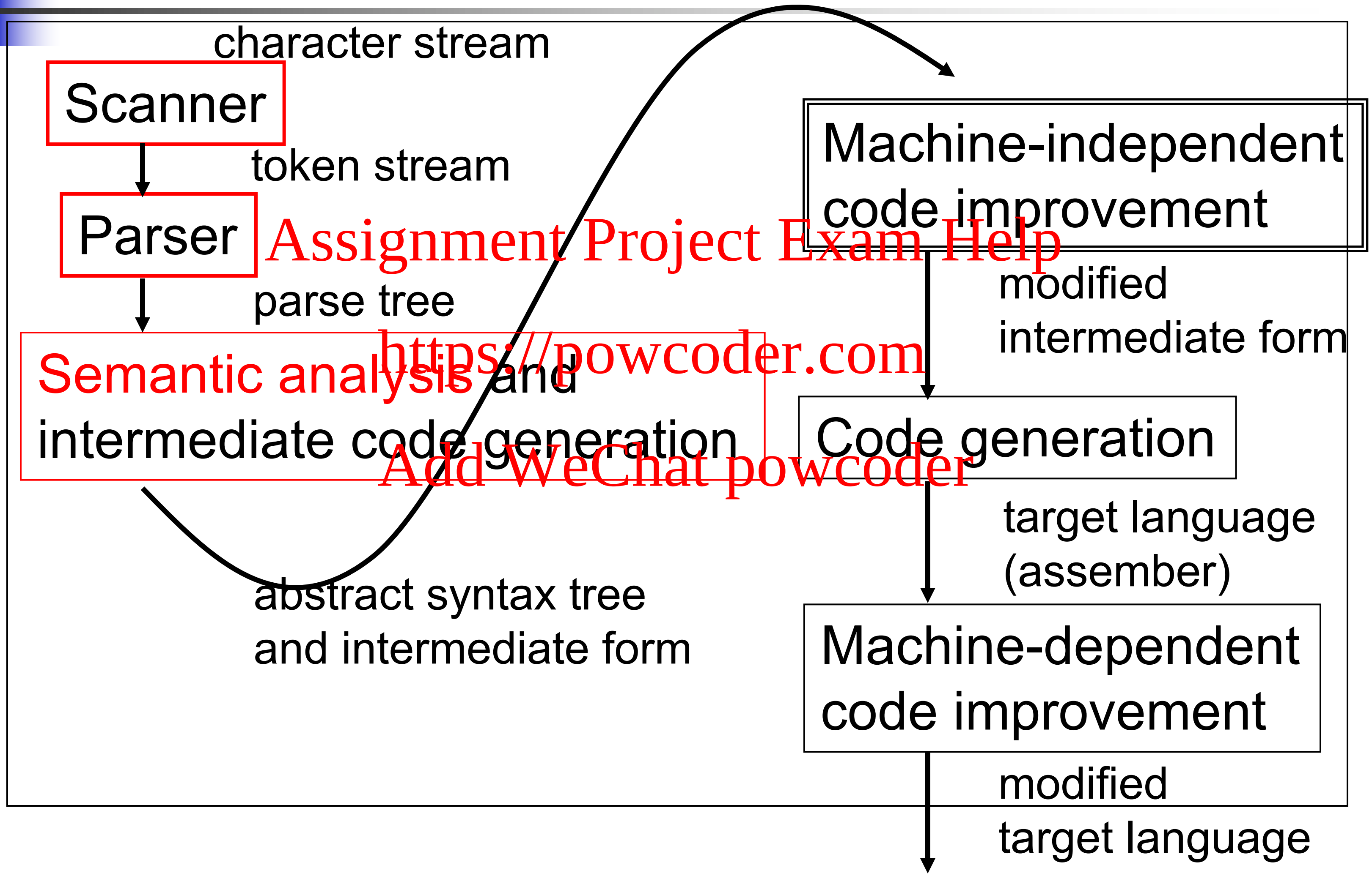
Programming Language Syntax

Read: Scott, Chapter 2.1

Lecture Outline

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

Last Class: Compiler



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`**

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
  - E.g., regular languages
  - E.g., context-free languages

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 Regular Expressions

    - 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

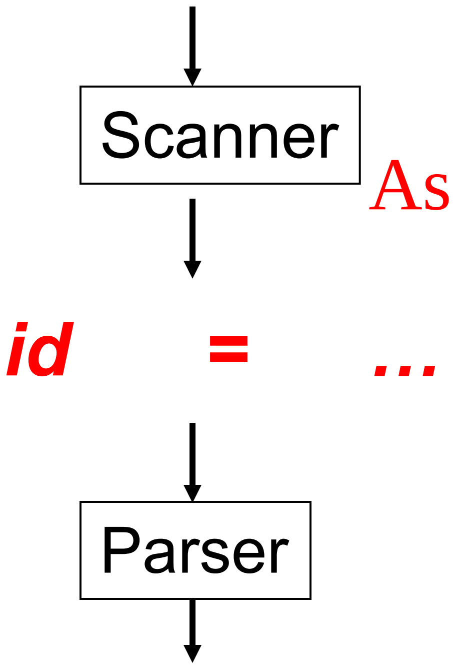
Formal Languages

- Main application of formal languages: enable proof of relative difficulty of computational problems

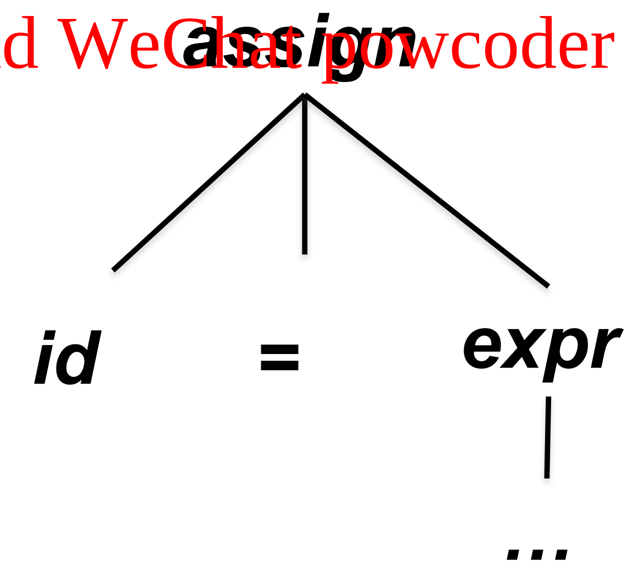
- Our focus: formal languages provide the formalism for describing PL constructs
  - A compelling application of formal languages!
  - Building a scanner
  - Building a parser
  - Central issue: build efficient, linear-time parsers

A Single Pass

position = initial + rate * 60;



- Scanner emits next token
- Parser consumes the token and continues building the parse tree (typically bottom up)



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 symbols (e.g., "+", "-") is a token
  - an identifier (e.g., position, rate, initial) is a token
  - a numeric constant (e.g., 59) is a token
  - etc.
- Recognized by a finite automaton

Regular Expressions

- A Regular Expression is one of the following:
  - A character, e.g., **a**

  - The empty string, denoted by **ε**

  - 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) (a a | b b)`** ?

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

Question

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

- What about `0* (1 0* 1 0*)*` ?

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)** ?
  2. *digit digit ** ?
  3. *digit * . digit digit ** ?

Regular Expressions in Programming Languages

- Which token is this:

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**

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

Specification of identifier:

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)

$digit \rightarrow \mathtt{1} | \mathtt{2} | \mathtt{3} | \mathtt{4} | \mathtt{5} | \mathtt{6} | \mathtt{7} | \mathtt{8} | \mathtt{9} | \mathtt{0}$
$letter \rightarrow \mathtt{a} | \mathtt{b} | \mathtt{c} | \mathtt{d} | \ldots | \mathtt{z}$
$id \rightarrow letter \mid id\ letter \mid id\ digit$

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+`

$$S \rightarrow A \mid B$$
$$A \rightarrow \mathtt{1} \mid A\ \mathtt{2}$$
$$B \rightarrow \mathtt{0} \mid B\ \mathtt{0}$$

Question

- Is this a regular grammar:

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

- 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. $\mathbf{0}^n\mathbf{1}^n$,$n \geq 1$ $S \rightarrow \mathbf{0}\ S\ \mathbf{1}$

  $S \rightarrow \mathbf{0}\ \ \mathbf{1}$

Question

- Examples of a 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*`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 <u>generate</u> strings in the context-free language (derivation)

- Can be used to <u>recognize</u> 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:

$expr \rightarrow$ **id** | **(** $expr$ **)** | $expr\ op\ expr$

$op \rightarrow$ **+** | *****

We can generate (derive) expressions:

$$
\begin{aligned}
expr &\Rightarrow expr\ op\ \underline{expr} \\
&\Rightarrow expr\ \underline{op}\ \mathbf{id} \\
&\Rightarrow \underline{expr}\ \mathbf{+}\ \mathbf{id} \\
&\Rightarrow expr\ op\ \underline{expr}\ \mathbf{+}\ \mathbf{id} \quad \longleftarrow \text{sentential form} \\
&\Rightarrow expr\ \underline{op}\ \mathbf{id}\ \mathbf{+}\ \mathbf{id} \\
&\Rightarrow \underline{expr}\ \mathbf{*}\ \mathbf{id}\ \mathbf{+}\ \mathbf{id} \\
&\Rightarrow \mathbf{id}\ \mathbf{*}\ \mathbf{id}\ \mathbf{+}\ \mathbf{id} \quad \longleftarrow \text{sentence, string or yield}
\end{aligned}
$$

Derivation

- A derivation is the process that starts from the start symbol, and at each step, replaces a nonterminal with the right-hand-side of a production

  - E.g., *expr op <u>expr</u>* derives *expr op* **`id`**
    We replaced the right (underlined) *expr* with **`id`**
    due to production *expr* $\rightarrow$ **`id`**
- An intermediate sentence is called a sentential form
  - E.g., *expr op* **`id`** is a sentential form

Derivation

- The resulting sentence is called yield
  - E.g., `id*id+id` is the yield of our derivation

- What is a left-most derivation?

  - Replaces the left-most nonterminal in the sentential form at each step

- What is a right-most derivation?
  - Replaces the right-most nonterminal in the sentential form at each step
- There are derivations that are neither left- nor right-most

Question

- What kind of derivation is this:

$$
\begin{aligned}
\mathit{expr} &\Rightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}} \\
&\Rightarrow \mathit{expr}\ \underline{\mathit{op}}\ \mathbf{id} \\
&\Rightarrow \underline{\mathit{expr}}\ \mathbf{+}\ \mathbf{id} \\
&\Rightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}}\ \mathbf{+}\ \mathbf{id} \\
&\Rightarrow \mathit{expr}\ \underline{\mathit{op}}\ \mathbf{id}\ \mathbf{+}\ \mathbf{id} \\
&\Rightarrow \underline{\mathit{expr}}\ \mathbf{*}\ \mathbf{id}\ \mathbf{+}\ \mathbf{id} \\
&\Rightarrow \mathbf{id}\ \mathbf{*}\ \mathbf{id}\ \mathbf{+}\ \mathbf{id}
\end{aligned}
$$

  - A right-most derivation. At each step we replace the right-most nonterminal

Question

- What kind of derivation is this:

$$
\begin{aligned}
\mathit{expr} &\Rightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}} \\
&\Rightarrow \mathit{expr}\ \underline{\mathit{op}}\ \mathbf{id} \\
&\Rightarrow \underline{\mathit{expr}}\ +\ \mathbf{id} \\
&\Rightarrow \underline{\mathit{expr}}\ \mathit{op}\ \mathit{expr}\ +\ \mathbf{id} \\
&\Rightarrow \mathbf{id}\ \mathit{op}\ \underline{\mathit{expr}}\ +\ \mathbf{id} \\
&\Rightarrow \mathbf{id}\ \underline{\mathit{op}}\ \mathbf{id}\ +\ \mathbf{id} \\
&\Rightarrow \mathbf{id}\ *\ \mathbf{id}\ +\ \mathbf{id}
\end{aligned}
$$

  - Neither left-most nor right-most

Parse

Recall our context-free grammar for expressions:

$\mathit{expr} \rightarrow \texttt{id} \mid \texttt{(}\ \mathit{expr}\ \texttt{)} \mid \mathit{expr}\ \mathit{op}\ \mathit{expr}$

$\mathit{op} \rightarrow \texttt{+} \mid \texttt{*}$

- A parse is the reverse of a derivation

$$\texttt{id * id + id} \Rightarrow \mathit{expr}\ \texttt{* id + id}$$

$$\Rightarrow \mathit{expr}\ \mathit{op}\ \underline{\texttt{id}}\ \texttt{+ id}$$
$$\Rightarrow \underline{\mathit{expr}\ \mathit{op}\ \mathit{expr}}\ \texttt{+ id}$$
$$\Rightarrow \mathit{expr}\ \underline{\texttt{+}}\ \texttt{id}$$
$$\Rightarrow \mathit{expr}\ \mathit{op}\ \underline{\texttt{id}}$$
$$\Rightarrow \underline{\mathit{expr}\ \mathit{op}\ \mathit{expr}}$$
$$\Rightarrow \mathit{expr}$$

Parse

- A parse starts with the string of terminals, and at each step, replaces the right-hand-side (rhs) of a production with the left-hand-side (lhs) of that production. E.g.,

$$\ldots \Rightarrow \underline{\mathit{expr\ op\ expr}} + \mathbf{id}$$

$$\Rightarrow \mathit{expr} + \mathbf{id}$$

Here we replaced *expr op expr* (the rhs of production *expr* $\rightarrow$ *expr op expr*) with *expr* (the lhs of the production)

Parse Tree

$expr \rightarrow$ **id** | **(** $expr$ **)** | $expr\ op\ expr$

$op \rightarrow$ **+** | *****

$expr \Rightarrow expr\ op\ \underline{expr}$

$\Rightarrow expr\ \underline{op}\ \mathbf{id}$

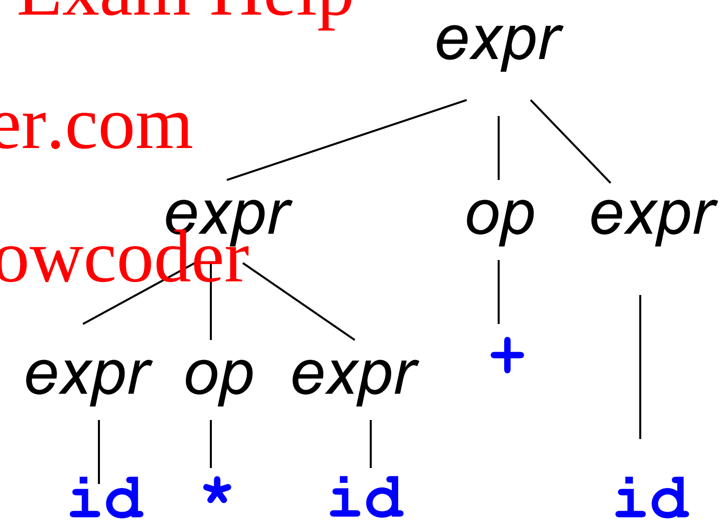
$\Rightarrow \underline{expr}\ \mathbf{+}\ \mathbf{id}$

$\Rightarrow expr\ op\ \underline{expr}\ \mathbf{+}\ \mathbf{id}$

$\Rightarrow expr\ \underline{op}\ \mathbf{id}\ \mathbf{+}\ \mathbf{id}$

$\Rightarrow \underline{expr}\ \mathbf{*}\ \mathbf{id}\ \mathbf{+}\ \mathbf{id}$

$\Rightarrow \mathbf{id}\ \mathbf{*}\ \mathbf{id}\ \mathbf{+}\ \mathbf{id}$



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

Ambiguity

- 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

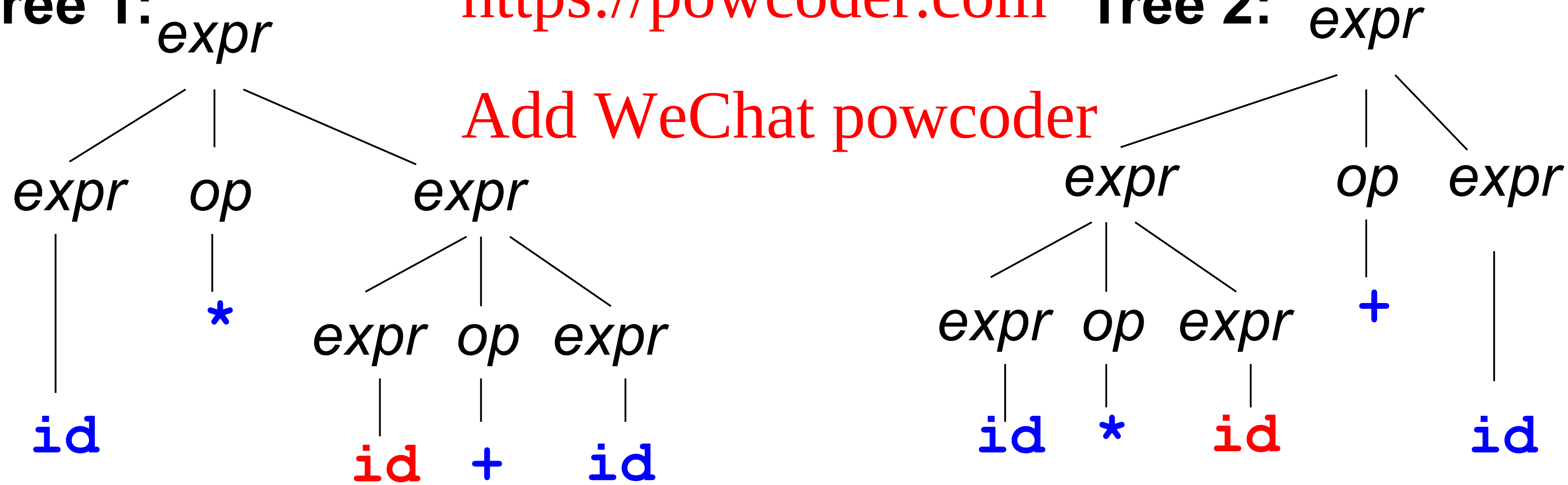
Ambiguity

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

op → **+** | *****

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

Tree 1: **Tree 2:**



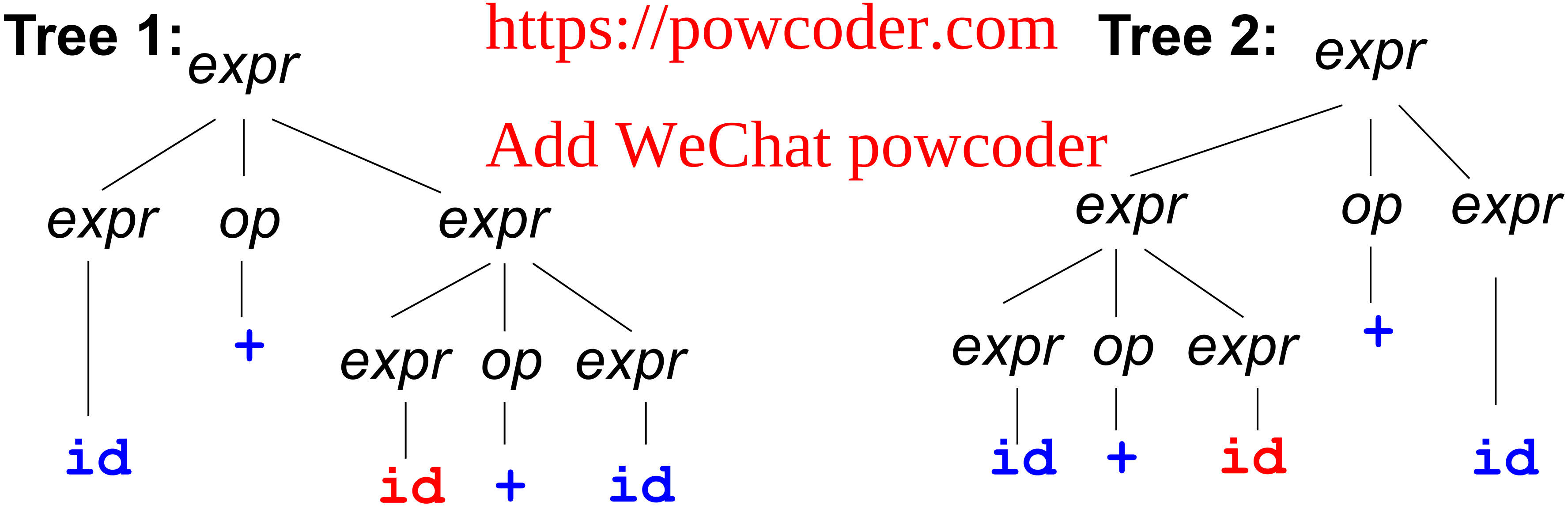
- Which one is "correct"?

Ambiguity

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

op → `+` | `*`

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



- Which one is "correct"?

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

$$expr \rightarrow \mathbf{id} \mid \mathbf{(}\ expr\ \mathbf{)} \mid expr\ op\ expr$$

$$op \rightarrow \mathbf{+} \mid \mathbf{*}$$

Handling Ambiguity

Our ambiguous grammar, slightly simplified:

$expr \rightarrow$ **id** | **(** $expr$ **)** | $expr$ **+** $expr$ | $expr$ ***** $expr$

- Rewrite the grammar into unambiguous one:

$expr \rightarrow expr$ **+** $term$ | $term$

$term \rightarrow term$ ***** $factor$ | $factor$

$factor \rightarrow$ **id** | **(** $expr$ **)**

- Forces left associativity of **+** and *****
- Forces higher precedence of ***** over **+**

Rewriting Expression Grammars: Intuition

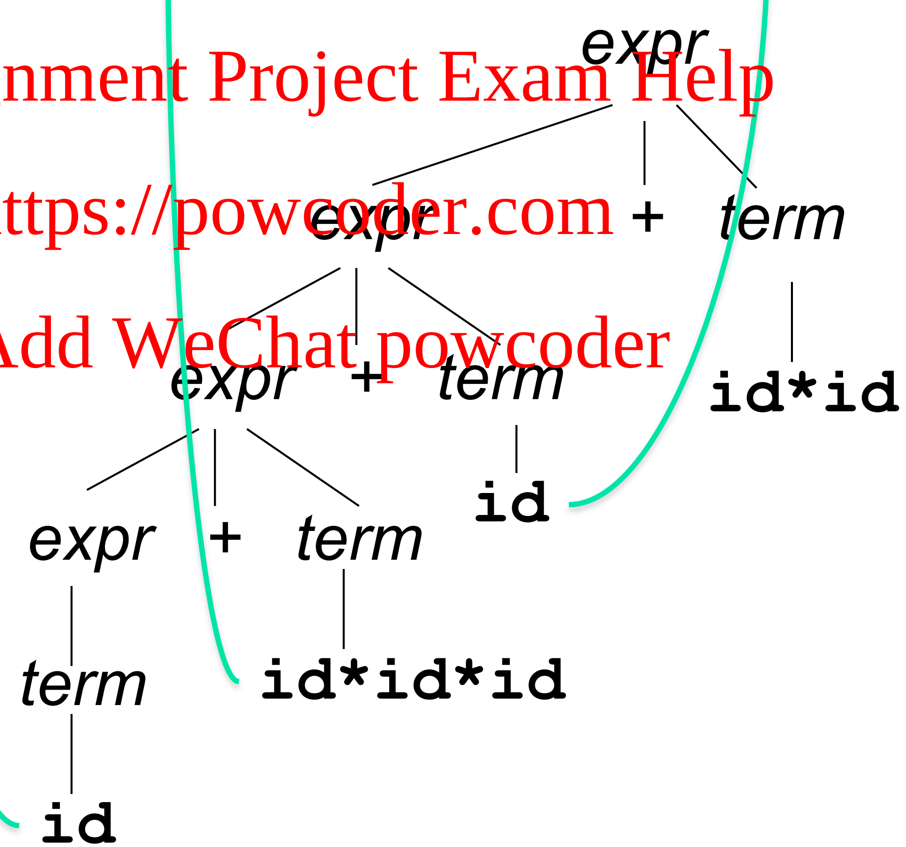
expr → **id** | **(** *expr* **)** | *expr* **+** *expr* | *expr* ***** *expr*

- 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

terms in the sum

E.g., look at `id + id*id*id + id + id*id`



Rewriting Expression Grammars: Intuition

- Another new nonterminal, *factor* and productions:
  - *term* → *term* **`*`** *factor* | *factor*
  - *factor* → **`id`** | (*expr*)

Exercise

$expr \rightarrow expr$ **×** $expr \mid expr$ **^** $expr \mid$ **id**

- How many parse trees for **id×id^id×id** ?

  - No need to draw them all

- Rewrite this grammar into an equivalent unambiguous grammar where
  - ^ has higher precedence than ×
  - ^ is right-associative
    - × is left-associative

The End

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder