CSE216
Programming Abstraction

Instructor: Zhoulai Fu

State University of New York, Korea

Some slides taken from https://pages.cs.wisc.edu/~aanjneya/courses/cs154/lectures/lec6.pdf Thanks!

Today

- This lecture 1: Exercises for regular expression
- This lecture 2: Context-free grammar
- Recitation: Exercises for Context-free grammar
- No quiz this week
- Homework to be announced later today. Due next Thursday.

Exercises:
Regular Expression

Exercise 1

Write regular expressions for:

- Non-negative integer constants

Demo: https://regex101.com/

Exercise 2

Write regular expressions for:

- Integer constants

Exercise 3

Write regular expressions for:

- Floating-point constants:
  - `3.14`
  - `3E8`
  - `+6.02E23`

Exercise 4

Write regular expressions for:

- Java variable names:
  - `xy`
  - `x12`
  - `_x`
  - `$x12`

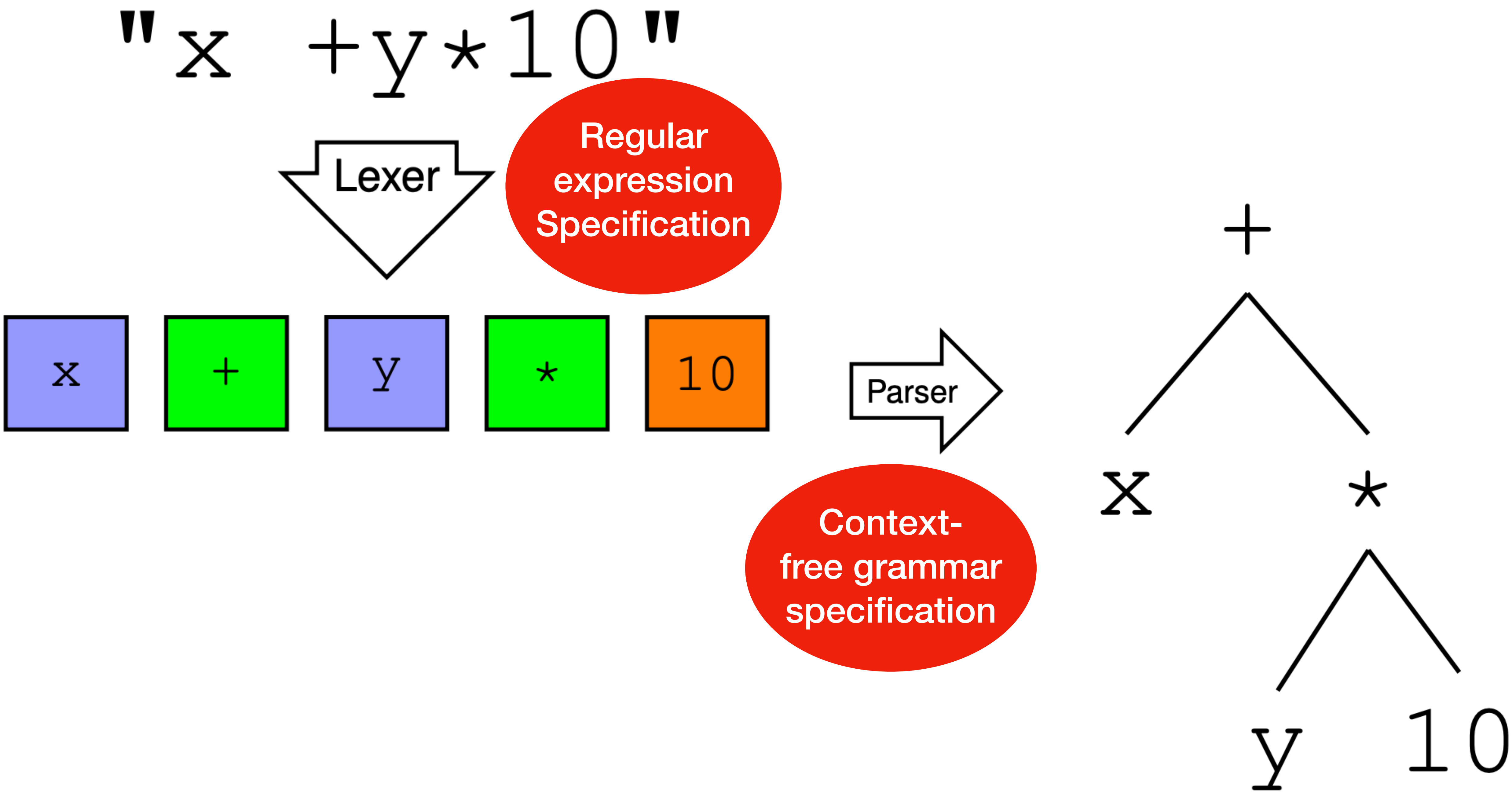
Exercise 5

Give a RE for: $L = \{0^i 1^j \mid i \text{ is even and } j \text{ is odd }\}$

0^i above refers to repeating “0” i times. E.g. 0^4 means “0000”

Context-free grammar

Parsing


"x +y*10"
Lexer
Regular expression Specification
x
+
y
*
10
Parser
Context-free grammar specification
+
x
*
y
10

The need for a grammar

"Afternoon good, I'd room a like."

Mr. Men and Little Miss Series

Context-Free Grammar

- A notation for describing languages.
- More powerful than regular expressions.
- It still cannot define all possible languages.
- Useful for recursive structures, e.g., most today's programming languages.

Basic idea

- A language can be decomposed to smaller parts
- Each part can be defined recursively
- Use **production rules** to generate the language

Example 1

```
Program ::= Stmt Program
          | ""
Stmt    ::= Var = AExpr
          | if ( BExpr ) Stmt else Stmt
          | while ( BExpr ) Stmt
AExpr   ::= AExpr + AExpr
          | AExpr − AExpr
          | AExpr ∗ AExpr
          | AExpr / AExpr
          | Var
          | Number
BExpr   ::= AExpr == AExpr
          | AExpr < AExpr
          | not (BExpr)
          | BExpr and BExpr
          ...
Var    ::= x | y | z ...
Number ::= 0 | 1 | 2 ...
```

Demo: Parse tree for x = 2

Example 2

- Production rule:
    - S → 01
    - S → 0S1
- Basis: 01 is in the language.
- Induction: If w is in the language, then so is 0w1.
- The generated language is {0^n 1^n, n>=1}

Overview

- Use **terminal** symbols a, b, c, d… for the alphabet of a language.
- Use **nonterminal** symbols A, B, C, D, recursively
- **Starting symbol** is a special nonterminal
- Use **production rules** to generate the language

Production

- A production has the form
  variable $\rightarrow$ string of variables and terminals
- Convention
  - A, B, C,... are variables.
  - a, b, c,... are terminals.
  - ..., X, Y, Z are either terminals or variables.
  - ..., w, x, y, z are strings of terminals only.
  - α, β, γ,... are strings of terminals and/or variables.

Put Everything Together

- Here is a formal CFG for $\{0^n1^n \mid n \geq 1\}$.
- Terminals $= \{0,1\}$.
- Variables $= \{S\}$.
- Start symbol $= S$.
- Productions $=$
  $S \rightarrow 01$
  $S \rightarrow 0S1$

Notation

- Symbol $::=$ is often used for $\rightarrow$.
- Symbol $|$ is used for or.
  - A shorthand for a list of productions with the same left side.

Example: $S \rightarrow 0S1 \mid 01$ is a shorthand for
$S \rightarrow 0S1$ and $S \rightarrow 01$.

Exercise 1: Construct a parse tree

```
Program ::= Stmt Program
          | ""
Stmt    ::= Var = AExpr
          | if ( BExpr ) Stmt else Stmt
          | while ( BExpr ) Stmt
AExpr   ::= AExpr + AExpr
          | AExpr - AExpr
          | AExpr * AExpr
          | AExpr / AExpr
          | Var
          | Number
BExpr   ::= AExpr == AExpr
          | AExpr < AExpr
          | not (BExpr)
          | BExpr and BExpr
          ...
Var    ::= x | y | z ...
Number ::= 0 | 1 | 2 ...
```

while (x<10) { x = x + 1 }

Exercise 2: Construct a parse tree

```
AExpr   ::= AExpr + AExpr
          | AExpr - AExpr
          | AExpr * AExpr
          | AExpr / AExpr
          | Var
          | Number
BExpr   ::= AExpr == AExpr
          | AExpr < AExpr
          | not (BExpr)
          | BExpr and BExpr
          ...
Var    ::= x | y | z ...
Number ::= 0 | 1 | 2 ...
```

1 + 0 * 2

Exercise 3: Construct a parse tree

```
expr   ::= term
         |  expr + term
         |  expr - term

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

factor ::= NUMBER
         |  NAME
         |  ( expr )
```

1 + 0 * 2

Ambiguous grammar

- An ambiguous grammar is a formal grammar that can produce **multiple parse trees** or interpretations for the same input sentence or sequence of symbols.

- This can be problematic in various contexts because it can **make it difficult to determine the correct meaning** or parse tree of a sentence or sequence of symbols.

- To avoid ambiguity, it is often necessary to **use unambiguous grammars** or to add rules or constraints to the ambiguous grammar to disambiguate the interpretations.

Summary

- Context free grammar concepts
- Parse tree
- Ambiguous grammar
- Grammar -> Language
- Language -> Grammar