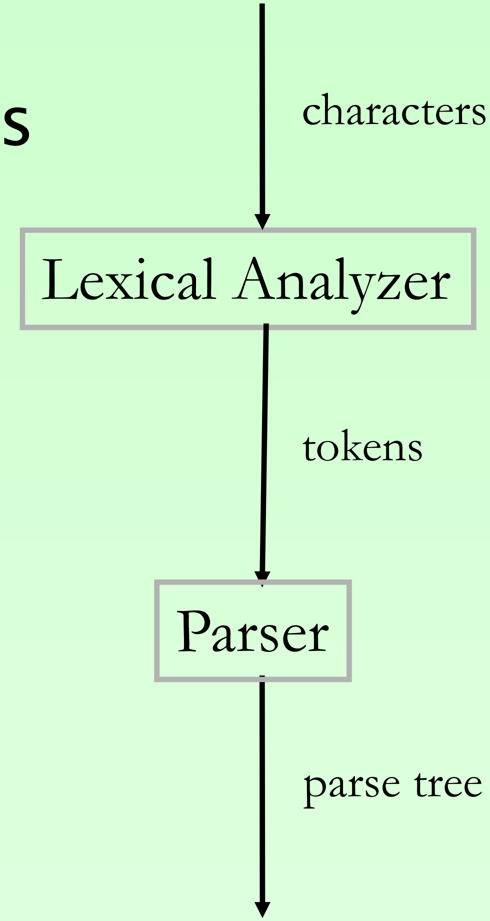
- **Language = Syntax + Semantics**
- The *syntax* of a language is concerned with the *form* of a program: how expressions, types, declarations and commands are formed.
- The *semantics* of a language is concerned with the *meaning* of a program : how programs should behave when executed on computers.

Language Syntax

- Defined by *grammar rules*
  - define how to make ‘expressions’ out of ‘words’
- For programming languages
  - sentences are called *expressions*
  - words are called *tokens*
  - grammar rules describe both tokens and statements

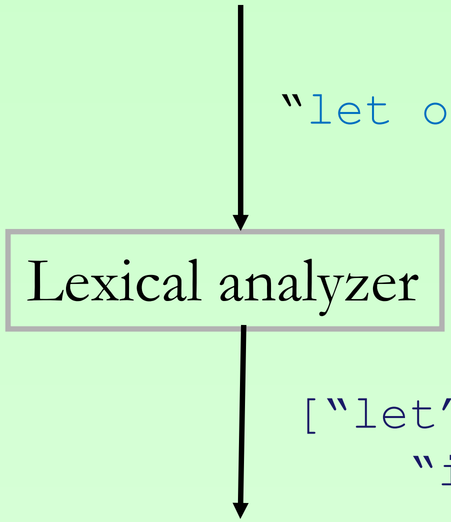
Language Syntax

- *Token* is sequence of characters
- *Expression* is sequence of tokens
- *Lexical analyzer* is a program
  - recognizes character sequence
  - produces token sequence
- *Parser* is a program
  - recognizes a token sequence
  - produces statement representation
- Statements are represented as *parse trees*



Lexical Analysis

- *Example:*



```
“let oneval=2 in oneval*3”
```

Lexical analyzer

```
[“let”, “oneval”, “=”, “2”,
    “in”, “oneval”, “*”, “3”]
```

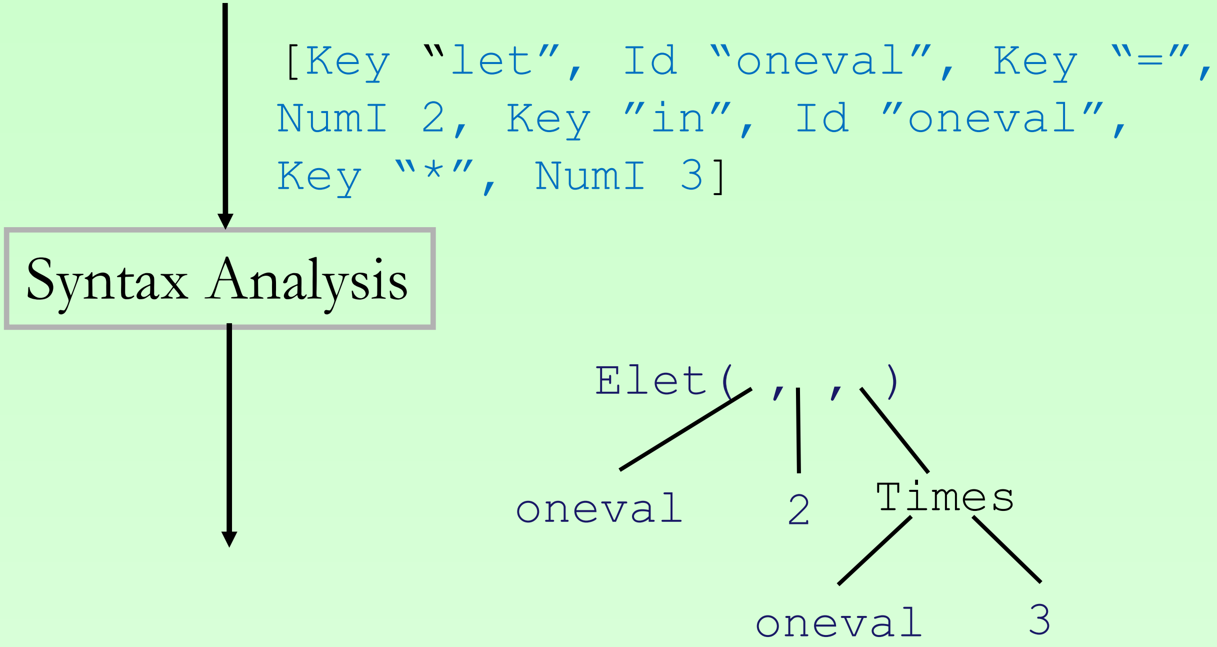
```
[Key “let”, Id “oneval”, Key “=”, NumI 2,
    Key ”in”, Id ”oneval”, Key “*”, NumI 3]
```

- Categorizing the tokens help with parsing.

```
data Token = Key String | Id String | NumI Int
```

Syntax Analysis

- Example:



```
[Key “let”, Id “oneval”, Key “=”,
NumI 2, Key ”in”, Id ”oneval”,
Key “*”, NumI 3]
```

Syntax Analysis

- We can define a parse tree using:

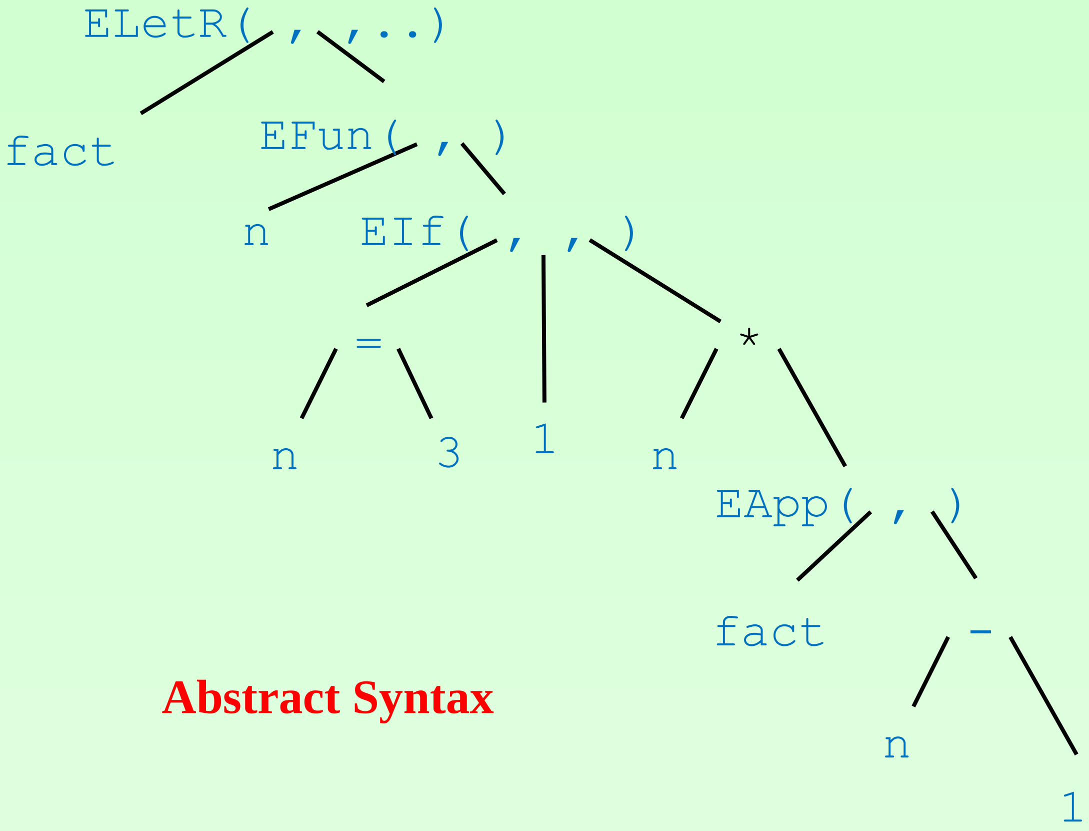
```
data Exp = Var String
         | Elet String Exp Exp
         | Enum Int
         | Times Exp Exp
         | …
```

Abstract Syntax Tree

- Used to represent code fragment as a data structure.
- Below is an example for recursive function definition in Haskell

```
let fact =
  (\ n ->
   if n = 0
     then 1
   else n*
     (fact(n-1)
  )
```

Concrete Syntax



Abstract Syntax

Context-Free Grammars

- A context-free grammar (CFG) is:
  - A set of terminal symbols T (tokens or constants)
  - A set of non-terminal symbols N
  - One (non-terminal) start symbol σ
  - A set of grammar (rewriting) rules Ω of the form

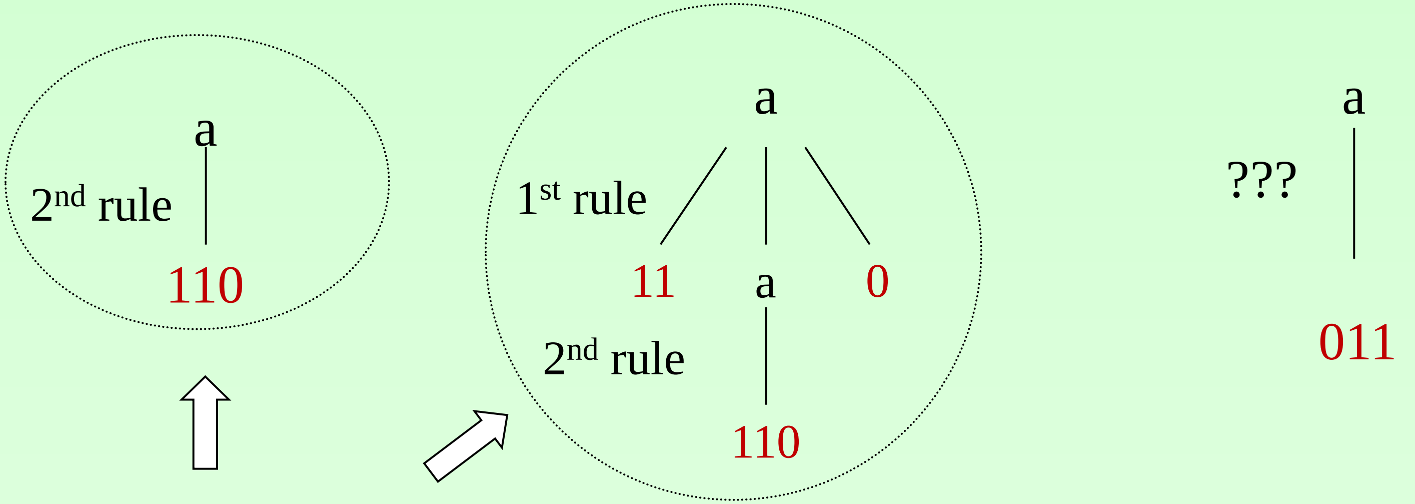
    ⟨nonterminal⟩ ::= ⟨sequence of terminals and nonterminals⟩
- Grammar rules (productions) can be used to
  - verify that an expression is legal
  - generate all possible statements
- The set of all possible statements generated by a grammar from the start symbol is called a (*formal*) *language*

An Example

- Let $N = \{\langle a\rangle\}$, $T = \{0,1\}$, $\sigma = \langle a\rangle$

  $\Omega = \{\langle a\rangle ::= 11\langle a\rangle 0,\ \langle a\rangle ::= 110\}$

$110 \in L(G)$ $111100 \in L(G)$ But $011 \notin L(G)$



These trees are called *parse trees* or *syntax trees* or *derivation trees.*

Why CFG?

- A programming language may have arbitrary number of nested statements, such as:

  `if-then-else`, `let-=-in-`, and so on.

- $L_1 = \{(\texttt{if-then})^n (\texttt{let-in})^m \mid n, m > 0\}$

- These can be easily supported by CFG.

Backus-Naur Form

- BNF is a common notation to define context-free grammars for programming languages
- ⟨**digit**⟩ is defined to represent one of the ten tokens 0, 1, …, 9

  ⟨digit⟩ ::= 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9

- (Positive) Integers

  ⟨integer⟩ ::= ⟨digit⟩ | ⟨digit⟩ ⟨integer⟩

  ⟨digit⟩ ::= 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9

- **⟨integer⟩** is defined as the sequence of a ⟨**digit**⟩ followed by zero or more ⟨**digit**⟩’s

Extended Backus-Naur Form

- EBNF is a more compact notation to define the syntax of programming languages.
- EBNF has the same power as CFG.
- It provided some *shortcuts* to denote repetition.

- With EBNF, (positive) integers may be defined as:

  ⟨integer⟩ ::= ⟨digit⟩ { ⟨digit⟩ }

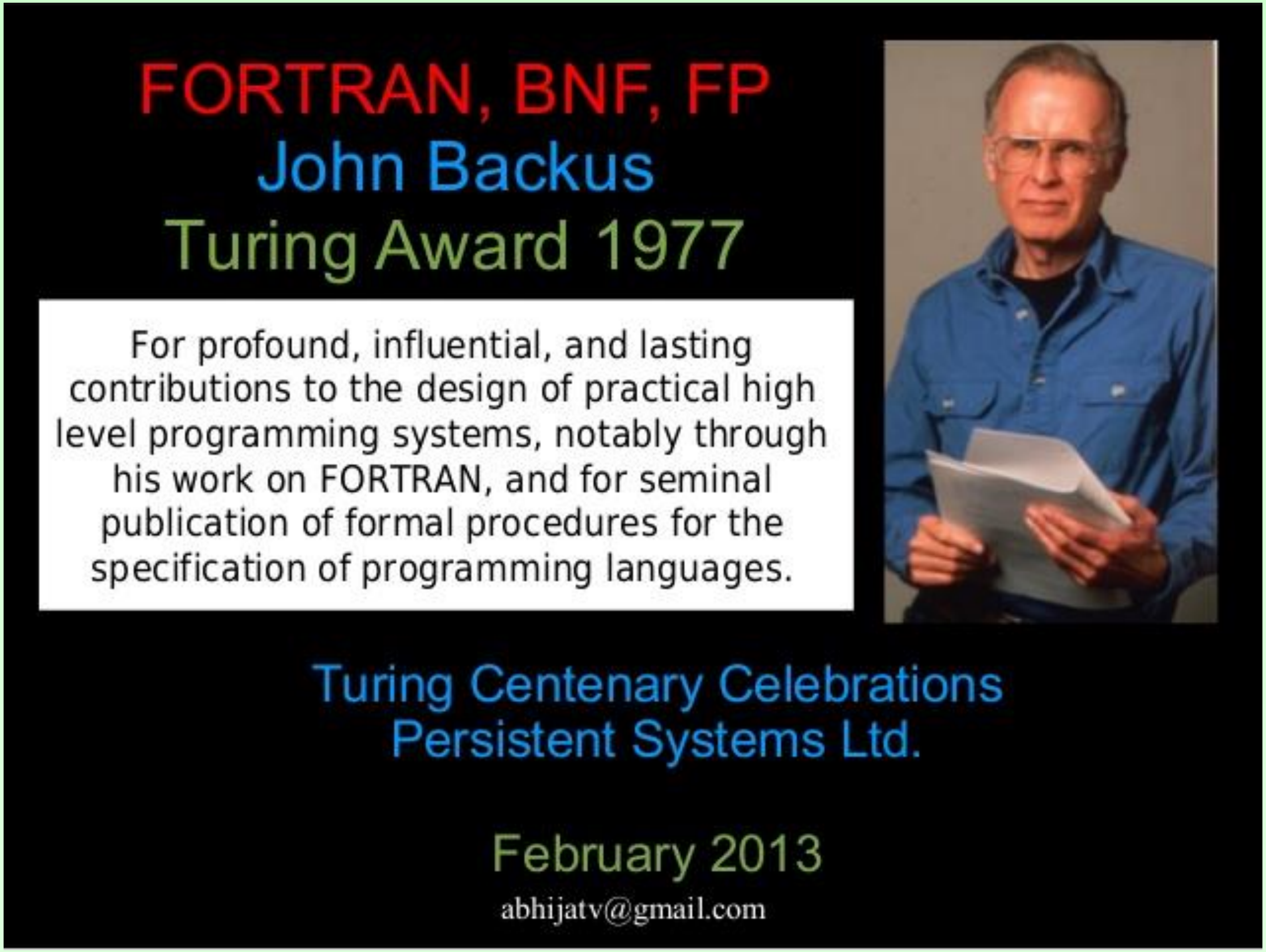
- Recall that BNF was:

  ⟨integer⟩ ::= ⟨digit⟩ | ⟨digit⟩ ⟨integer⟩

Notations

$\langle x \rangle$	nonterminal *x*
$\langle x \rangle$ ::= *Body*	$\langle x \rangle$ is defined by *Body*
$\langle x \rangle \mid \langle y \rangle$	either $\langle x \rangle$ or $\langle y \rangle$ (choice)
$\langle x \rangle \langle y \rangle$	the sequence $\langle x \rangle$ followed by $\langle y \rangle$
$\{ \langle x \rangle \}$	sequence of zero or more occurrences of $\langle x \rangle$
$\{ \langle x \rangle \}^{+}$	sequence of *one or more* occurrences of $\langle x \rangle$
$[\langle x \rangle]$	*zero or one* occurrence of $\langle x \rangle$

Our Pioneers



FORTRAN, BNF, FP

John Backus

Turing Award 1977

For profound, influential, and lasting contributions to the design of practical high level programming systems, notably through his work on FORTRAN, and for seminal publication of formal procedures for the specification of programming languages.

Turing Centenary Celebrations
Persistent Systems Ltd.

February 2013

abhijatv@gmail.com

EBNF for Haskell Fragment

⟨expression⟩ ::=

⟨lowercase-identifier⟩

| ⟨constants⟩

| let ⟨pattern⟩ = ⟨expression⟩ in <expression>

| if ⟨expression⟩ then ⟨expression⟩ [else ⟨expression⟩]

| \ {⟨pattern⟩}$^+$ –> ⟨expression⟩

| ⟨expression⟩ {⟨expression⟩}$^+$

| (⟨expression⟩) | …

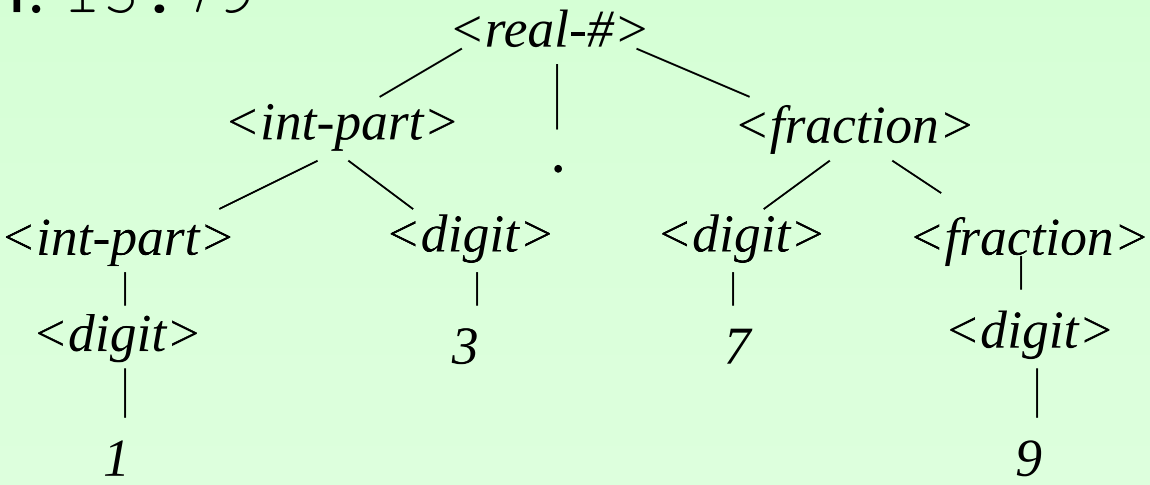
Regular Grammar for Lexical Analysis

Examples

- Description of (positive) real numbers:

<real-#> ::= <int-part> . <fraction>
<int-part> ::= <digit> | < int-part > <digit>
<fraction> ::= <digit> | <digit> <fraction>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Token: `13.79`



Regular Grammars

- A regular grammar (RF) is a subset of CFG:
  - A set of terminal symbols T (tokens or constants)
  - A set of non-terminal symbols N
  - One (non-terminal) start symbol σ
  - A set of grammar (rewriting) rules Ω of the form

⟨nonterminal⟩ ::= ⟨terminal> [<nonterminal>]

- Regular grammar can be expressed in terms of iterative construct of EBNF.

<int-part> ::= *<digit>*

<int-part> ::= *<digit> <int-part>*

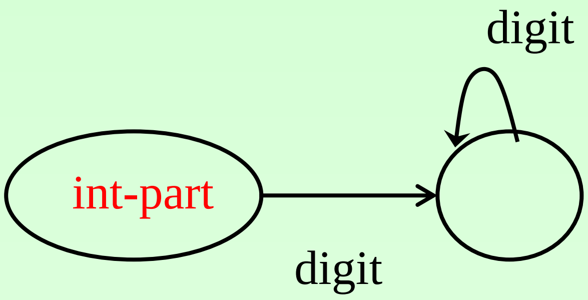
⇩

<int-part> ::= { *<digit>* }+

Regular Grammars

- Can be recognized by a finite state machine (and tail-recursive methods)

<int-part>   ::=   { *<digit>* }+



Implementation

- Two ways of recognizing regular grammars
  - Use tail-recursive methods
  - Use combinators for regular grammars

(Details are omitted in Lecture)

Parsers

A Generic Parser

- We can build generic parser.
- Each parser is expected to have the following type:

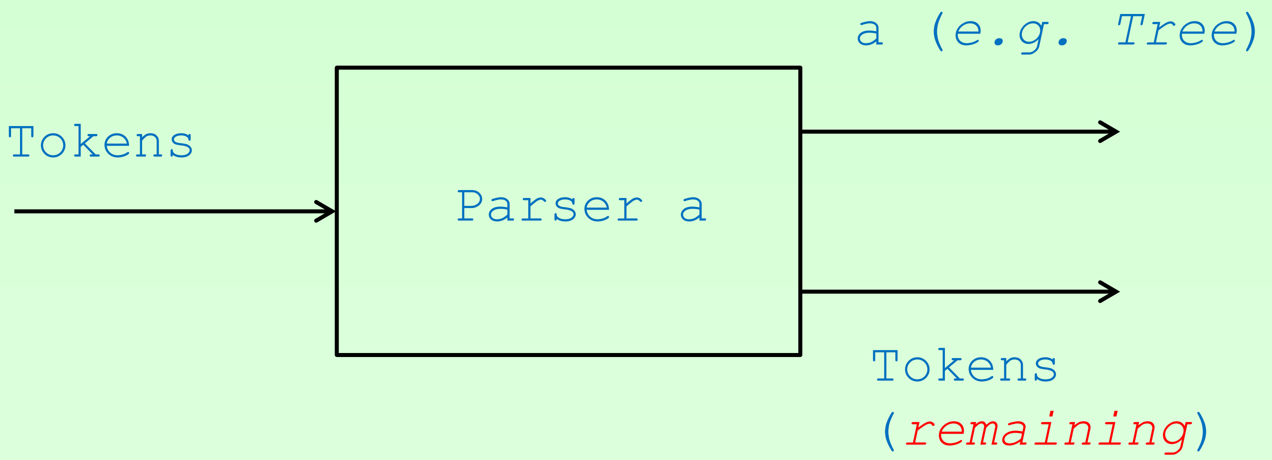
```
data Token = …
type Tokens = [Token]

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

- It consumes some token so as to produce a parse tree object of type a.

A Generic Parser

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



Keyword Parser

- Simple parsers would recognize keywords, identifiers and constants.

```
symbol :: String -> Parser ()
symbol s (Key b:toks) =
    if s == b then
        ((), toks’)
    else
        error ("Symbol "+s+" expected.")

symbol s _ = error ("Symbol “ + s + " expected.")
```

- Exception is thrown when expected keyword is not present.
- Why is unit returned?

Parser for Identifier

- A parser for identifier would return the string of the identifier.

```
ident :: Parser String
ident (ID a::toks) = (a, toks’)
ident _            = error ("Identifier expected")
```

- Exception is thrown when identifier is not the next token.
- Note that the identifier token is consumed.

Parser for Constants

- A parser for constant would return the constant itself. In the case of integer, we expect the parser to have int type as its result.

```
numb :: Parser Int
numb (Num a:toks) = (a, toks)
numb _            = error ("Integer expected")
```

Exception is thrown when integer constant is not the next token.

Parser Combinators

- We need some ways to build bigger parsers from smaller parsers.

- We would like to have:
  - Sequential composition.
  - Alternative parsing.
  - Repetitive parsing
  - Optional parsing
  - A parser with a mapping transformer

Recall Notations

$\langle x \rangle$	nonterminal x
$\langle x \rangle ::= \mathit{Body}$	$\langle x \rangle$ is defined by *Body*
$\langle x \rangle \mid \langle y \rangle$	either $\langle x \rangle$ or $\langle y \rangle$ (choice)
$\langle x \rangle \langle y \rangle$	the sequence $\langle x \rangle$ followed by $\langle y \rangle$
$\{ \langle x \rangle \}$	sequence of zero or more occurrences of $\langle x \rangle$
$\{ \langle x \rangle \}^{+}$	sequence of *one or more* occurrences of $\langle x \rangle$
$[\langle x \rangle]$	*zero or one* occurrence of $\langle x \rangle$

Monadic World

```
class Monad m where
    (>>=)     :: m a -> (a -> m b) -> m b
    return    :: a -> m a
class Functor f where
    fmap      :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
    pure      :: a -> f a
    (<*>)     :: f (a -> b) -> f a -> f b
    (<*>)      = liftA2 (\x->x)
    liftA2    :: (a -> b -> c) -> f a -> f b -> f c
    liftA2 f x = (<*>) (fmap f x)
```

Monadic World

```
class Applicative f => Alternative f where
    empty         :: f a
    (<|>)         :: f a -> f a -> f a

-- Example of instance
instance Alternative Maybe where
    empty         = Nothing
    Nothing <|> x = x
    Just x  <|> _ = Just x
    -- Maybe can only hold up to one result,
    -- so we discard the second one.
instance Alternative [] where
    empty         = []
    (<|>)         = (++)
```

Do-Notation Sugar

Syntactic Sugar for bind operation:

conceptually:

```
m_expr ::= return expr
         | m_expr >>= (\v -> m_expr)
```

```
m_expr >>= (\var -> return expr)
```

is same as

```
do
   var <- m_expr
   return expr
```

Sequential Parser Combinator

- Sequential parser applies one parser and then another.
- It can be implemented using:

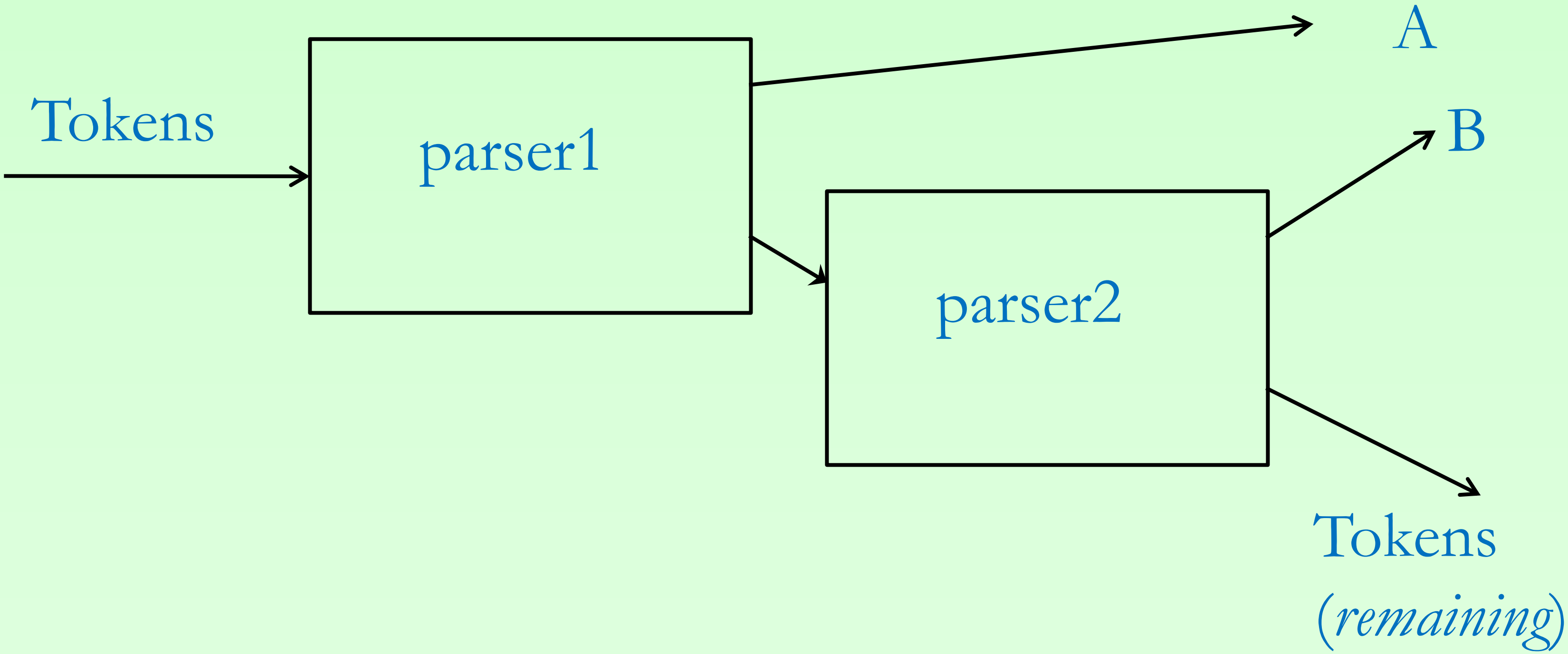
```
(~~) :: Monad SParsec => SParsec a -> SParsec b -> SParsec (a, b)
(~~) ph1 ph2 = do
    tup1 <- ph1
    tup2 <- ph2
    return (tup1, tup2)

hexPre = char ‘0’ ~~ char ‘x’
parse hexPre “(source-filename)” -- “0xAB0”
```

- Note that a tuple (a,b) is being returned.

A Generic Parser

parser1 ~~ parser2



Alternative Parser Combinator

- Alternative combinator **<|>** allows a choice between two parsers.

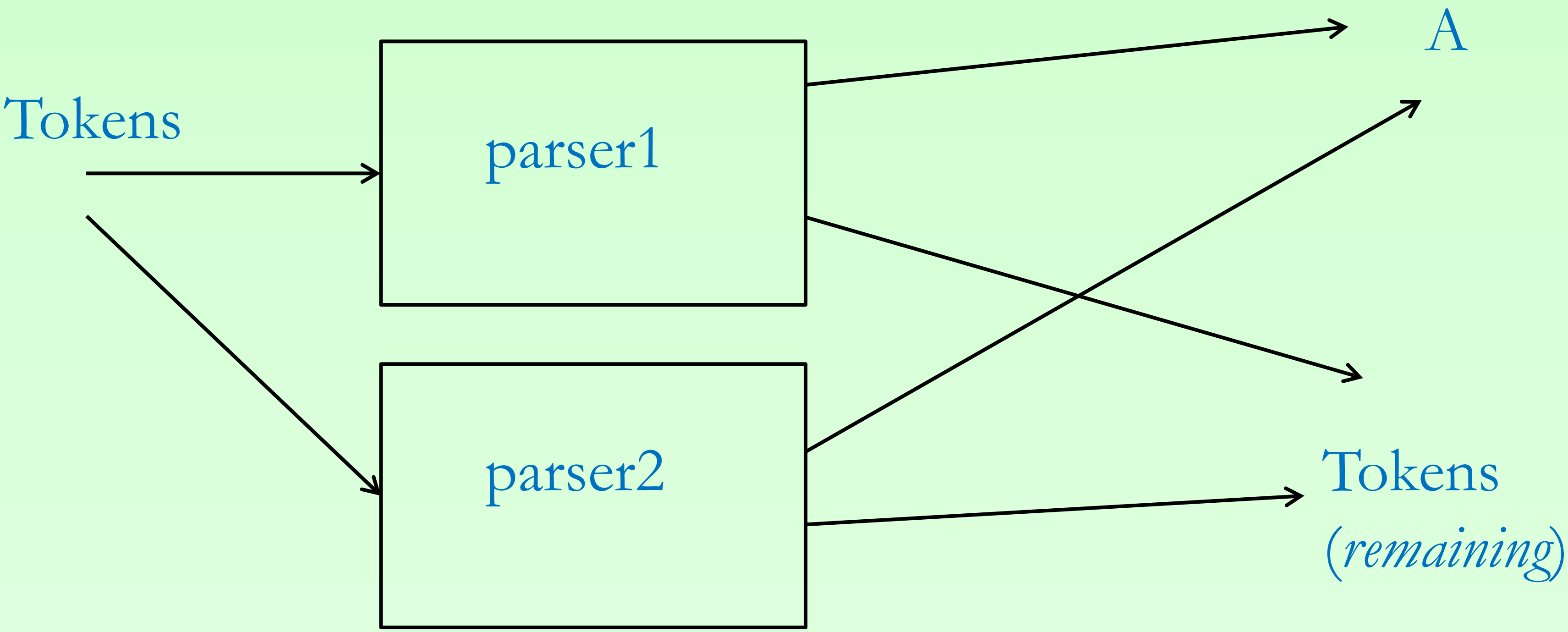
```
 (<|>) ::  SParsec a -> SParsec a -> SParsec a


parserA <|> parserB
```

- Recall that it tries the first alternative, and would only consider the second parser if the first parser fails and no input is consumed.

A Generic Parser

parser1 <|> parser2



Parser Mapping

- This is used to change the result of a parser by a map operation.
- It can be used to discard or transform outcomes:

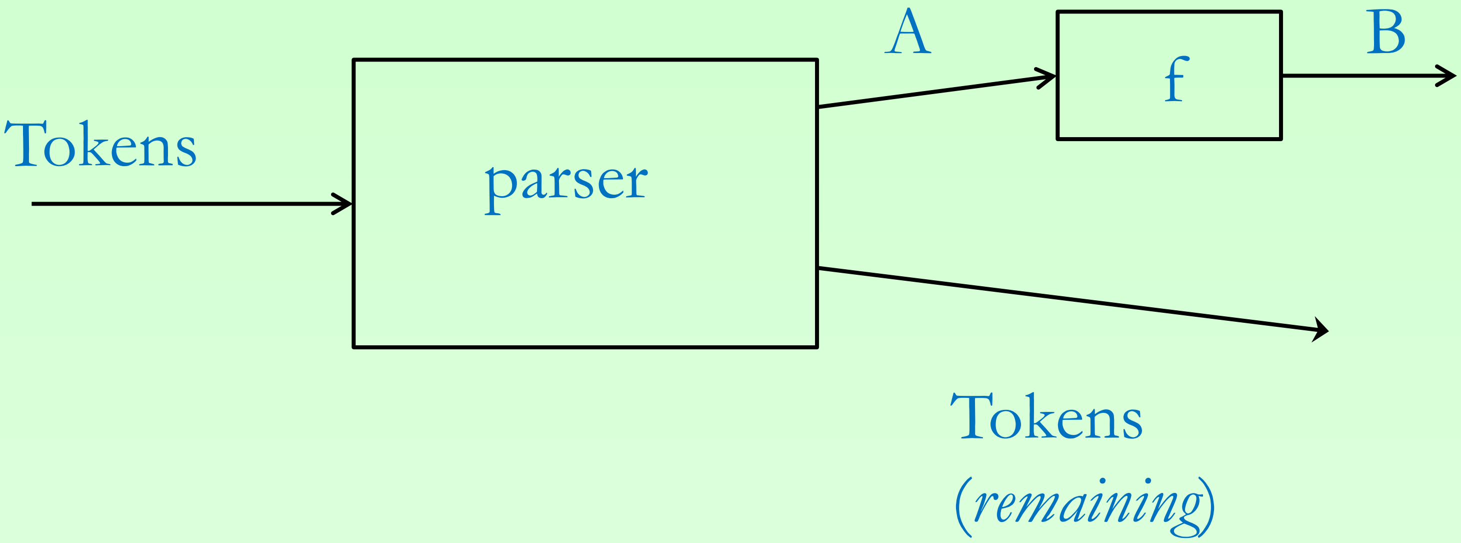
```
parsecMap :: (a -> b) -> SParsec a -> SParsec b
parsecMap f m = do
                   x <- m
                   return $ f x

parsecMap (read :: Int) (many1 digits)
```

- Function f changes the result of the parser.

A Generic Parser

parsecMap f parser



Repetitive Parsing

- We can repeat a parser zero or more times.
- It can be implemented using:

```
many  :: SParsec a -> SParsec [a]
many1 :: SParsec a -> SParsec [a]
many1 p = do x <- p
             xs <- many p
             return (x:xs)

many1 digits
```

- Note it repeats until the given parser fails.

Some Other Combinators

- We may have a parser optionally applied.

```
option    :: a -> SParsec a -> SParsec a
option x p = p <|> return x

choice :: [SParsec a] -> SParsec a
choice ps = foldr (<|>) mzero ps

chainl :: SParsec a -> SParsec (a -> a -> a) -> a -> SParsec a
chainl1 :: SParsec a -> SParsec (a -> a -> a) -> SParsec a
-------------------------------------------------------
option 0 (many digits)
choice [char ‘x’, char ‘y’, char ‘z’]
```

Text.Parsec

- Combinators:

```
p1 ~~ p2                     -- sequencing: must match p1 followed by p2
p1 <|> p2  -- alternation: must match either p1 or p2, with preference given to p1
p1 <?> st                    -- may match p1 or show st as error message
try p1                       -- does not consume any input if fails
choice [p1,p2,…]             -- applies alternation sequentially
many p1                      -- repeating zero or more times
many1 p1                     -- repeating zero or more times
skipMany p1                  -- like many but skips its result
between open p1 close        -- parses open, then p1, and at last close
parseMap f p1                -- map function f to the parser’s result
p1 *> p2                     -- like ~~ but ignore left member
p1 <* p2                     -- like ~~ but ignore right member
```

An Example :
Arithmetic Expression Parser

Arithmetic Grammar Rule

- Right-recursive to support right associativity of operators:

```
<fac>  ::= <constant> | “(“ <expr> “)”
<term> ::= <fac>      | <fac> (“*” | ”/”) <term>
<expr> ::= <term>     | <term> (“+” | ”-”) <expr>
```

- This runs but recursive case need to be considered where possible to cover larger expression.

Arithmetic Grammar Rule

- Left-recursive to support left associativity of operators:

```
<fac>  ::= <id>   | <constant> | “(“ <expr> “)”
<term> ::= <fac>  | <term> (“*”|”/”) <fac>
<expr> ::= <term> | <expr> (“+”|”-”) <term>
```

- Such left-recursion works well <u>only</u> if we had a non-deterministic parser since such parsers always terminate whenever there is a base case.

Arithmetic Grammar Rule

- Solution : Use repetition construct of EBNF to handle left associativity.

```
<fac>  ::= <constant> | “(“ <expr> “)”
<term> ::= <fac> { (“*”|”/”) <fac> }
<expr> ::= <term> { (“+”|”-”) <term> }
```

Example Parser

```
module SParsec where

import Text.Parsec

data Expr = Const Int       | Plus Expr Expr
          | Minus Expr Expr | Mult Expr Expr
          | Div Expr Expr   deriving (Show)

eAdd  x y  = Plus x y
eSub  x y  = Minus x y
eMult x y  = Mult x y
eDiv  x y  = Div x y
sToC  s    = Const (read s)

type SParsec = Parsec String ()
```

Example Parser

```
expr :: TParsec Expr
expr = chainl1 term addop                 -- x+y-z+…

term :: TParsec Expr
term = chainl1 factor mulop               -- x*y/z*…

factor :: TParsec Expr
factor = (parens expr) <|> constants -- 12 | (…)

parens :: TParsec Expr -> TParsec Expr
parens ex = do char '(‘
               x <- ex
               char ')’
               return x
```

Example Parser

```
constants :: SParsec Expr
constants = parsecMap sToC $ many1 digit

digit :: SParsec Char
digit =
    char '0' <|> char '1' <|> char ‘2’ <|>
    char '3' <|> char '4' <|> char '5’ <|>
    char '6' <|> char '7' <|> char '8' <|> char '9'

mulop :: SParsec (Expr -> Expr -> Expr)
mulop = do { char '*'; return eMult }
    <|> do { char '/'; return eDiv}

addop :: SParsec (Expr -> Expr -> Expr)
addop = do { char '+'; return eAdd }
    <|> do { char '-'; return eSub }
```

Example Parser

```
calcE ::  String -> Either ParseError Expr
calcE x = parse expr "" x

eval :: Either ParseError Expr -> Either ParseError Int
eval x =
      let eval’ :: Expr -> Int
          eval’ Const  x = x
          eval’ Add  x y = eval x + eval y
          eval’ Sub  x y = eval x - eval y
          eval’ Mult x y = eval x * eval y
          eval’ Div  x y = div (eval x)  (eval y)
      in
          fmap eval’ x
```

Tail-Recursive Implementation

Implementation Considerations

- We must identify reserved keywords and special symbols.
- We can support it using two list of reserved ids:

```
-- permitted symbols
symbols = [“=”,“+”,“-”,“*”,“/”,“~”,“(”,“)”]

-- reserved keywords in calculator
alphas = ["let", "in"]
```

Implementation for Lexical Analyser

- We must define the token type and a pretty printer:

```
data Token =
  -- a lexical word
     Key String
  |  Id String
  |  Num Int
  -- exception for lexical analyser
data Exception = LexErr String

instance Show Token where
  show Key s = "Key "  ++ s
  show Id  s = "Id “   ++ s
  show Num n = "NumI “ ++ (show n)
```

Implementation for Lexical Analyser

- Let us implement a scanner for alphanumeric identifier after a letter has been captured in parameter id.

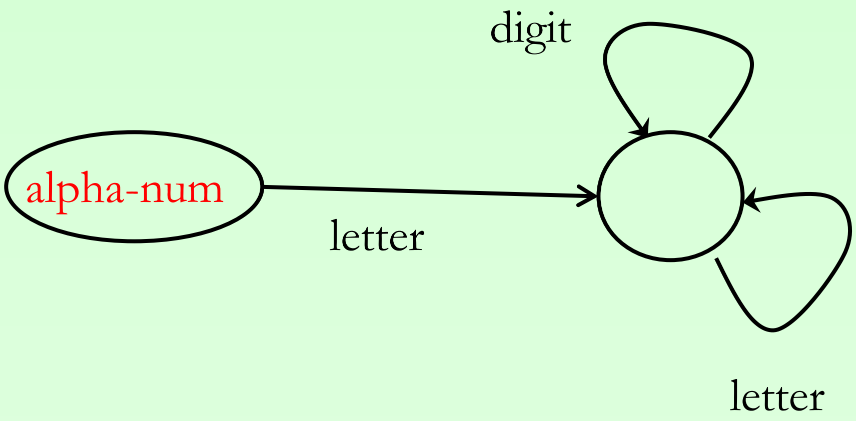
```
alphanum :: [Char] -> [Char] -> ([Char], [Char])
alphanum []      id = (reverse id, [])
alphanum (c:cs) id = if is_letter_or_digit c then
                            alphanum cs (c:id)
                       else
                            (reverse id, (c:cs))
```

- This implementation should be *tail-recursive* as tokens are expressible using regular grammar that can be recognized by finite automata.

Regular Grammars

Can be recognized by a finite state machine (and tail-recursive methods)

<alphanum> **::=** ***<letter>*** **{** ***<letter>*****|*****<digit>*** **}**



Identifying Alphanumeric Keyword

- Whenever we have identified an alphanumeric variable, we can convert it to a token by checking if it is a keyword or an identifier.

```
is_mem x = foldl false (\a z -> a || x=z)

tokenof :: [Char] -> Token
tokenof a = if is_mem a alphas then
                   Key a
            else
                   Id a
```

Implementation for Lexical Analyser

- Scanner for symbolic keyword would search for shortest possible match, as these symbolic keywords need *not* be separated by white spaces.

```
symbolic :: [Char] -> String -> (String, [Char])
symbolic []       sy = (sy, [])
symbolic (c:cs) sy =
          if is_mem sy symbols then
               (sy,c:cs)
          else if not (is_mem c specials) then
                error ("Unrecognized lex symbol " ++ sy)
          else
                symbolic cs (sy ++ [c])
```

Regular Expression in Haskell

Regex

- Text.Regex.Posix gives basic functionality of regular expression in Haskell.
- Declare a date format with regular expression with three groups.

```
rgDate = "([1-2][0-9]{3})-([0-1]?[0-9])-([0-3]?[0-9])”
```

- (=~) is a polymorphic operator to match regular expression
- We can check if a string matches a regular expression

```
"2017-9-22" =~ rgDate :: Bool
-- returns True
```

Regex

- We can get the matched sub-string

```
"2017-9-22 ..." =~ rgDate :: String
-- returns "2017-9-22"
```

- We can retrieve all matches and the sub-expressions

```
"1.2017-9-22 2.1999-12-31" =~ rgDate :: [[String]]
-- [["2017-9-22","2017","9","22"],["1999-12-31","1999","12","31"]]
```

- We can also retrieve their offsets and lengths

```
"1.2017-9-22 2.1999-12-31" =~ rgDate :: [MatchArray]


-- [array (0,3) [(0,(2,9)),(1,(2,4)),(2,(7,1)),(3,(9,2))],
    array (0,3) [(0,(14,10)),(1,(14,4)),(2,(19,2)),(3,(22,2))]]
```

Regex

- Find all matches in a string

```
getAllTextMatches ("1.2017-9-22 2.1999-12-31" =~ rgDate)
   :: [String]
-- ["2017-9-22","1999-12-31"]
```

- Find string before matches, when matches, after matches, and all matched sub-expressions

```
"1.2017-9-22 2.1999-12-31" =~ rgDate
   :: (String, String, String,[String])
-- ("1.","2017-9-22"," 2.1999-12-31",["2017","9","22"])
```