CS 314 Principles of Programming Languages

Lecture 5: Syntax Analysis (Parsing)

Zheng (Eddy) Zhang

Rutgers University

January 31, 2018

Class Information

- Homework 1 is being graded now.
  The sample solution will be posted soon.
- Homework 2 posted. Due in one week.

Review: Context Free Grammars (CFGs)

- A formalism to for describing languages
- A CFG $\mathcal{G}$ = < **T**, **N**, **P**, **S** >:
  1. A set T of terminal symbols (tokens).
  2. A set N of nonterminal symbols.
  3. A set P production (rewrite) rules.
  4. A special start symbol S.
- The language L($\mathcal{G}$) is the set of sentences of terminal symbols in T* that can be derived from the start symbol S:

$$L(\mathcal{G}) = \{w \in T^* \mid S \Rightarrow^* w\}$$

Review: Context Free Grammar

…

<if-stmt> ::= **if** <expr> **then** <stmt>
<expr> ::= **id <= id**
<stmt> ::= **id := num**

…

Context free grammar

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

Not a context free grammar

CFGs are rewrite systems with restrictions on the form of rewrite (production) rules that can be used. The left hand side of a production rule can only be **one non-terminal symbol**.

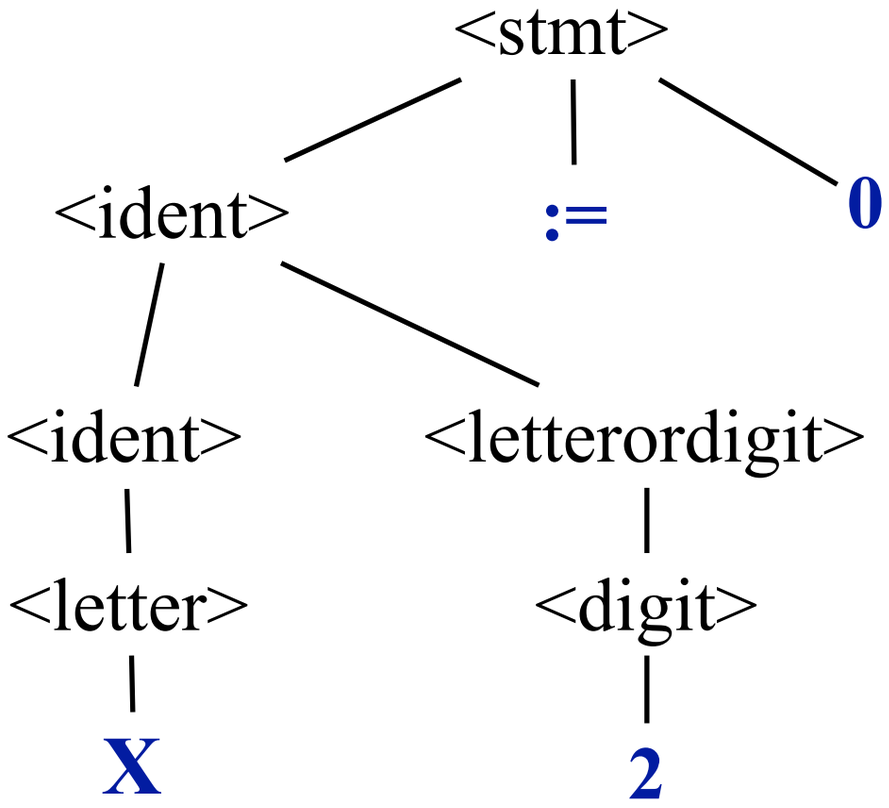
A Language May Have Many Grammars

Consider $\mathcal{G}$':

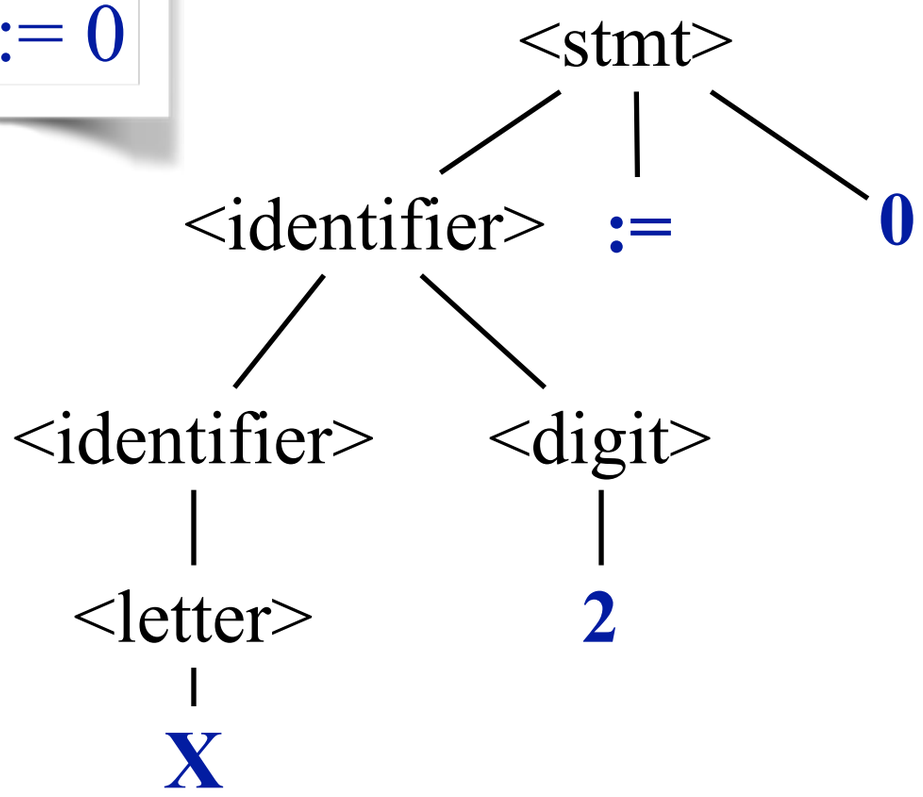
1. < letter > ::= **A** | **B** | **C** | … | **Z**
2. < digit > ::= **0** | **1** | **2** | … | **9**
3. < ident > ::= < letter > |
4. < ident > < letterordigit >
5. < stmt > ::= < ident > **:=** 0
6. < letterordigit > ::= < letter > | < digit >

The Original Grammar $\mathcal{G}$:

1. < letter > ::= **A** | **B** | **C** | … | **Z**
2. < digit > ::= **0** | **1** | **2** | … | **9**
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > **:=** 0



X2 := 0



Review: Grammars and Programming Languages

Many grammars may correspond to one programming language.

Good grammars:

- Captures the logic structure of the language

  ⇒ structure carries some semantic information
  (example: expression grammar)
- Use meaningful names
- Are easy to read
- Are unambiguous
- …

Review: Ambiguous Grammars

*"Time **flies like** an arrow; fruit **flies like** a banana."*

A grammar $\mathcal{G}$ is ambiguous iff there exists a w $\in$ L($\mathcal{G}$) such that there are:

- two distinct parse trees for w, or
- two distinct leftmost derivations for w, or
- two distinct rightmost derivations for w.

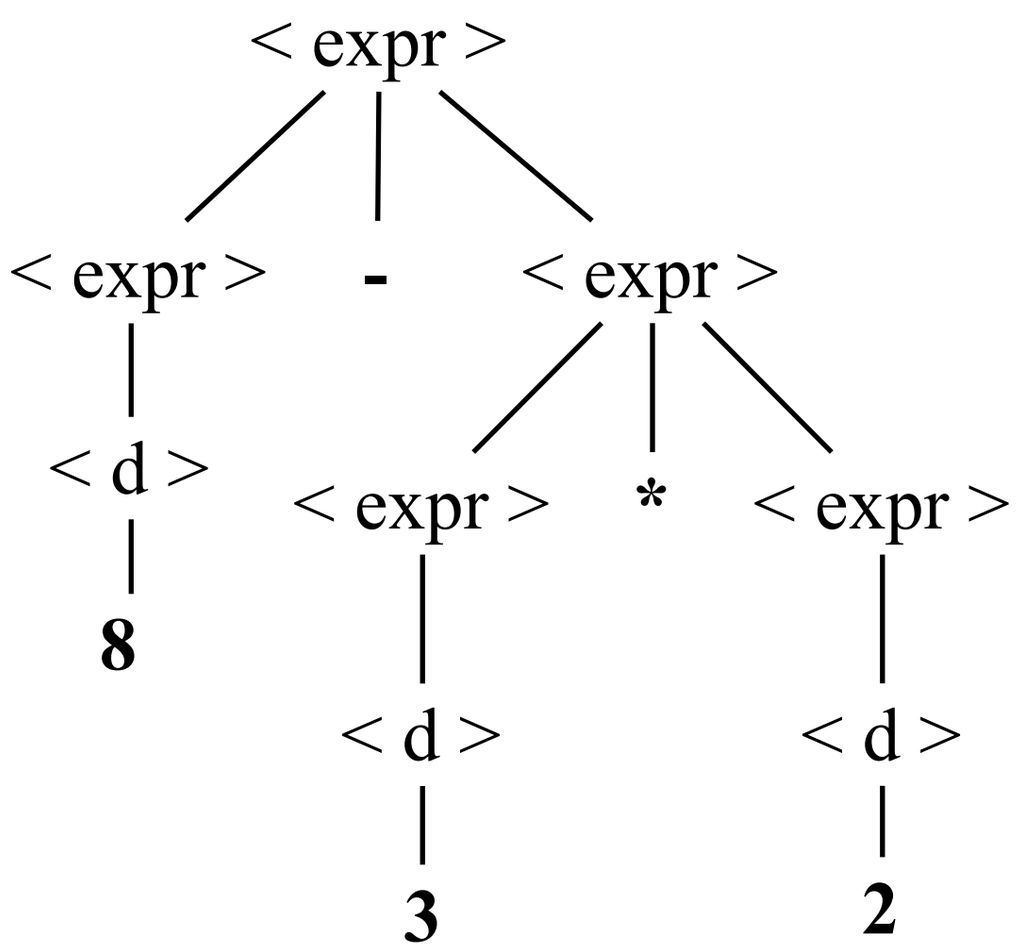
*We want **a unique semantics** of our programs, which typically requires **a unique syntactic structure**.*

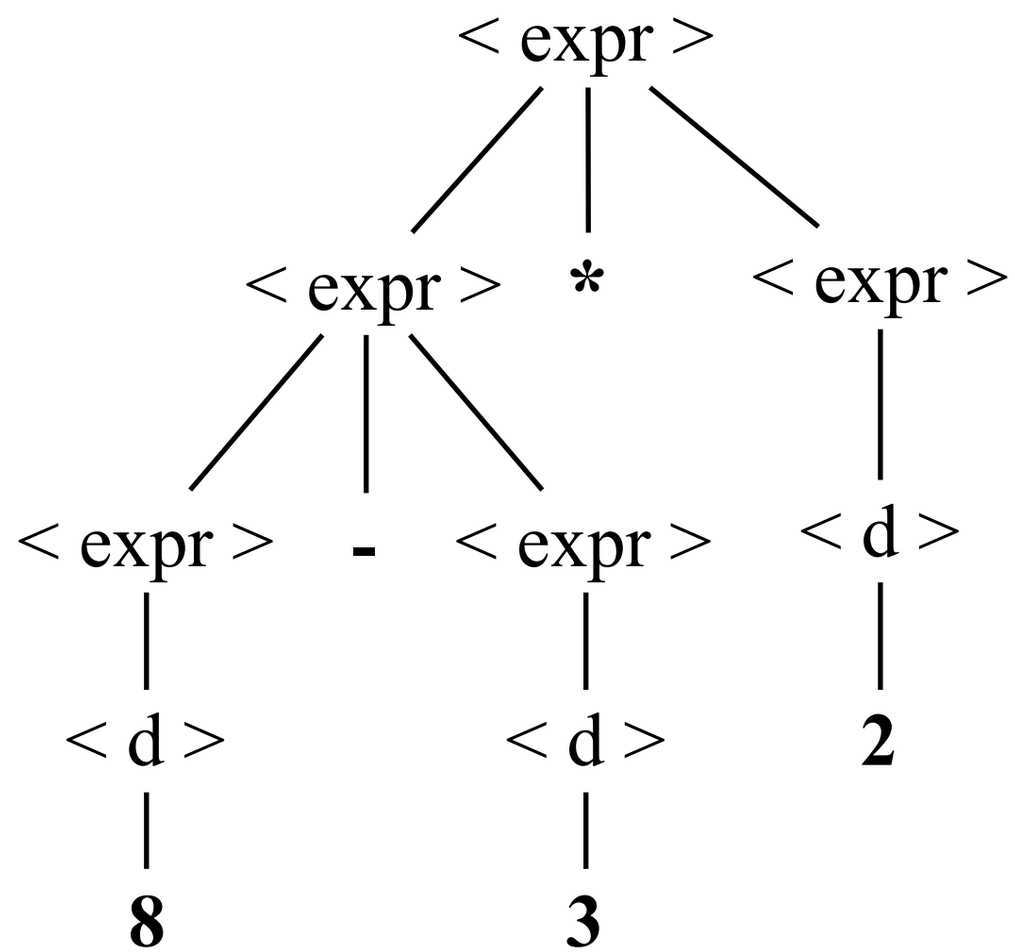
Review: Arithmetic Expression Grammar

Parse "8 - 3 * 2":

```
< start > ::=  < expr >
< expr > :: = < expr > - < expr >  |
              < expr > * < expr > |
              < d > | < l >
< d >     :: = 0 | 1 | 2 | 3 | … | 9
< l >     :: = a | b | c | … | z
```

Two parse trees!

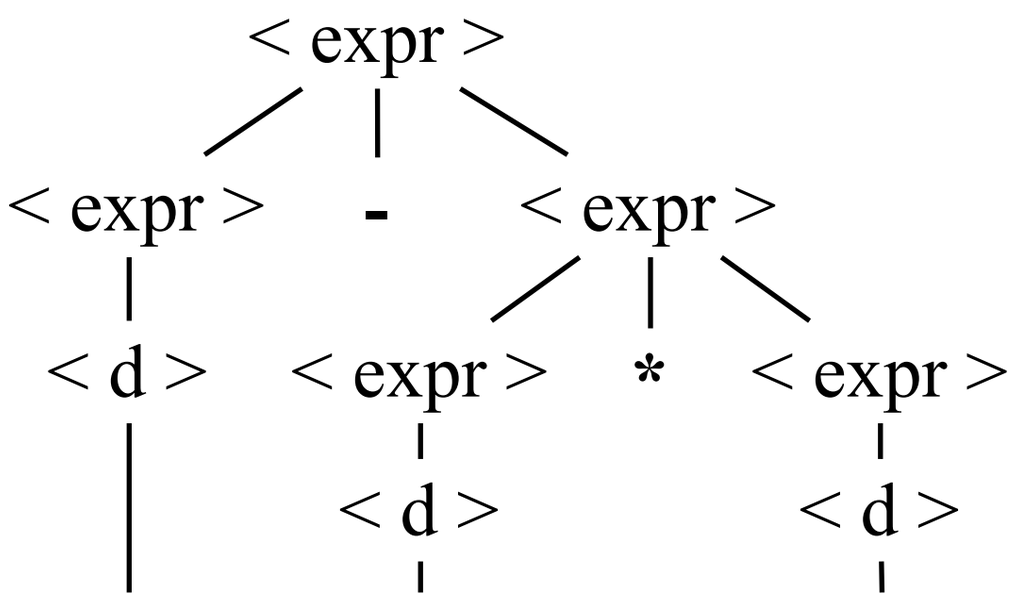




Review: Arithmetic Expression Grammar

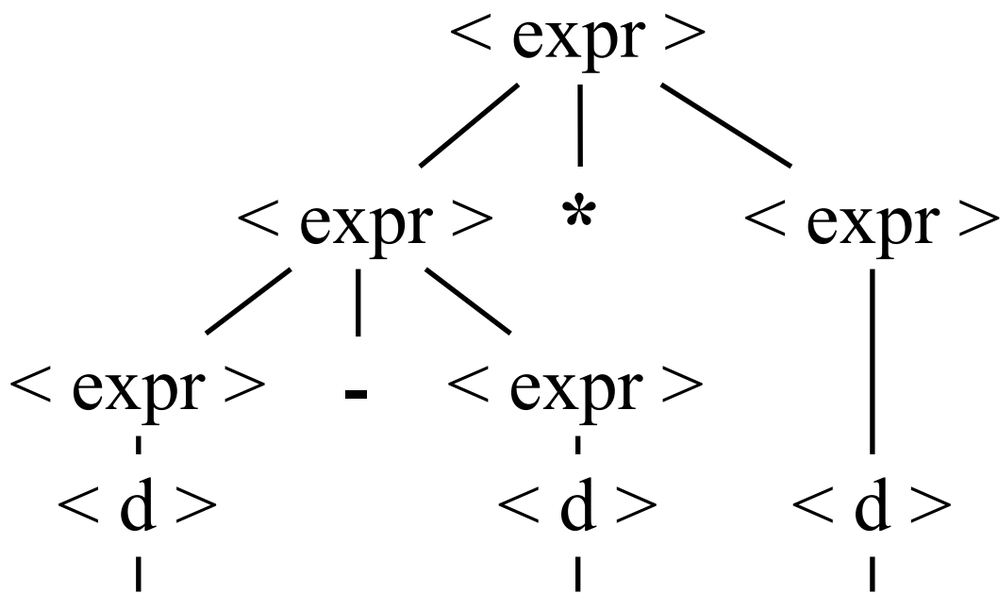
Parse “8 - 3 * 2”:

Two leftmost derivations!



8 3 2

leftmost derivation	
<expr>	⇒L
<expr> - <expr>	⇒L
<d> - <expr>	⇒L
<d> - <expr> * <expr>	⇒L
<d> - <d> * <expr>	⇒L
<d> - <d> * <d>	



8 3 2

leftmost derivation	
<expr>	⇒L
<expr> * <expr>	**⇒L**
<expr> - <expr> * <expr>	**⇒L**
<d> - <expr> * <expr>	⇒L
<d> - <d> * <expr>	⇒L
<d> - <d> * <d>	

Review: Ambiguity

How to deal with ambiguity?

- Change the language
  Example: Adding new terminal symbols as delimiters.
  Fix the *dangling else*, *expression* grammars.
- Change the grammar
  Example: Impose **associativity** and **precedence** in an *arithmetic expression* grammar.

Changing the Grammar to Impose Precedence

< start > ::= < expr >
< expr > ::= < expr > + < expr > |
< expr > - < expr > |
< expr > * < expr > |
< expr > / < expr > |
< d > | < l >
< d > ::= **0** | **1** | **2** | **3** | … | **9**
< l > ::= **a** | **b** | **c** | … | **z**

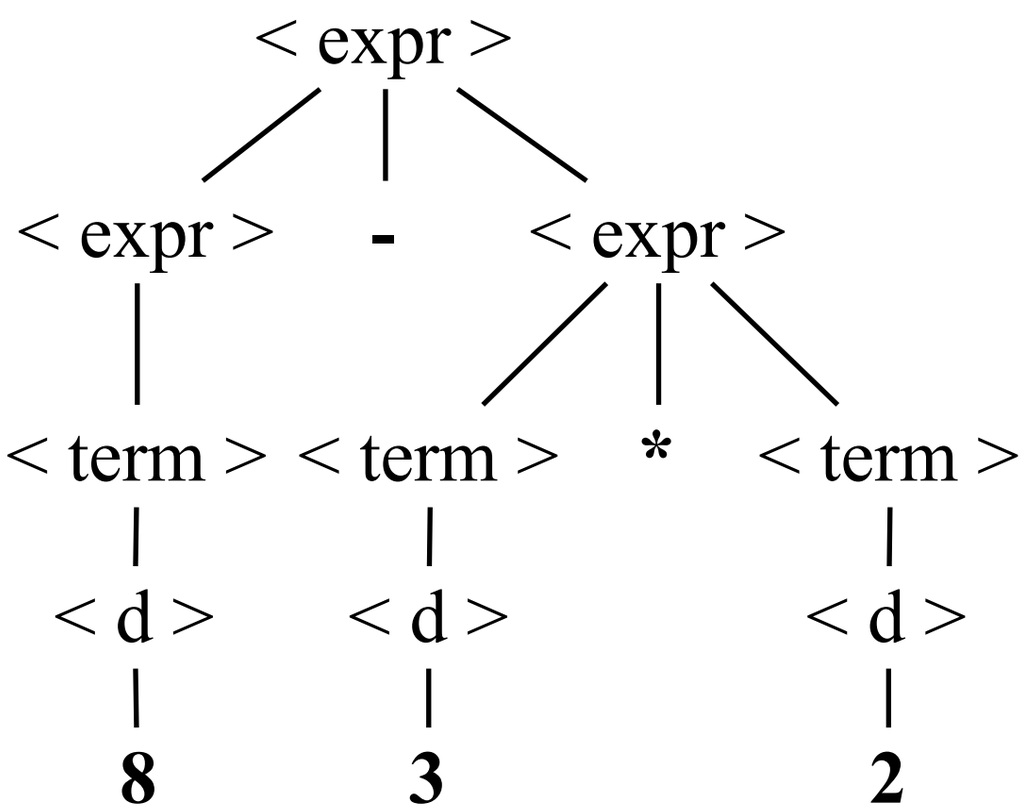
Original Grammar $\mathcal{G}$

< start > ::= < expr >
< expr > ::= < expr > + < expr > |
< expr > - < expr > |
< term >
< term > ::= < term > * < term > |
< term > / < term > |
< d > | < l >
< d > ::= **0** | **1** | **2** | **3** | … | **9**
< l > ::= **a** | **b** | **c** | … | **z**

Modified Grammar $\mathcal{G}$'

Grouping in Parse Tree Now Reflects Precedence

Parse “8 - 3 * 2”:



< start > ::= < expr >

< expr > :: = < expr > + < expr > |

< expr > - < expr > |

< term >

< term > ::= < term > * < term > |

< term > / < term > |

< d > | < l >

< d > :: = **0** | **1** | **2** | **3** | … | **9**

< l > :: = **a** | **b** | **c** | … | **z**

Modified Grammar $\mathcal{G}$’

Only One Possible Parse Tree

Precedence

- *Low Precedence:*
  Addition + and Subtraction -
- *Medium Precedence:*
  Multiplication * and Division /
- *Highest Precedence:*
  Exponentiation ^

```
< start > ::=  < expr >
< expr > :: = < expr > + < expr > |
              < expr > - < expr > |
              < term >
< term > ::=  < term > * < term > |
               < term > / < term > |
                  < d > | < l >
< d >      :: = 0 | 1 | 2 | 3 | … | 9
< l >      :: = a | b | c | … | z
```

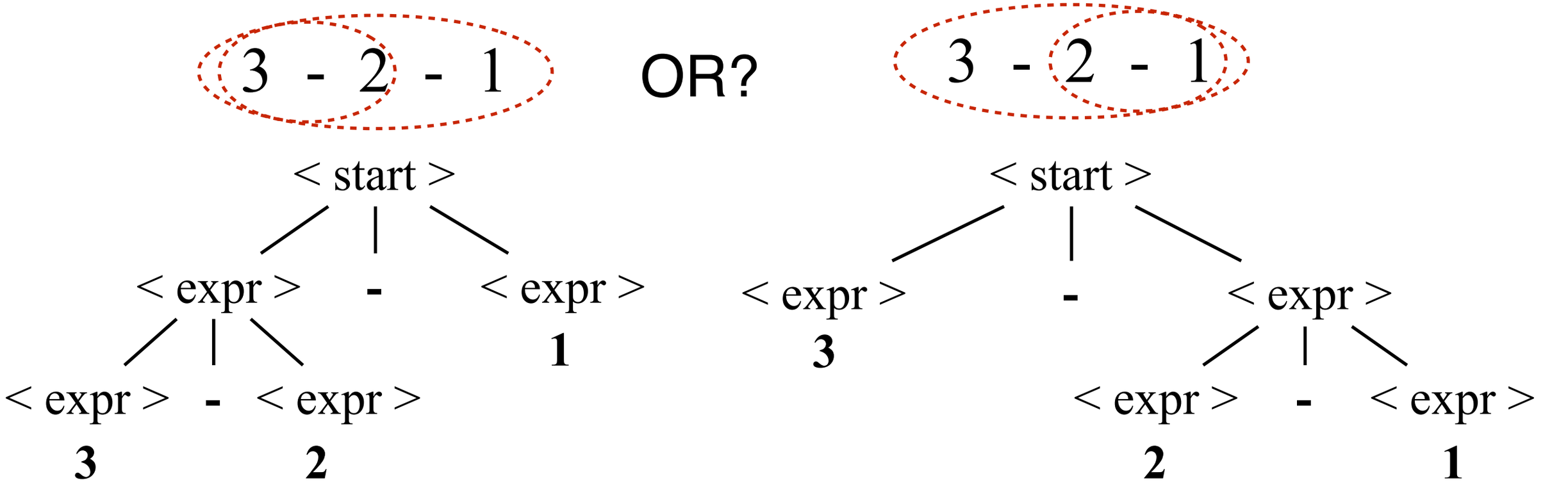
Still Have Ambiguity…

How about 3 - 2 - 1 ?

< start > ::= < expr >
< expr > :: = < expr > + < expr > | < expr > - < expr > | < term >
< term > ::= < term > * < term > | < term > / < term > | < d > | < l >
< d > :: = **0** | **1** | **2** | **3** | … | **9**
< l > :: = **a** | **b** | **c** | … | **z**

Still Have Ambiguity…

How about 3 - 2 - 1 ?



< start > ::= < expr >
< expr > :: = < expr > + < expr > | < expr > - < expr > | < term >
< term > ::= < term > * < term > | < term > / < term > | < d > | < l >
< d > :: = **0** | **1** | **2** | **3** | … | **9**
< l > :: = **a** | **b** | **c** | … | **z**

Still Have Ambiguity…

- Grouping of operators of same precedence not disambiguated.
- Non-commutative operators: only one parse tree correct.

< start > ::= < expr >

< expr > :: = < expr > + < expr > | < expr > - < expr > | < term >

< term > ::= < term > * < term > | < term > / < term > | < d > | < l >

< d > :: = **0** | **1** | **2** | **3** | … | **9**

< l > :: = **a** | **b** | **c** | … | **z**

Imposing Associativity

Same grammar with left / right recursion for - :

Our choices:

< expr > :: = < d > - < expr > |
< d >

< d > :: = **0** | **1** | **2** | **3** | … | **9**

<expr> ⇒
< d > - <expr> ⇒
< d > - < d > - <expr> ⇒
< d > - < d > - < d > - <expr> ⇒
…

Or:

< expr > :: = < expr > - < d > |
< d >

< d > :: = **0** | **1** | **2** | **3** | … | **9**

<expr> ⇒
<expr> - < d > ⇒
<expr> - < d > - < d > ⇒
<expr> - < d > - < d > - <d> ⇒
…

Which one do we want for - in the calculator language?

Associativity

- Deals with operators of same precedence
- Implicit grouping or parenthesizing
- Left to right: *, /, +, -
- Right to left: ^

Complete, Unambiguous Arithmetic Expression Grammar

< start > ::= < expr >

< expr > :: = < expr > + < expr > |
< expr > - < expr > |
< expr > * < expr > |
< expr > / < expr > |
< expr > ^ < expr > |
< d > | < l >

< d > :: = **0** | **1** | **2** | **3** | … | **9**

< l > :: = **a** | **b** | **c** | … | **z**

Original Ambiguous Grammar $\mathcal{G}$

< start > ::= < expr >

< expr > ::= < expr > + < term > |
< expr > − < term > |
< term >

< term > ::= < term > * < factor > |
< term > / < factor > |
< factor >

< factor >::= < g > ^ < factor > |
< g >

< g > ::= (< expr >) | < d > | < l >

< d > ::= **0** | **1** | **2** | … | **9**

< l > ::= **a** | **b** | **c** | … | **z**

Unambiguous Grammar $\mathcal{G}$

Abstract versus Concrete Syntax

Concrete Syntax:

Representation of a construct in a particular language, including placement of keywords and delimiters.

Abstract Syntax:

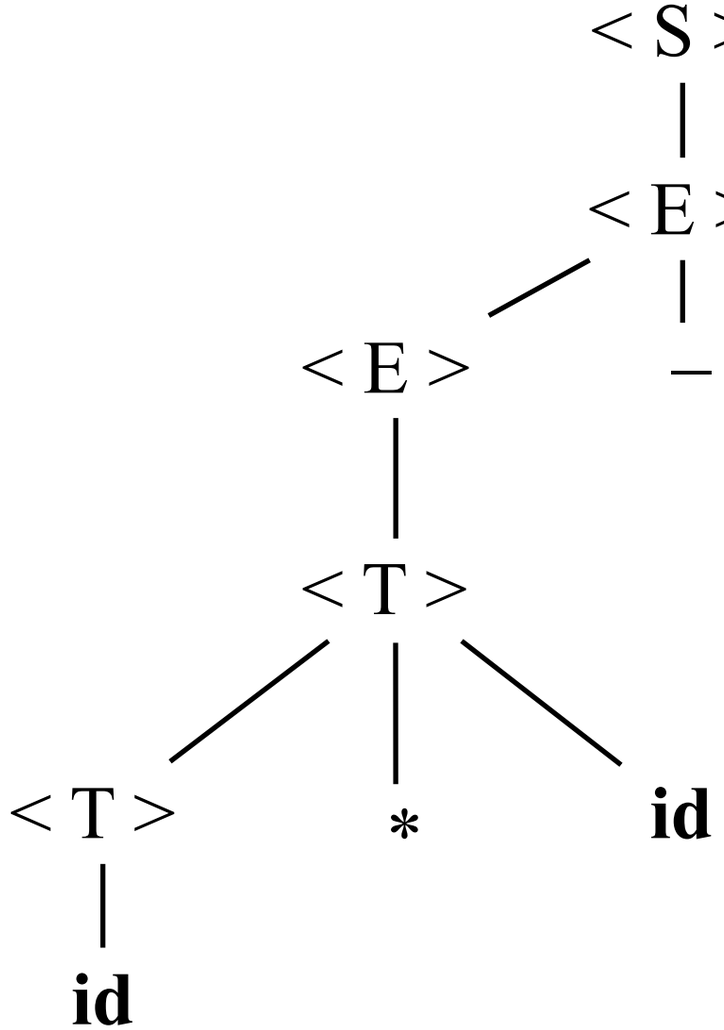
Structure of meaningful components of each language construct.

Example:

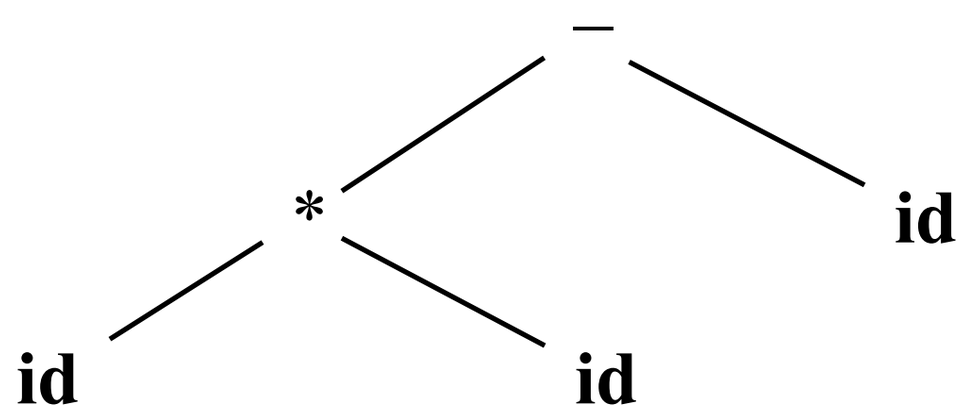
Consider A * B − C:

< S > ::= < E >
< E > ::= < E > − < T > | < T >
< T > ::= < T > * **id** | **id**

Concrete Syntax Tree



Abstract Syntax Tree
(AST)



Abstract versus Concrete Syntax

Same abstract syntax, different concrete syntax:

Pascal

```
while x <> A[i], do
    i := i + 1
end
```

C

```
while ( x != A[i])
    i = i + 1;
```

Regular vs. Context Free

- All Regular languages are context free languages
- Not all context free languages are regular languages

Example:

N ::= X | Y
X ::= a | X b is equivalent to: a b* | c^+
Y ::= c | Y c

Question:

Is $\{a^n b^n \mid n \geq 0\}$ *a context free language?*

Is $\{a^n b^n \mid n \geq 0\}$ *a regular language?*

Regular Grammars

CFGs with restrictions on the shape of production rules.

Left-linear:

X ::= a | X b
N ::= X a b

Right-linear:

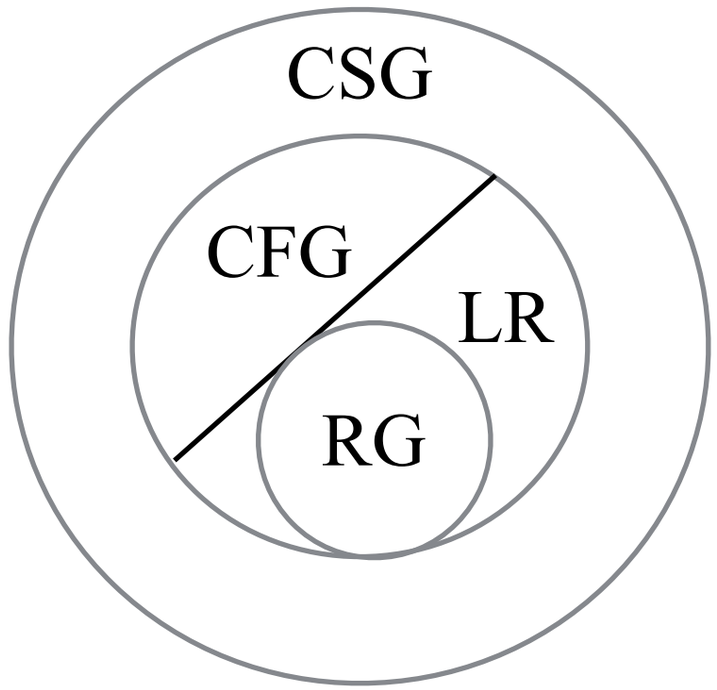
Y ::= a b | a b Y
N ::= b | b Y

Complexity of Parsing

Classification of languages that can be recognized by specific grammars.

Complexity:

Regular grammars	DFAs	**O**(n)
LR grammars	Kunth's algorithm	**O**(n)
Arbitrary CFGs	Earley's algorithm	**O**(n^3)
Arbitrary CSGs	LBAs	P-SPACE COMPLETE

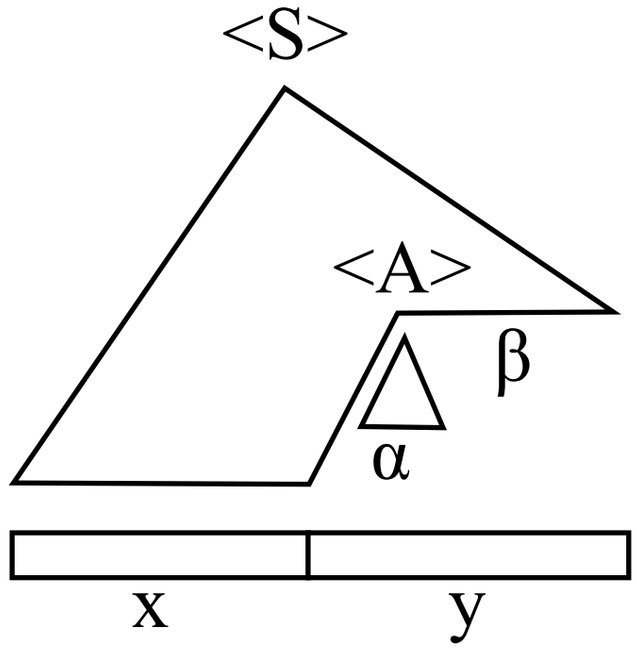


Reading:

Scott Chapter 2.3.4 (for LR parser) and Chapter 2.4.3 for language class.

Earley, Jay (1970), "An efficient context-free parsing algorithm", Communications of the ACM.

Top - Down Parsing - LL(1)



Basic Idea:

- The parse tree is constructed from the root, expanding non-terminal nodes on the tree's frontier following a **leftmost** derivation.
- The input program is read from **left** to right, and input tokens are read (consumed) as the program is parsed.
- The next non-terminal symbol is replaced using one of its rules. The particular choice has to be unique and uses parts of the input (partially parsed program), for instance the first **token** of the remaining input.

Top - Down Parsing - LL(1) (cont.)

Example:

S ::= a S b | ε

How can we parse (automatically construct a leftmost derivation) the input string **a a a b b b** using a PDA (push-down automaton) and only the first symbol of the remaining input?

INPUT: a a a b b b eof

Predictive Parsing

Basic idea:

For any two productions A ::= α | β, we would like a distinct way of choosing the correct production to expand.

For some rhs α ∈ G, define **FIRST**(α) as the set of tokens that appear as the first symbol in some string derived from α.

That is

x ∈ FIRST(α) iff α ⇒* **x**γ for some γ, and

ε ∈ FIRST(α) iff α ⇒* ε

Predictive Parsing

For a non-terminal A, define **FOLLOW**(A) as the set of terminals that can appear immediately to the right of A in some sentential form.

Thus, a non-terminal's **FOLLOW** set specifies the tokens that can legally appear after it. A terminal symbol has no **FOLLOW** set.

FIRST and FOLLOW sets can be constructed automatically

Predictive Parsing (cont.)

Key Property:

Whenever two productions A ::= α and A ::= β both appear in the grammar, we would like

- $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$, and
- if $\alpha \Rightarrow^* \varepsilon$, then $FIRST(\beta) \cap FOLLOW(A) = \varnothing$
- Analogue case for $\beta \Rightarrow^* \varepsilon$. Note: due to first condition, at most one of α and β can derive ε.

This would allow the parser to make a correct choice with a lookahead of only one symbol!

LL(1) Grammar

Define $FIRST^{+}(\delta)$ for rule A ::= δ

- $FIRST(\delta)$ - { ε } U Follow (A), if $\varepsilon \in FIRST(\delta)$
- $FIRST(\delta)$ otherwise

A Grammar is LL(1) iff

(A ::= α and A ::= β) implies

$$FIRST^{+}(\alpha) \cap FIRST^{+}(\beta) = \varnothing$$

Next Lecture

Next Time:

- Review and more on LL(1) parsing
- Read Scott, Chapter 2.3.1 - 2.3.2 (and some chapter on companion site)