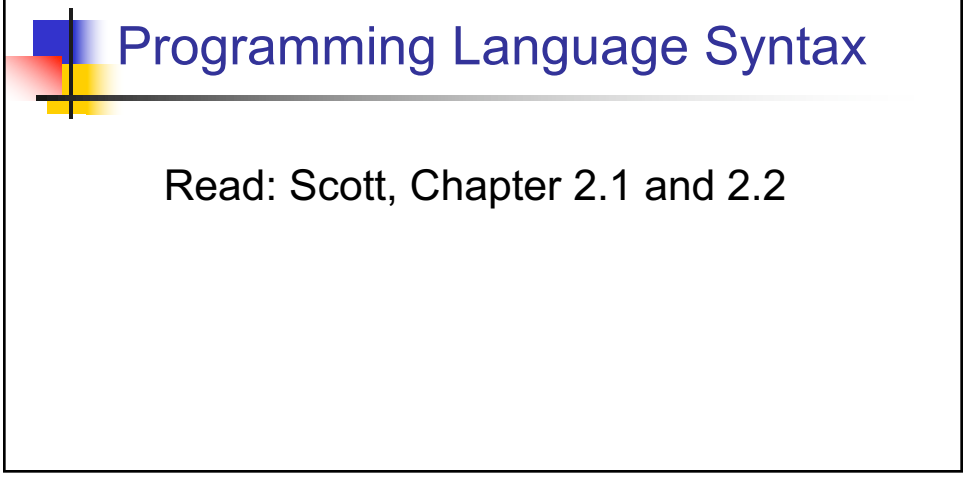
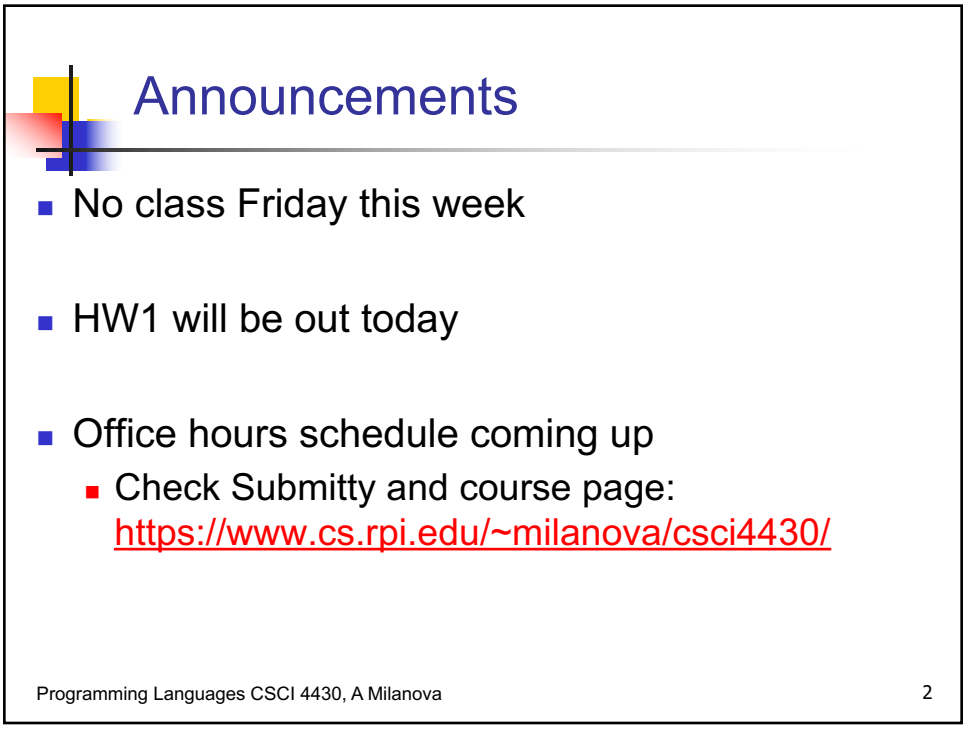

Programming Language Syntax
Read: Scott, Chapter 2.1 and 2.2

1


Announcements
No class Friday this week
HW1 will be out today
Office hours schedule coming up
Check Submitty and course page:
https://www.cs.rpi.edu/~milanova/csci4430/
Programming Languages CSCI 4430, A Milanova
2

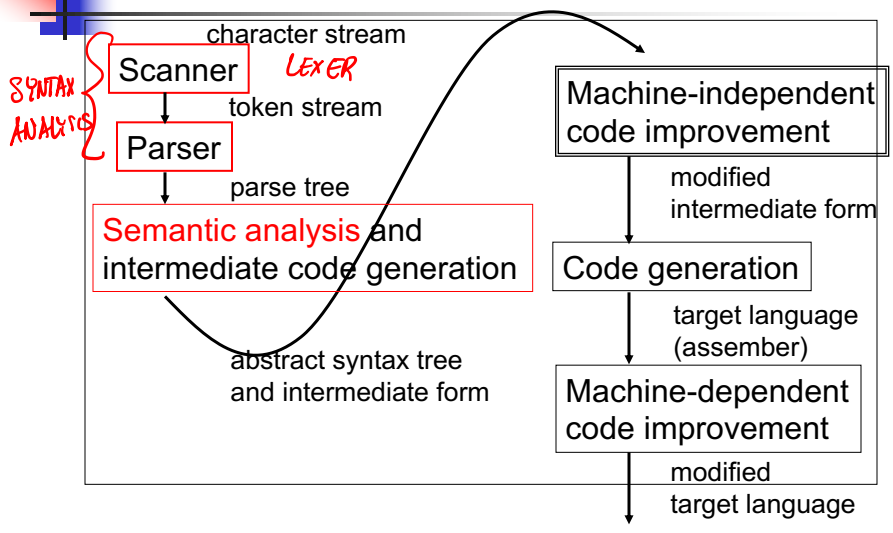
2

Lecture Outline

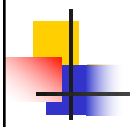
- *Formal languages*
- *Regular expressions*
- *Context-free grammars*
  - *Derivation*
  - *Parse*
  - *Parse trees*
  - *Ambiguity*
- *Expression grammars*

3

Last Class: Compiler



4

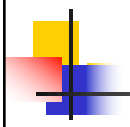


Syntax and Semantics

- Syntax is the form or structure of expressions, statements, and program units of a given language
  - Syntax of a Java **while** statement:
    - `while (` *boolean_expr* `)` *statement*
- Semantics is the meaning of expressions, statements and program units of a given language
  - Semantics of `while (` *boolean_expr* `)` *statement*
    - Execute *statement* repeatedly (0 or more times) as long as *boolean_expr* evaluates to `true`

5

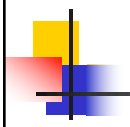
5



Formal Languages

- Theoretical foundations – Automata theory
- A language is a set of strings (also called sentences) over a finite alphabet
- A generator is a set of rules that generate the strings in the language
- A recognizer reads input strings and determines whether they belong to the language
- Languages are characterized by the complexity of generation/recognition rules

6



Question

- What are the classes of formal languages?
- The Chomsky hierarchy:
  - Regular languages
  - Context-free languages
  - Context-sensitive languages
  - Recursively enumerable languages

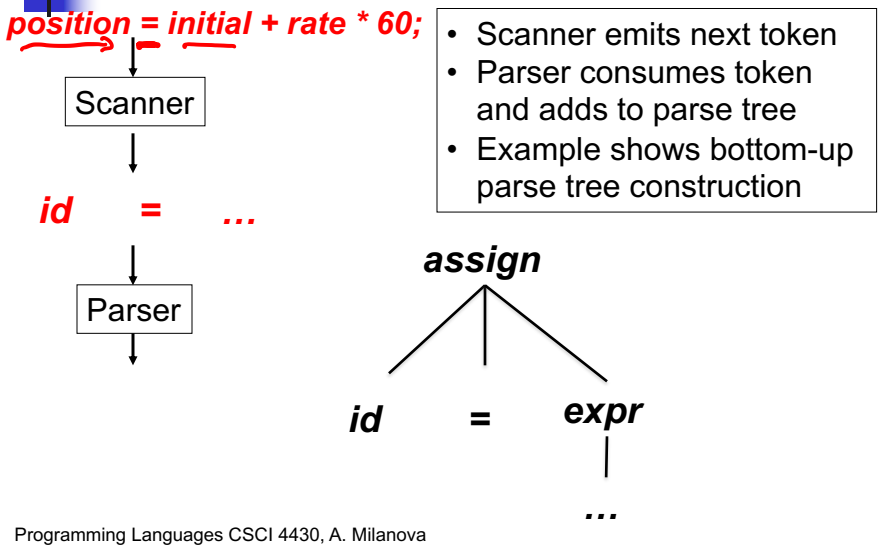
Formal Languages

- Generators and recognizers become more complex as languages become more complex
  - Regular languages
    - Describe PL tokens (e.g., keywords, identifiers, numeric literals)
    - Generated by a regular expression
    - Recognized by a finite automaton (scanner)
  - Context-free languages
    - Describe more complex PL constructs (e.g., expressions and statements)
    - Generated by a context-free grammar
    - Recognized by a push-down automaton (parser)
  - Even more complex constructs

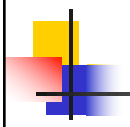
Formal Languages

- Key application of formal languages: enable proof of relative difficulty of computational problems
- Our focus: formal languages provide the formalism for describing PL constructs
  - A compelling application of formal languages!
  - Building a scanner
  - Building a parser
  - Central issue: build efficient, linear-time parsers

A Single Pass

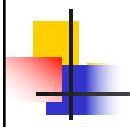


- Scanner emits next token
- Parser consumes token and adds to parse tree
- Example shows bottom-up parse tree construction



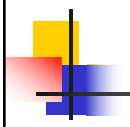
Lecture Outline

- *Formal languages*
- *Regular expressions*
- *Context-free grammars*
  - *Derivation*
  - *Parse*
  - *Parse trees*
  - *Ambiguity*
- *Expression grammars*



Regular Expressions

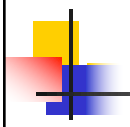
- Simplest structure
- Formalism to describe the simplest programming language constructs, the tokens
  - Each symbol (e.g., “+”, “-”) is a token
  - An identifier (e.g., position, rate, initial) is a token
  - A numeric constant (e.g., 60) is a token
- Recognized by a finite automaton



Regular Expressions

- A regular expression is one of the following:

  Base case:
  - A character, e.g., **a**
  - The empty string, denoted by ε



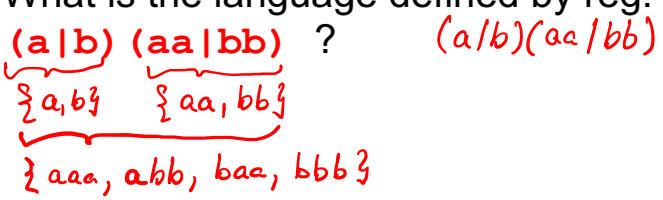
Regular Expressions

- A regular expression is one of the following:

  Inductive case:
  - Two regular expressions next to each other: $R_1\ R_2$
    - Meaning: $R_1\ R_2$ generates the language of strings that are made up of any string generated by R_1, followed by any string generated by R_2
  - Two regular expressions separated by |: $R_1 \mid R_2$
    - Meaning: $R_1 \mid R_2$ generates the language that is the union of the strings generated by R_1 with the strings generated by R_2
  - ?

Question

- What is the language defined by reg. exp. `(a|b)(aa|bb)` ?



- We saw concatenation and alternation. What operation is still missing?

Regular Expressions

- A Regular Expression is one of the following:
  - A character, e.g., `a`
  - The empty string `ε`
  - R_1 R_2
  - R_1 | R_2
  - Regular expression followed by a Kleene star, R*
    - Meaning: the concatenation of zero or more strings generated by R
    - E.g., `a*` generates {`ε, a, aa, aaa,` … }
    - E.g., `(a|b)*` generates all strings of `a`'s and `b`'s

Regular Expressions

- Precedence
  - Kleene * has highest precedence
  - Followed by concatenation
  - Followed by alternation |

  - E.g., `ab|c` is `(ab)|c` not `a(b|c)`
    - Generates language `{ab,c}` not `{ab,ac}`
  - E.g., `ab*` generates `{a,ab,abb,…}` not `{ε, ab, abab, ababab,…}`

((ab)|c)) (a(b*))

ab* is a(b*)

ab|c is (ab)|c

Question

- What is the language generated by regular expression `(0|1)*1` ?

ALL STRINGS OF 0'S AND 1'S THAT END IN 1.

0101

- What about `0*(10*10*)*` ?

ALL STRINGS OF 0'S AND 1'S WITH EVEN NUMBER OF 1'S.

E.g. 11, 10001 are in the language. 1 is not.

Regular Expressions in Programming Languages

- Describe tokens
- Let

  letter → **a | b | c |** … **| z**

  digit → **1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0**

- Which token is this?

  1. *letter* (*letter* | *digit*)* ? IDENTIFIER
  2. *digit digit* * ? INTEGER CONSTANT (NON NEG)
  3. *digit* *. *digit digit* * ? REAL CONSTANT (NON NEG)

Regular Expressions in Programming Languages

- Which token is this:

6.02 E+23 ≈ 6.02 × 10^{23}
decimal exponent

6.02

6.02 is recognized by "decimal"'s automaton

number → *integer* | *real*

real → *integer exponent* | *decimal* (*exponent* | **ε**)

decimal → *digit** (**.** *digit* | *digit* **.**) *digit**

exponent → (**e** | **E**) (**+** | **-** | **ε**) *integer*

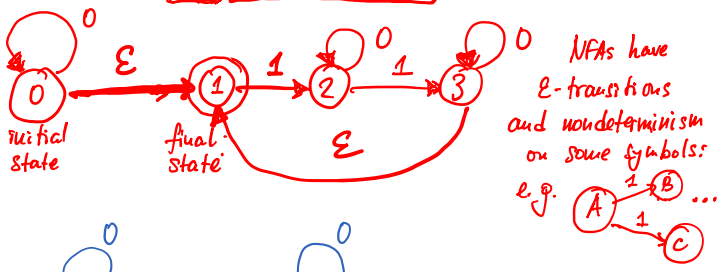
integer → *digit digit**

digit → **1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0**

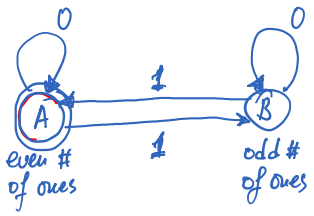
Exercises

Regular expression: `0*(10*10*)*`

NFA:



DFA:



Lecture Outline

- *Formal languages*
- *Regular expressions*
- *Context-free grammars*
  - *Derivation*
  - *Parse*
  - *Parse trees*
  - *Ambiguity*
- *Expression grammars*

Context-Free Grammars

- Unfortunately, regular languages cannot specify all constructs in programming
- E.g., can we write a regular expression that specifies valid arithmetic expressions?
  - `id * ( id + id * ( number - id ) )`
  - Among other things, we need to ensure that parentheses are matched!
  - Answer is no. We need context-free languages and context-free grammars!

Grammar

- A grammar is a formalism to describe the strings of a (formal) language
- A grammar consists of a set of terminals, set of nonterminals, a set of productions, and a start symbol
  - Terminals are the characters in the alphabet
  - Nonterminals represent language constructs
  - Productions are rules for forming syntactically correct constructs
  - Start symbol tells where to start applying the rules

Notation

A → A0 | B0
B → A1 | B1
A → ε
B → ε

Specification of identifier:

DESCRIPTIVE NAMES FOR NONTERMINALS

Regular expression: *letter* (*letter* | *digit*)*

BNF: <*digit*> ::= *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *0*
<*letter*> ::= *a* | *b* | *c* | … | *x* | *y* | *z*
<*id*> ::= <*letter*> | <*id*> <*letter*> | <*id*> <*digit*>

Textbook and slides: (also BNF)

digit → **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **0**
letter → **a** | **b** | **c** | **d** | … | **z**
id → *letter* | *id letter* | *id digit*

Nonterminals shown in *italic*

Terminals shown in **typewriter**

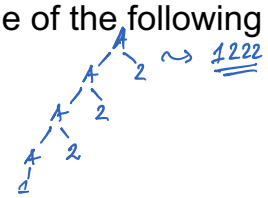
Regular Grammars

- Regular grammars generate regular languages
- The rules in regular grammars are of the form:
  - Each left-hand-side (lhs) has exactly one nonterminal
  - Each right-hand-side (rhs) is one of the following
    - A single terminal symbol or
    - A single nonterminal symbol or
    - A nonterminal followed by a terminal

e.g., **1** **2*** | **0**+

KLEENE + : 00*

S → *A* | *B*
A → **1** | *A* **2**
B → **0** | *B* **0**



"A" GENERATES 12*

"B" GENERATES 00*

Question

- Is this a regular grammar:

$S \rightarrow 0\ A$
$A \rightarrow S\ 1$
$S \rightarrow \varepsilon$

$S \Rightarrow 0A \Rightarrow 0S1 \Rightarrow 01$
(GENERATED STRING 01)

  - No, this is a context-free grammar
    - It generates 0^n1^n, the canonical example of a context-free language
    - rhs should be nonterminal followed by a terminal, thus, $S \rightarrow 0\ A$ is not a valid production

Lecture Outline

- *Formal languages*
- *Regular expressions*
- *Context-free grammars*
  - *Derivation*
  - *Parse*
  - *Parse trees*
  - *Ambiguity*
- *Expression grammars*

Context-free Grammars (CFGs)

- Context-free grammars generate context-free languages
  - Most of what we need in programming languages can be specified with CFGs
- Context-free grammars have rules of the form:
  - Each left-hand-side has exactly one nonterminal
  - Each right-hand-side contains an arbitrary sequence of terminals and nonterminals
- A context-free grammar
  e.g., $\mathbf{0}^n\mathbf{1}^n$, $n \geq 1$ $S \rightarrow \mathbf{0}\ S\ \mathbf{1}$
  $S \rightarrow \mathbf{0}\ \mathbf{1}$

A SINGLE NONTERMINAL AT LHS.

A SEQUENCE OF TERMINALS AND NONTERMINALS AT RHS.

29

29

Question

- Examples of non-context-free languages?
  - E.g., $\mathbf{a}^n\mathbf{b}^m\mathbf{c}^n\mathbf{d}^m$ $n \geq 1, m \geq 1$
  - E.g., $w\mathbf{c}w$ where w is in `(0|1)*`
  - E.g., $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ $n \geq 1$ (canonical example)

Context-free Grammars

- Can be used to generate strings in the context-free language (derivation)
- Can be used to recognize well-formed strings in the context-free language (parse)
- In programming languages and compilers, we are concerned with two special CFGs, called LL and LR grammars

Derivation

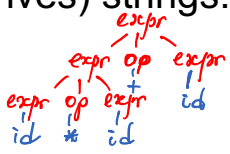
Simple context-free grammar for expressions:

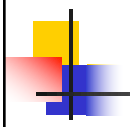
expr → **id** | **(** *expr* **)** | *expr op expr*

op → **+** | *****

A derivation generates (derives) strings. E.g.,:

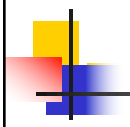
expr ⇒ *expr op* *expr*
⇒ *expr* *op* **id**
⇒ *expr* **+ id**
⇒ *expr op* *expr* **+ id** ← sentential form
⇒ *expr* *op* **id + id**
⇒ *expr* *** id + id**
⇒ **id * id + id** ← sentence, string or yield





Derivation

- A derivation starts from the start symbol and at each step replaces a nonterminal with the right-hand side of a production for that nonterminal
  - E.g., *expr op* *expr* derives *expr op* **`id`**
    We replaced the right (underlined) *expr* with **`id`**
    due to production *expr* → **`id`**
- An intermediate sentence is called a sentential form
  - E.g., *expr op* **`id`** is a sentential form



Derivation

- Resulting sentence is called yield
  - E.g., **`id*id+id`** is the yield of our derivation
- What is a leftmost derivation?
  - Replaces the leftmost nonterminal in the sentential form at each step
- What is a rightmost derivation?
  - Replaces the rightmost nonterminal in the sentential form at each step
- Some derivations are neither left- nor rightmost

Question

- What kind of derivation is this:

expr ⇒ *expr op* *expr*
⇒ *expr* *op* `id`
⇒ *expr* `+ id`
⇒ *expr op* *expr* `+ id`
⇒ *expr* *op* `id + id`
⇒ *expr* `* id + id`
⇒ `id * id + id`

  - A rightmost derivation. At each step we replace the rightmost nonterminal

Question

- What kind of derivation is this:

expr ⇒ *expr op* *expr*
⇒ *expr* *op* `id`
⇒ *expr* `+ id`
⇒ *expr op expr* `+ id`
⇒ `id` *op* *expr* `+ id`
⇒ `id` *op* `id + id`
⇒ `id * id + id`

  - Neither leftmost nor rightmost

Parse

PROCESS OF RECOGNIZING A STRING AS BELONGING TO THE LANGUAGE

expr → **id** | **(** *expr* **)** | *expr op expr*

op → **+** | *****

- A parse takes a string and constructs a derivation (if one exists). E.g.:

id * id + id ⇒ *expr* *** id + id**
⇒ *expr op* **id + id**
⇒ *expr op expr* **+ id**
⇒ *expr* **+ id**
⇒ *expr op* **id**
⇒ *expr op expr*
⇒ *expr*

Parse

- A bottom-up parse builds parse tree bottom-up, i.e., from leaves towards root (start symbol)
- Example was a bottom-up parse:

… ⇒ *expr op expr* **+ id**
⇒ *expr* **+ id**

Replacing *expr op expr* with *expr* corresponds *to* adding node *expr* to tree

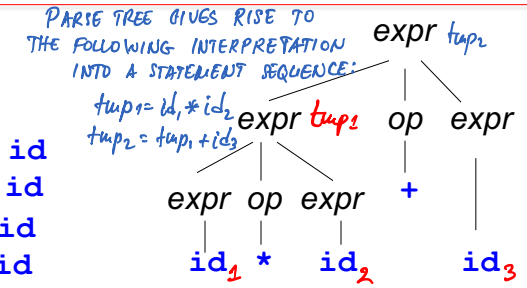
- A top-down parse builds parse tree top-down, i.e., from root towards leaves (a lot more on this later)

Parse Tree

expr → **id** | **(** *expr* **)** | *expr op expr*

op → **+** | *****

expr ⇒ *expr op* <u>*expr*</u>
⇒ *expr* <u>*op*</u> **id**
⇒ <u>*expr*</u> **+ id**
⇒ *expr op* <u>*expr*</u> **+ id**
⇒ *expr* <u>*op*</u> **id + id**
⇒ <u>*expr*</u> *** id + id**
⇒ **id * id + id**



Internal nodes are nonterminals. Children are the rhs of a rule for that nonterminal.
Leaf nodes are terminals.

39

39

Ambiguity

- A grammar is ambiguous if some string can be generated by two or more distinct parse trees
  - There is no algorithm that can tell if an arbitrary context-free grammar is ambiguous
- Ambiguity arises in programming language grammars
  - Arithmetic expressions
  - If-then-else: the dangling else problem
- Ambiguity is bad

40

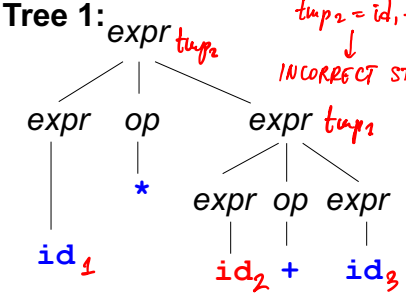
Ambiguity

expr → **id** | **(** *expr* **)** | *expr op expr*

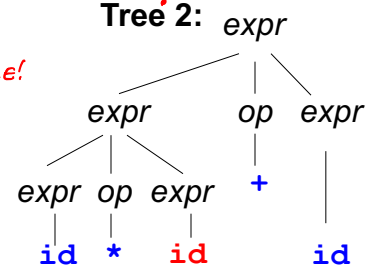
op → **+** | *****

- How many parse trees for **id * id + id** ?

Tree 1:



Tree 2:



- Which one is "correct"?

41

41

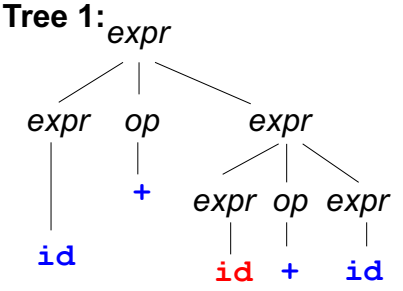
Ambiguity

expr → **id** | **(** *expr* **)** | *expr op expr*

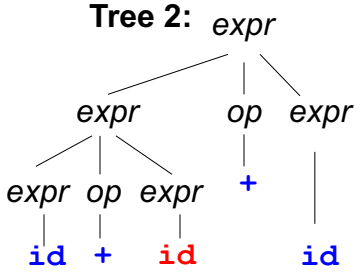
op → **+** | *****

- How many parse trees for **id + id + id** ?

Tree 1:



Tree 2:



- Which one is "correct"?

42

42

Lecture Outline

- *Formal languages*
- *Regular expressions*
- *Context-free grammars*
  - *Derivation*
  - *Parse*
  - *Parse trees*
  - *Ambiguity*
- *Expression grammars*

Expression Grammars

- Generate expressions
  - Arithmetic expressions
  - Regular expressions
  - Other
- Terminals: operands, operators, and parentheses

expr → **`id`** | **`(`** *expr* **`)`** | *expr op expr*

op → **`+`** | **`*`**

Handling Ambiguity

Our ambiguous grammar, slightly simplified:

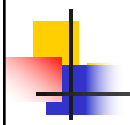
expr → **id** | **(** *expr* **)** | *expr* **+** *expr* | *expr* ***** *expr*

- Rewrite the grammar into unambiguous one:

  expr → *expr* **+** *term* | *term*

  term → *term* ***** *factor* | *factor*

  factor → **id** | **(** *expr* **)**

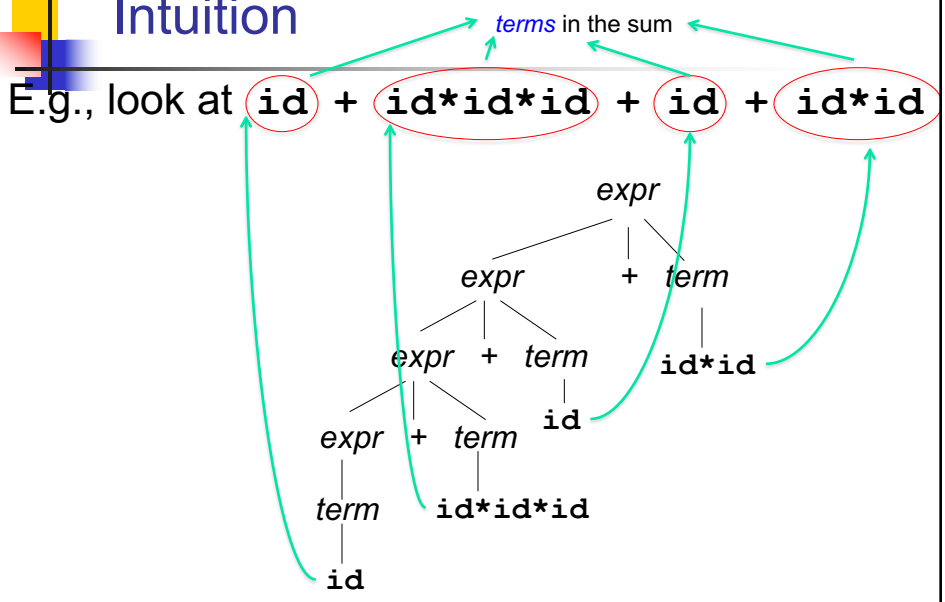


Rewriting Expression Grammars: Intuition

expr → **id** | **(** *expr* **)** | *expr* **+** *expr* | *expr* ***** *expr*

- A new nonterminal, *term*
- *expr* ***** *expr* becomes *term*. Thus, ***** gets pushed down the tree, forcing higher precedence of *****
- *expr* **+** ***expr*** *becomes expr* **+** *term*. Pushes leftmost **+** down the tree, forcing operand to associate with **+** on its left
  - *expr* → *expr* **+** *expr* becomes *expr* → *expr* **+** *term* | *term*

Rewriting Expression Grammars: Intuition



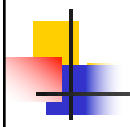
Rewriting Expression Grammars: Intuition

- To parse terms, we add a new nonterminal, *factor,* and productions:
  - *term* → *term* ***** *factor* | *factor*
  - *factor* → **id** | (*expr*)

Exercise

expr → *expr* **×** *expr* | *expr* **^** *expr* | **id**

- How many parse trees for **id × id^id × id** ?
  - No need to draw them all
- Rewrite above grammar into an equivalent unambiguous grammar such that
  - **^** has higher precedence than **×**
  - **^** is right-associative
  - **×** is left-associative



The End