CMPT383 Comparative Programming Languages

Lecture 12: Monadic Parsing II

Yuepeng Wang

Spring 2023

Overview

- Derived Primitives
- Handling whitespaces
- Introduction to grammars

Review

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

- Parser is an instance of Functor, Applicative, Monad, Alternative
- Parsing primitives
  - item consumes a single character if the input is not empty
  - return v always succeeds with the result v but does not consume input
  - empty always fails
  - sat p consumes a single character if it satisfies predicate p, e.g. digit, lower, alphanum, char c, …
- p <|> q means “choose p, but if p fails, then choose q”

Derived Primitive: string

- Define a parsing primitive called string. It takes as input xs and succeeds with result xs if the input starts with xs, otherwise fails

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

- Execution example

```
ghci> parse (string "abc") "abcdefg"
[("abc","defg")]
ghci> parse (string "abc") "abd"
[]
```

Derived Primitives: many and some

- Two primitives many and some are provided by Alternative

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

  many x = some x <|> pure []
  some x = pure (:) <*> x <*> many x
```

- For Parser, many p applies a parser p as many times as possible until it fails, with the results from each successful application in a list

- some p is similar to many p but requires at least one successful application

Derived Primitives: many and some

- Execution Example

```
ghci> parse (many digit) "123abc"
[("123","abc")]
ghci> parse (many digit) "abc"
[("","abc")]
ghci> parse (some digit) "abc"
[]
```

- Example: identifiers starting with a lower case letter followed by alphanumeric characters (letters/digits)

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

Derived Primitives: many and some

- Example: natural numbers

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

- Example: whitespaces (space, \t, \r, …)

```
space :: Parser ()
space = do many (sat isSpace)
           return ()
```

- Execution example

```
ghci> parse ident "abc def"
[("abc"," def")]
ghci> parse nat "123 def"
[(123," def")]
ghci> parse space "   abc"
[((),"abc")]
```

Handling Whitespaces

- Many parsers allow spacing around the basic tokens

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

- We can define different parsers ignoring spaces

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

natural :: Parser Int
natural = token nat

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

Handling Whitespaces

- Example: a list of natural numbers

```
nats :: Parser [Int]
nats = do symbol "["
          n <- natural
          ns <- many (do symbol ","
                         natural)
          symbol "]"
          return (n : ns)
```

- Execution example

```
ghci> parse nats "[1,  2,3]"
[([1,2,3],"")]
ghci> parse nats "[1,2,]"
[]
```

Grammars

- The syntactic structure of a language can be formalized using grammars. A grammar is a set of rules that describe how strings of the language can be constructed
- Example: arithmetic expressions over natural numbers

```
expr ::= expr '+' expr | expr '*' expr | '(' expr ')' | nat
nat  ::= '0' | '1' | '2' | ...
```

- expr, nat, +, *, (,), 0, 1, 2, … are symbols
- X ::= Y Z… is called a production; X ::= Y | Z stands for X ::= Y or X ::= Z
- A terminal symbol cannot be changed using the production rules
- Otherwise, it is a non-terminal symbol, e.g., expr can be changed to nat
- A grammar also has a designated start symbol, e.g., expr

Grammars

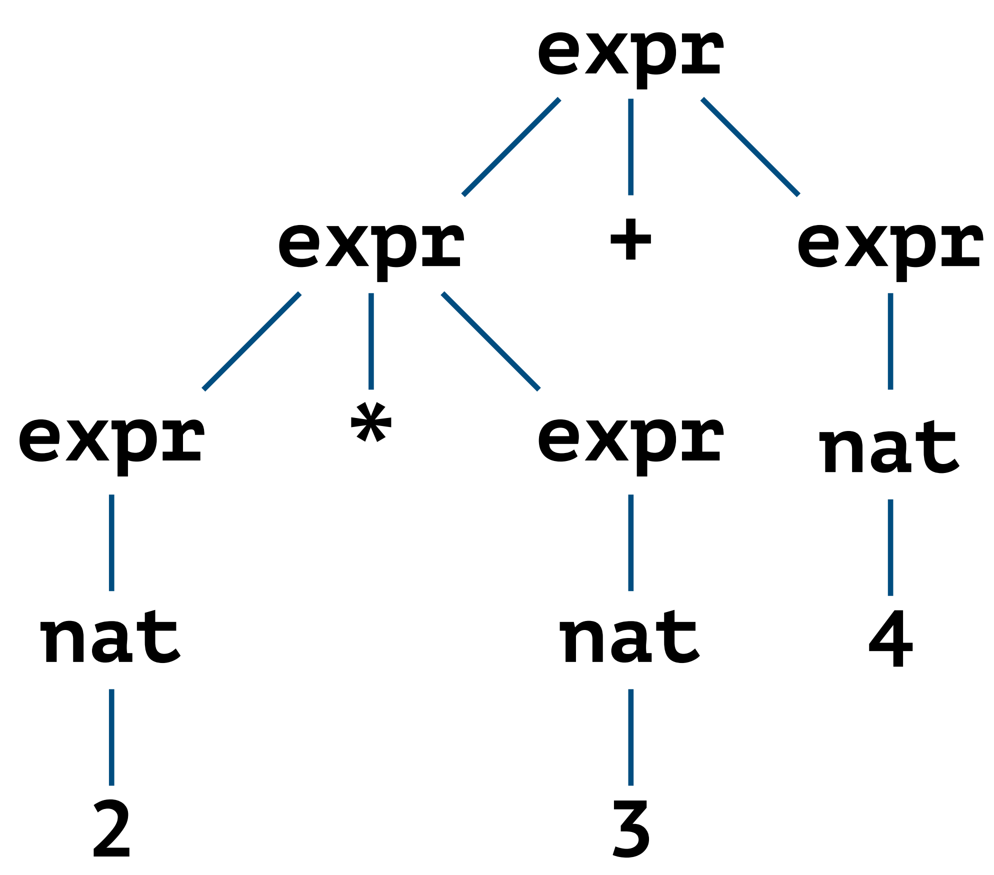
```
expr ::= expr '+' expr | expr '*' expr | '(' expr ')' | nat
nat  ::= '0' | '1' | '2' | ...
```

- A derivation begins with the start symbol, repeatedly replaces LHS of a production with its RHS, until there are only terminals left
- A string is in the language of a grammar if it can be derived from the start symbol
- Example: 2*3+4 is in the language of above grammar
- Example: 2+*3 is not
- The derivation process is basically the parsing process

Parse Trees

```
expr ::= expr '+' expr | expr '*' expr | '(' expr ')' | nat
nat  ::= '0' | '1' | '2' | ...
```

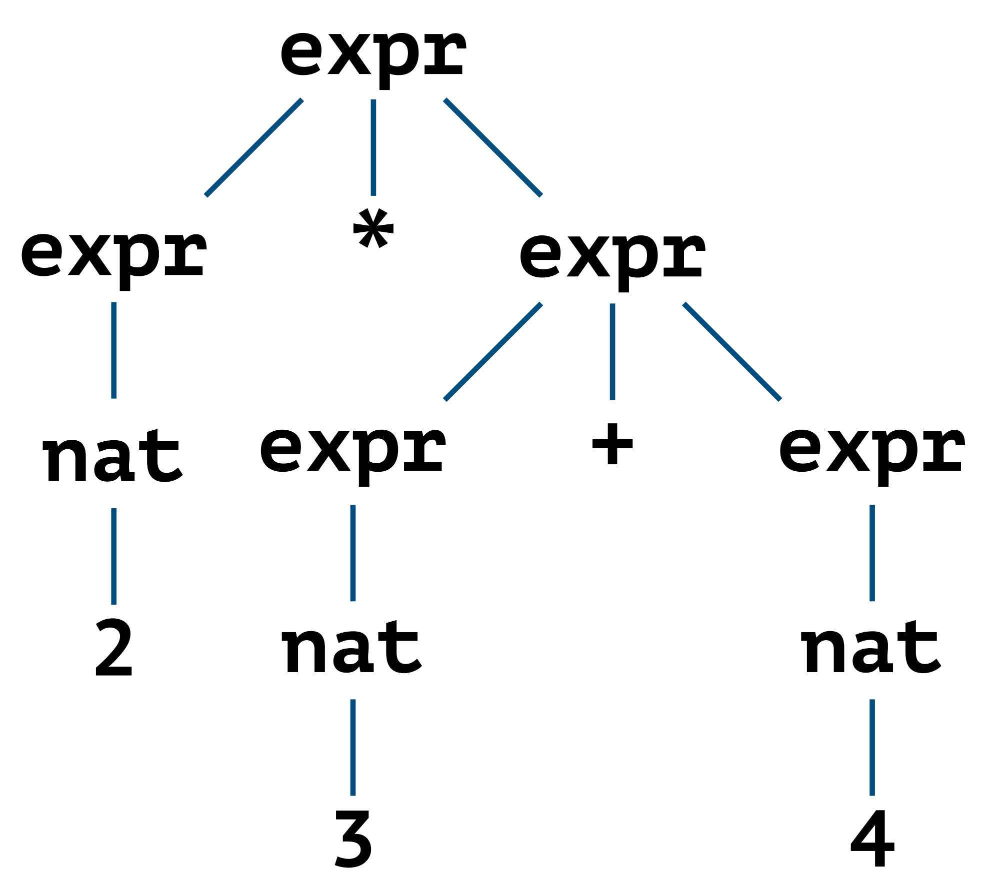
- The parsing process can be represented as parse trees
  - The root node corresponds to the start symbol
  - Branches correspond to a production: LHS is parent, RHS are children
  - A leaf node corresponds to a terminal
- Example: 2*3+4




```
   expr
→ expr + expr
→ expr * expr + expr
→ nat * expr + expr
→ 2 * expr + expr
→ 2 * nat + expr
→ 2 * 3 + expr
→ 2 * 3 + nat
→ 2 * 3 + 4
```

Parse Trees

- 2*3+4 has another way to parse (corresponding to 2*(3+4))



- There are multiple parse trees because | stands for non-deterministic choice

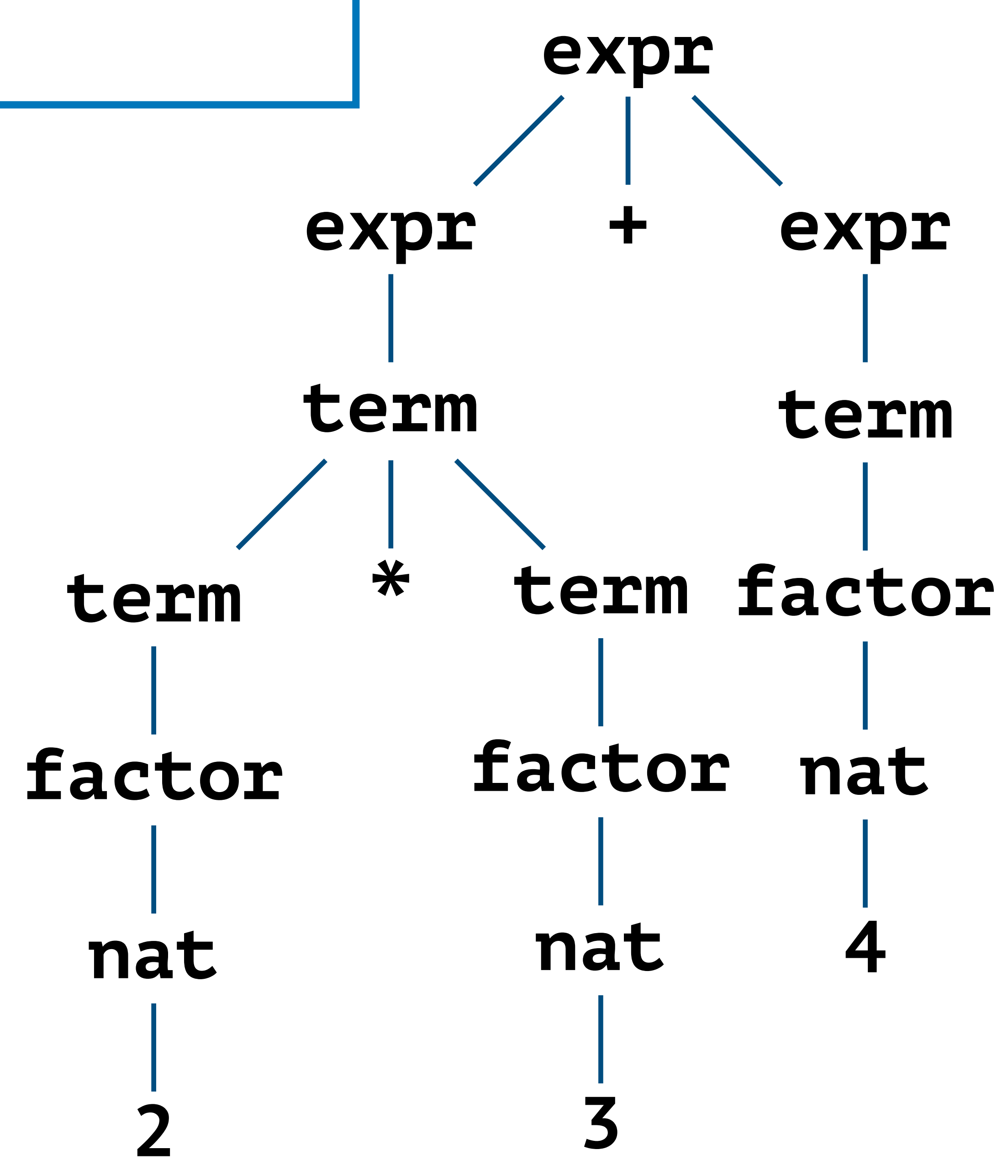
- The precedence of * and + in this case is not conventional

Precedence

- To enforce precedence between operators, we can modify the grammar to have separate production rules for each level of precedence

```
expr ::= expr '+' expr | term
term ::= term '*' term | factor
factor ::= '(' expr ')' | nat
nat  ::= '0' | '1' | '2' | ...
```

- Top level: +, lowest precedence
- Second level: *, medium precedence
- Third level: parenthesis,
  natural numbers, highest precedence
- 2*3+4 now only has one parse tree



Associativity

- The grammar still allows two parse trees for expression 2+3+4. One corresponds to (2+3) + 4, and the other corresponds to 2 + (3+4)
- Suppose we want both operators to be right-associative, we can enforce it by making the related production recursive in the right argument only

```
expr ::= term '+' expr | term
term ::= factor '*' term | factor
factor ::= '(' expr ')' | nat
nat   ::= '0' | '1' | '2' | ...
```

- Now, 2+3+4 only has one parse tree corresponding to 2 + (3+4)
- This grammar is unambiguous, because every well-formed expression has exactly one parse tree

Parsers and Grammars

- The parser for language defined by a grammar is closely related to the grammar itself

```
expr ::= term '+' expr | term
term ::= factor '*' term | factor
factor ::= '(' expr ')' | nat
nat  ::= '0' | '1' | '2' | ...
```

- Constructs of grammars correspond to parsing primitives
    - Sequencing in the grammar corresponds to the do notations
    - Choice | corresponds to the <|> operator

Example

- Write a parser that evaluates expressions defined by the grammar

```
expr ::= term '+' expr | term
term ::= factor '*' term | factor
factor ::= '(' expr ')' | nat
nat  ::= '0' | '1' | '2' | ...
```

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

** Note that the indentation of <|> should be larger than do but not larger than the body of do

```haskell
term :: Parser Int
term = do f <- factor
          symbol "*"
          t <- term
          return (f * t)
        <|> factor

factor :: Parser Int
factor = do symbol "("
            e <- expr
            symbol ")"
            return e
          <|> natural
```

Example

- Execution example

```
ghci> parse expr "2*3+4"
[(10,"")]
ghci> parse expr "2+3*4"
[(14,"")]
ghci> parse expr "2*(3+1)"
[(8,"")]
```