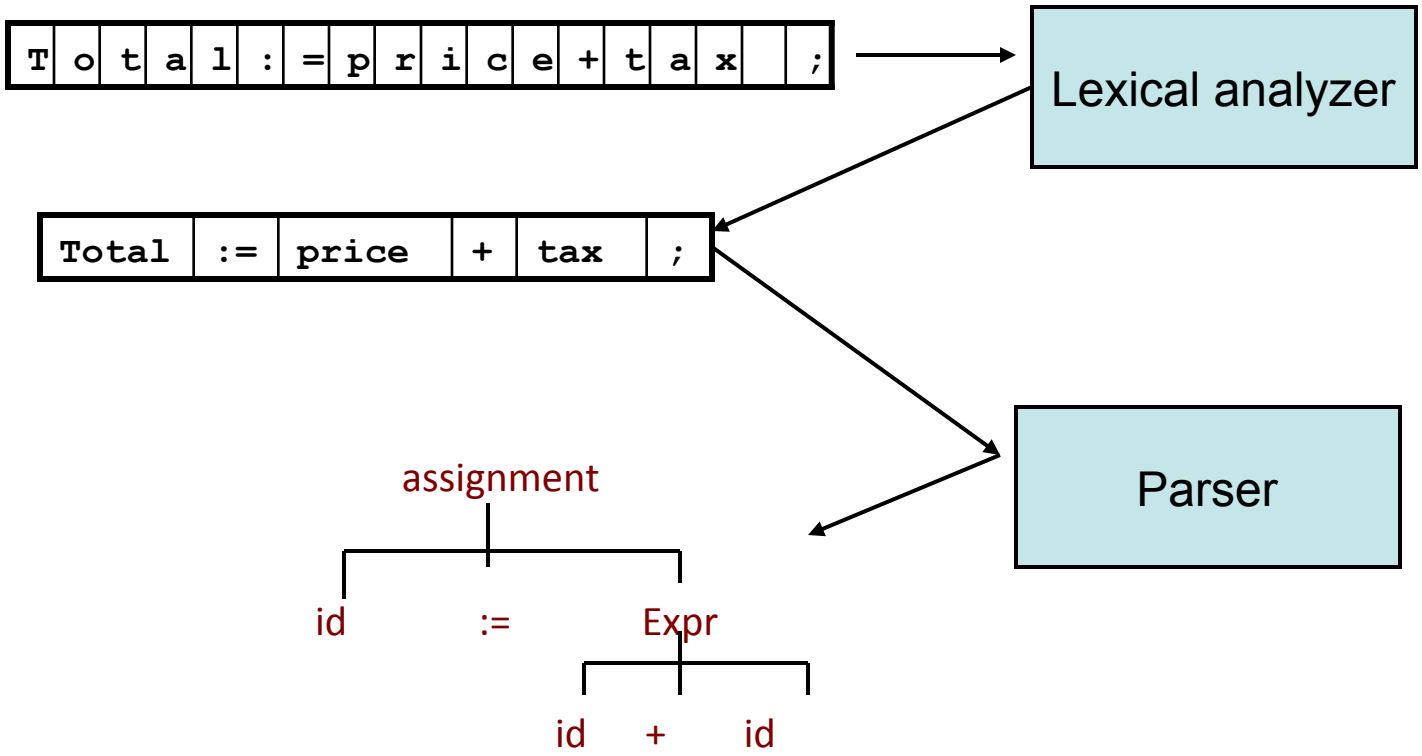
03-60-214 Syntax analysis

Jianguo Lu
School of Computer Science
University of Windsor


T o t a l : = p r i c e + t a x ;
Lexical analyzer
Total := price + tax ;
Parser
assignment
id := Expr
id + id

Grammars

Formal definition of language

- A language is a set of strings
  - English language

    {“the brown dog likes a good car”, … …}

    {sentence | sentence written in English}
  - Java language {program |program written in Java}
  - HTML language {document |document written in HTML}
- How do you define a language?
- It is unlikely that you can enumerate all the sentences, programs, or documents

How to define a language

- How to define English
  - A set of words, such as brown, dog, like
  - A set of rules
    - A sentence consists of a subject, a verb, and an object;
    - The subject consists of an optional article, followed by an optional adjective, and followed by a noun;
    - … …
  - More formally:
    - Words ={a, the, brown, friendly, good, book, refrigerator, dog, car, sings, eats, likes}
    - Rules:
      1) SENTENCE → SUBJECT VERB OBJECT
      2) SUBJECT → ARTICLE ADJECTIVE NOUN
      3) OBEJCT → ARTICLE ADJECTIVE NOUN
      4) ARTICLE → a | the| EMPTY
      5) ADJECTIVE → brown | friendly | good | EMPTY
      6) NOUN → book| refrigerator | dog| car
      7) VERB → sings | eats | likes

Derivation of a sentence

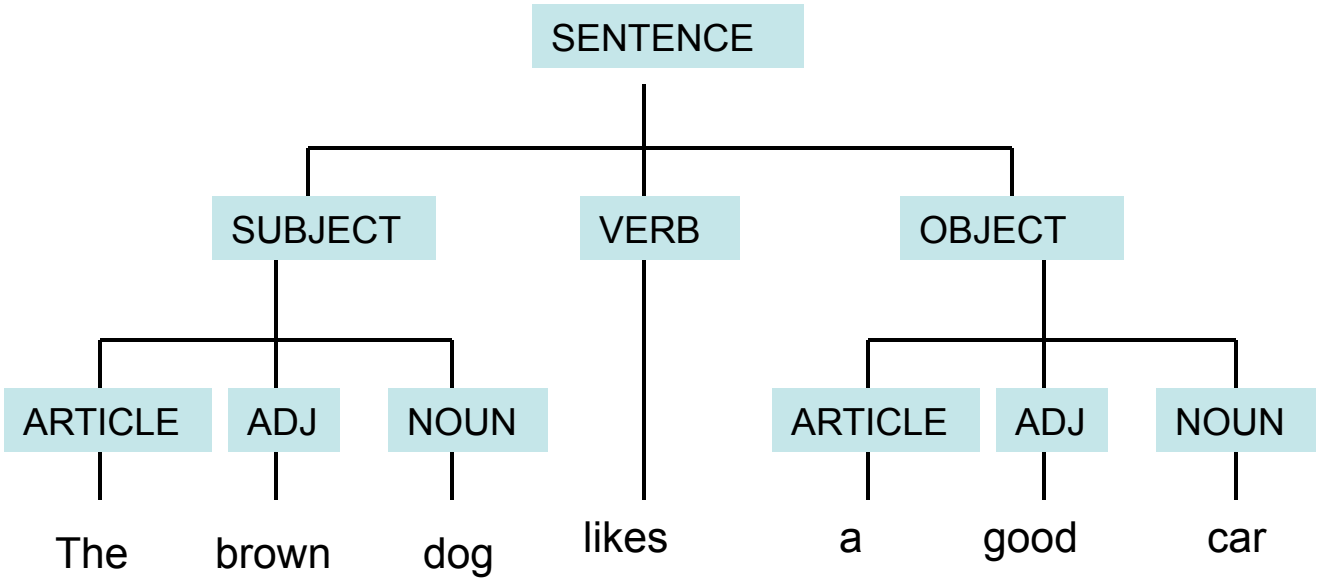
- Derivation of a sentence *"the brown dog likes a good car"*

  SENTENCE
  → SUBJECT VERB OBJECT
  → ARTICLE ADJECTIVE NOUN VERB OBJECT
  → the brown dog VERB OBJECT
  → the brown dog likes ARTICLE ADJECTIVE NOUN
  → the brown dog likes a good car

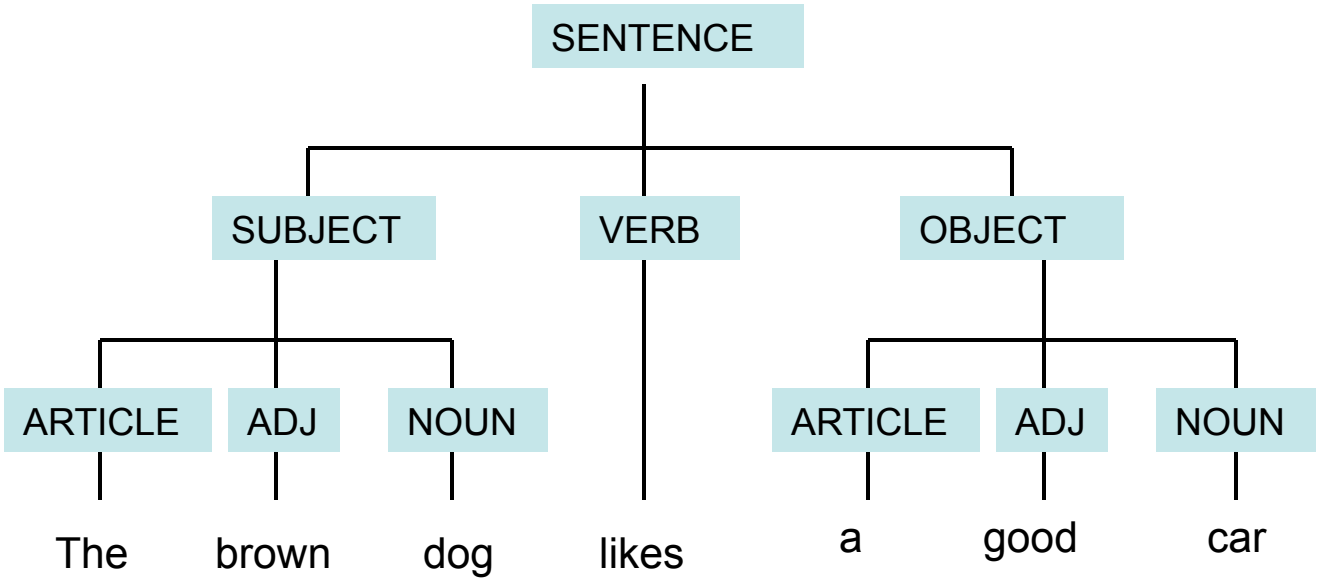
- Rules:
  1) SENTENCE → SUBJECT VERB OBJECT
  2) SUBJECT → ARTICLE ADJECTIVE NOUN
  3) OBEJCT → ARTICLE ADJECTIVE NOUN
  4) ARTICLE → a | the| EMPTY
  5) ADJECTIVE → brown | friendly | good | EMPTY
  6) NOUN → book| refrigerator | dog| car
  7) VERB → sings | eats | likes

The parse tree of the sentence

Parse the sentence: “the brown dog likes a good car”
The top-down approach



Top down and bottom up parsing



Types of parsers

- Top down
  - Repeatedly rewrite the start symbol
  - Find the left-most derivation of the input string
  - Easy to implement
- Bottom up
  - Start with the tokens and combine them to form interior nodes of the parse tree
  - Find a right-most derivation of the input string
  - Accept when the start symbol is reached
- Bottom up is more prevalent

Formal definition of grammar

- A grammar is a 4-tuple G = (Σ, N, P, S)
  - Σ is a finite set of terminal symbols;
  - N is a finite set of nonterminal symbols;
  - P is a set of productions;
  - S (from N) is the start symbol.
- The English sentence example
  - Σ ={a, the, brown, friendly, good, book, refrigerator, dog, car, sings, eats, likes}
  - N={SENTENCE, SUBJECT, VERB, NOUN, OBJECT, ADJECTIVE, ARTICLE}
  - S={SENTENCE}
  - P={rule 1) to rule 7) }

Recursive definition

- Number of sentence can be generated:

ARTICLE	ADJ	NOUN	VERB	ARTICLE	ADJ	NOUN	sentences
3 *	4*	4*	3*	3*	4*	4*	= 6912

- How can we define an infinite language with a finite set of words and finite set of rules?
- Using recursive rules:
  - SUBJECT/OBJECT can have more than one adjectives:
    1) SUBJECT → ARTICLE ADJECTIVES NOUN
    2) OBEJCT → ARTICLE ADJECTIVES NOUN
    3) ADJECTIVES→ ADJECTIVE | ADJECTIVES ADJETIVE
  - Example sentence:

    "the good brown dog likes a good friendly book"

Chomsky hierarchy

- Noam Chomsky hierarchy is based on the form of production rules
- General form

  $\alpha_1\ \alpha_2\ \alpha_3\ \ldots\alpha_n \rightarrow \beta_1\ \beta_2\ \beta_3\ \ldots\ \beta_m$

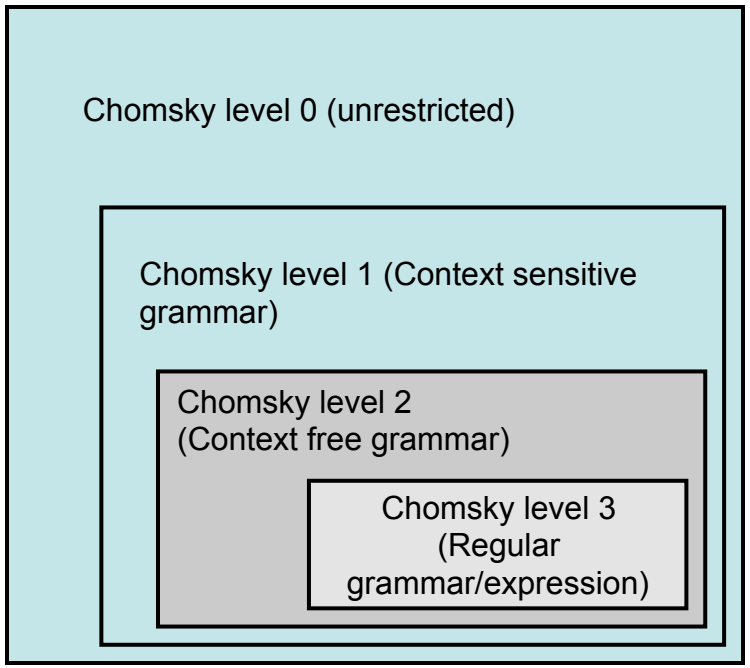
  Where α and β are from terminals and non terminals, or empty.
- Level 3: Regular grammar
  - Of the form $\alpha \rightarrow \beta$ or $\alpha \rightarrow \beta_1\ \beta_2$
  - n=1, and α is a non terminal.
  - β is either a terminal or a terminal followed by a nonterminal
  - RHS contains at most one non-terminal at the right end.
- Level 2: Context free grammar
  - Of the form $\alpha \rightarrow \beta_1\ \beta_2\ \beta_3 \ldots\ \beta_m$
  - α is non terminal.
- Level 1: Context sensitive grammar
  - n<m. The number of symbols on the lhs must not exceed the number of symbols on the rhs
- Level 0: unrestricted grammar

Context sensitive grammar

- Called context sensitive because you can construct the grammar of the form
  - $A\alpha B \rightarrow A\ \beta\ B$
  - $A\alpha C \rightarrow A\ \gamma\ B$
- The substitution of α depending on the surrounding context A and B or A and C.

Chomsky hierarchy

- regular ⊆ Context-free ⊆ Context-sensitive ⊆ unrestricted



- The more powerful the grammar, the more complex the program required to recognize the legal inputs.

Grammar examples

- Regular grammar
  - L={w|w consists of arbitrary number of ‘a’s and ‘b’s}
  - Grammar: S→a | b | a S| bS
  - Example sentence: “abb” is a legal sentence in this language
  - Derivation:

    S → a S

    → a b S

    → a b b

Context free grammar example

- Context free grammar example
  - Language $L=\{a^n b^n\}$
  - Grammar: S→ab | aSb
  - Notice the difference with regular grammar.
  - Example sentence: “aaabbb” is a legal sentence in this language
  - Derivation:

    S → a S b

    → a a S bb

    → a a a b b b

Characterize different types of grammars

- Regular grammar
  - Being able to count one item
- Context free grammar
  - being able to count pairs of items
  - $a^n b^n$
- Context sensitive grammar
  - Being able to count arbitrarily;
  - $a^n b^n c^n$

Implications of different grammars in applications

- Regular grammar
  - Recognize words
- Context free grammar
  - Recognize pairs of parenthesis

    ((a+b) *c)/2
  - Recognize blocks

```
{   statement1;
    { statement2;
      statement3;
    }
}
```

- Context sensitive grammar

Context free grammars and languages

- Many languages are not regular. Thus we need to consider larger classes of languages;
- Context Free Language (CFL) played a central role in natural languages since 1950's (Chomsky) and in compilers since 1960's (Backus);
- Context Free Grammar (CFG) is the basis of BNF syntax;
- CFG is increasingly important for XML and DTD (XML Schema).

Informal Example of CFG

- Palindrome:
  - Madam, I'm adam.
  - A man, a plan, a canal, panama!
- Consider L_{pal} ={w|w is a palindrome on symbols 0 and 1}
- Example: 1001, 11 are palindromes
- How to represent palindrome of any length?
- Basis: ε, 0 and 1 are palindromes;
  1. P→ε
  2. P→0
  3. P→1
- Induction: if *w* is palindrome, so are *0w0* and *1w1*. nothing else is a palindrome.
  1. P→0P0
  2. P→1P1

The informal example (cont.)

- CFG is a formal mechanism for definitions such as the one for L_{pal}
  1. $P \rightarrow \varepsilon$
  2. $P \rightarrow 0$
  3. $P \rightarrow 1$
  4. $P \rightarrow 0P0$
  5. $P \rightarrow 1P1$
- 0 and 1 are terminals
- P is a variable (or nonterminal, or syntactic category)
- P is also the start symbol.
- 1-5 are productions (or rules)

Formal definition of CFG

- A CFG is a 4-tuple G=(Σ,N,P,S) where
  1. Σ is a finite set of terminals;
  2. N is finite set of variables (non-terminals);
  3. P is a finite set of productions of the form A→α, where A is a nonterminal and α consists of symbols from Σ and N;
     1. A is called the head of the production;
     2. α is called the body of the production;
  4. S is a designated non-terminal called the start symbol.
- Example:
  - G_{pal}=({0,1}, {P}, A,P), where A ={P→ε, P→0, P→1, P→0P0, P→1P1}
- Sometimes we group productions with the same head.
  - e.g., A={P→ε|0|1|0P0|1P1}.

Another CFG example: arithmetic expression

- G=({+,*,(,),a,b,0,1}, {E, R, S}, P, E)
  1. E→R
  2. E→E+E
  3. E→E*E
  4. E→(E)

  5. R→aS
  6. R→bS
  7. S→ ε
  8. S→aS
  9. S→bS
  10. S→0S
  11. S→1S

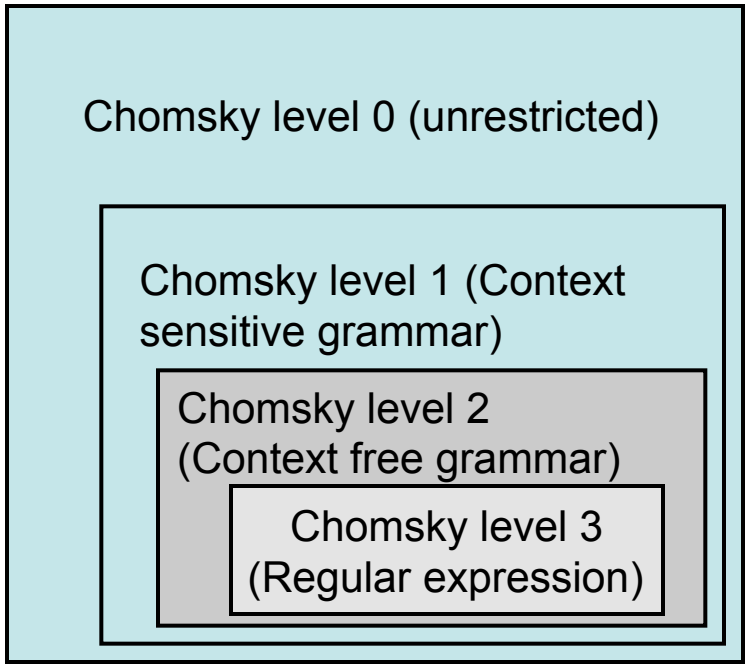
Two level descriptions

- Context-free syntax of arithmetic expressions
  1. E→R
  2. E→E+E
  3. E→E*E
  4. E→(E)
- Lexical syntax of arithmetic expressions
  1. R→aS
  2. R→bS
  3. S→ ε
  4. S→aS
  5. S→bS
  6. S→0S
  7. S→1S

  R→(a|b)(a|b|0|1)*
- Why two levels
  - We think that way (sentence, word, character).
  - besides, … (see next slide)

Why use regular expression

- Every regular set is a context free language.
- Since we can use CFG to describe regular language, why using regular expression to define lexical syntax of a language? Why not using CFG to describe everything?
  - Lexical rules are simpler;
  - Regular expressions are concise and easier to understand;
  - More efficient lexical analyzers can be constructed from regular expression;
  - It is good to modularize compiler into two separate components.



BNF and EBNF

- BNF and EBNF are commonly accepted ways to express productions of a context-free grammar.
- BNF
  - Introduced by John Backus, first used to describe Algol 60. John won Turing award in 1977.
  - "Backus Normal Form", or "Backus Naur Form".
  - EBNF stands for Extended BNF.
- BNF format
  - lhs ::= rhs
  - Quote terminals or non-terminals
    - <> to delimit non-terminals,
    - bold or quotes for terminals, or 'as is'
  - vertical bar for alternatives
  - There are many different notations, here is an example

opt-stats ::= stats-list | EMPTY .

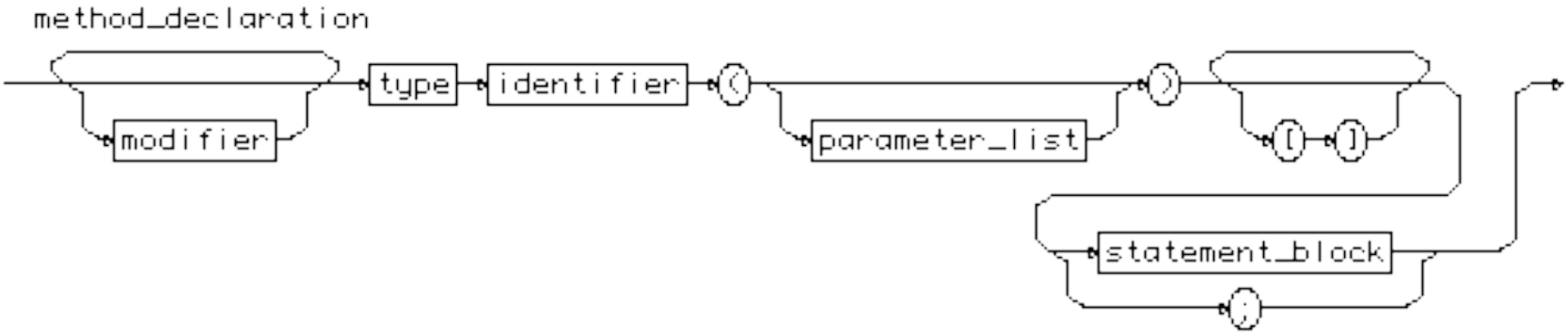
stats-list ::= statement | statement ';' stats-list .

EBNF

- An extension of BNF, use regular- expression-like constructs on the right-hand-side of the rules:
  - write *[A]* to say that *A* is optional
  - Write *{A}* or *A** to denote 0 or more repetitions of *A* (ie. the Kleene closure of *A*).
- Using EBNF has the advantage of simplifying the grammar by reducing the need to use recursive rules.
- Both BNF and EBNF are equivalent to CFG
- Example:

BNF	EBNF
block ::= '{' opt-stats '}' opt-stats ::= stats-list | EMPTY stats-list ::= statement | statement ';' stats-list	block ::= '{' [stats-list] '}' stats-list ::= statement (';' statement)*

- **Java method_declaration**
- method_declaration ::= { modifier } type identifier
  "(" [parameter_list] ")" { "[" "]" } (statement_block | ";")



Languages, Grammars, and automata

Language Class	Grammar	Automaton
3	Regular	NFA or DFA
2	Context-Free	Push-Down Automaton
1	Context-Sensitive	Linear-Bounded Automaton
0	Unrestricted (or $Free$)	Turing Machine

Closer to machine

More expressive

Another arithmetic expression example

(p1) exp→exp+digit
(p2) exp→exp-digit
(p3) exp→digit
(p4) digit→0|1|2|3|4|5|6|7|8|9

- the "|" means OR
- So the rules can be simplified as:
  (P1-3) Exp → exp + digit | exp - digit | digit
  (p4) digit→0|1|2|3|4|5|6|7|8|9

Derivation

- One step derivation ($\Rightarrow$ relation)
  - If A$\rightarrow \gamma$ is a production, α A β is a string of terminals and variables, then

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

  - Example:

    9 - digit + *digit* $\Rightarrow$ 9 - digit + 2

    using rule P4 : *digit* $\rightarrow$ 2

- Zero or more steps of derivation ($\Rightarrow^*$)
  - Basis: $\alpha \Rightarrow^* \alpha$
  - Induction: if $\alpha \Rightarrow^* \beta$, $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$.
  - Example:
    - 9 - digit + digit $\Rightarrow^*$ 9 - digit + digit
    - 9 - digit + digit $\Rightarrow^*$ 9 - 5 + *2*
    - 9 - digit + digit $\Rightarrow$ 9 - 5 + digit $\Rightarrow$ 9 - 5 + 2
- One or more steps of derivation ($\Rightarrow^+$)
  - Example: 9 - digit + digit $\Rightarrow^+$ 9 - 5 + *2*

Example of derivation

- We can derive the string 9 - 5 + 2 as follows:

$exp \Rightarrow exp + digit$ (P1 : $exp \rightarrow exp + digit$)

$\Rightarrow exp - digit + digit$ (P2 : $exp \rightarrow exp - digit$)

$\Rightarrow digit - digit + digit$ (P3 : $exp \rightarrow digit$)

$\Rightarrow 9 - digit + digit$ (P4 : $digit \rightarrow 9$)

$\Rightarrow 9 - 5 + digit$ (P4 : $digit \rightarrow 5$)

$\Rightarrow 9 - 5 + 2$ (P4 : $digit \rightarrow 2$)

$exp \Rightarrow^{+} 9 - 5 + 2$

$exp \Rightarrow^{*} 9 - 5 + 2$

Left most and right most derivations

Left most derivation:

$exp \Rightarrow_{lm} exp + digit$

$\Rightarrow_{lm} exp - digit + digit$

$\Rightarrow_{lm} digit - digit + digit$

$\Rightarrow_{lm} 9 - digit + digit$

$\Rightarrow_{lm} 9 - 5 + digit$

$\Rightarrow_{lm} 9 - 5 + 2$

$exp \Rightarrow^{+}_{lm} 9 - 5 + 2$

$exp \Rightarrow^{*}_{lm} 9 - 5 + 2$

Right most derivation:

$exp \Rightarrow_{rm} exp + digit$

$\Rightarrow_{rm} exp + 2$

$\Rightarrow_{rm} exp - digit + 2$

$\Rightarrow_{rm} exp - 5 + 2$

$\Rightarrow_{rm} digit - 5 + 2$

$\Rightarrow_{rm} 9 - 5 + 2$

$exp \Rightarrow^{+}_{rm} 9 - 5 + 2$

$exp \Rightarrow^{*}_{rm} 9 - 5 + 2$

(p1) exp→exp+digit

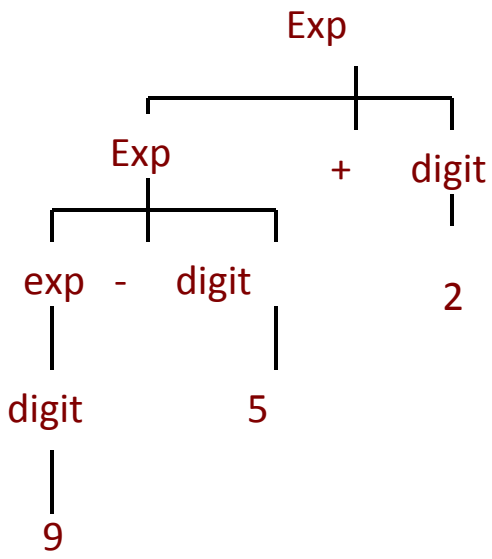
(p2) exp→exp-digit

(p3) exp→digit

(p4) digit→0|1|2|3|4|5|6|7|8|9

Parse tree

- This derivation can also be represented via a *Parse Tree.*



(p1) exp→exp+digit

(p2) exp→exp-digit

(p3) exp→digit

(p4) digit→0|1|2|3|4|5|6|7|8|9

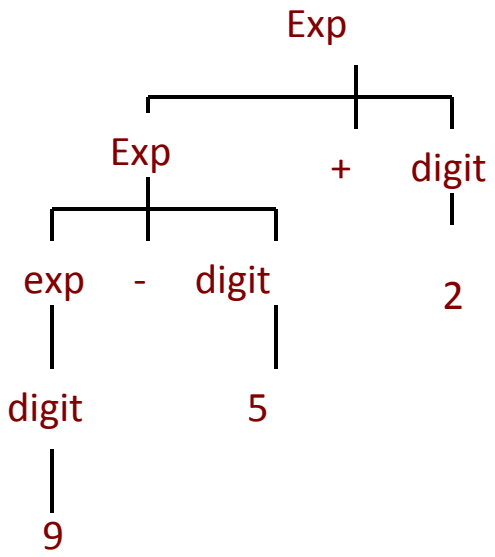
9 - 5 + 2

Tree terminologies

- Trees are collections of nodes, with a parent-child relationship.
  - A node has at most one parent, drawn above the node;
  - A node has zero or more children, drawn below the node.
- There is one node, the *root*, that has no parent. This node appears at the top of the tree
- Nodes with no children are called *leaves*.
- Nodes that are not leaves are *interior nodes*.

Formal definition of parse tree

- Parse tree shows the derivation of a string using a grammar.
- Properties of a parse tree:
  - The root is labeled by the start symbol;
  - Each leaf is labeled by a terminal or ε;
  - Each interior node is labeled by a nonterminal;
  - If A is the nonterminal node and X1, …, Xn are the children nodes of A, then A→ X1 … Xn is a production.
- Yield of a parse tree
  - look at the leaves of a parse tree, from left to right, and concatenate them
  - Example: 9-5+2



The language of a grammar

- If G is a grammar, the language of the grammar, denoted as L(G), is the set of terminal strings that have derivations from the start symbol.
- If a language L is the language of some context free grammar, then L is said to be a context free language.
- Example: the set of palindromes is a context free language.

Derivation and the parse tree

- The followings are equivalent:
  - $A \Rightarrow^* w$;
  - $A \Rightarrow^*_{lm} w$;
  - $A \Rightarrow^*_{rm} w$;
  - There is a parse tree with root A and yield w.

Applications of CFG

- Parser and parser generator;
- Markup languages.

Ambiguity of Grammar

- What is ambiguous grammar;
- How to remove ambiguity;

Example of ambiguous grammar

- Ambiguous sentence:
  - Fruit flies like a banana
- Consider a slightly modified grammar for expressions

  expr → expr + expr | expr * expr | digit
- Derivations of 9+5*2

  expr

  $\Rightarrow$ expr + expr

  $\Rightarrow$ expr + expr * expr

  $\Rightarrow_{+}$ 9+5*2

  expr

  $\Rightarrow$ expr*expr

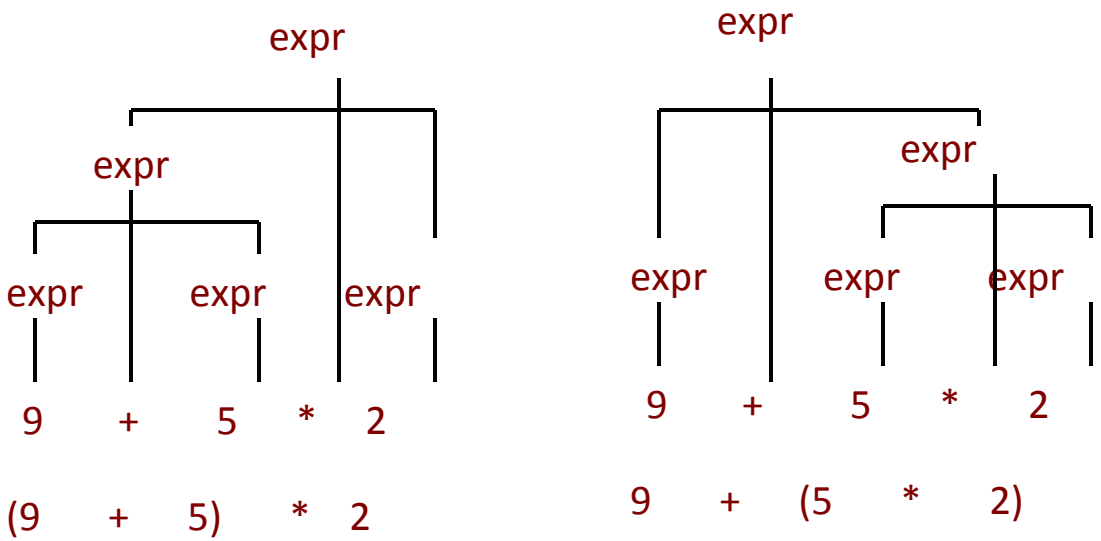
  $\Rightarrow$ expr+expr*expr

  $\Rightarrow_{+}$ 9+5*2
- There are different derivations

Ambiguity of a grammar

- Ambiguous grammar: produce more than one parse tree

  expr → expr + expr | exp * expr | digit



- Problems of ambiguous grammar
  - one sentence has different interpretations

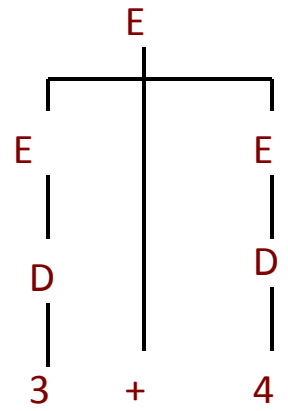
Several derivations can not decide whether the grammar is ambiguous

- Using the expr grammar, 3+4 has many derivations:

E
⇒ E+E
⇒ D+E
⇒ 3+E
⇒ 3+D
⇒ 3+4

E
⇒E+E
⇒E+D
⇒E+4
⇒D+4
⇒3+4

- Based on the existence of two derivations, we can not deduce that the grammar is ambiguous;
- It is not the multiplicity of derivations that causes ambiguity;
- It is the existence of more than one parse tree.
- In this example, the two derivations will produce the same tree



Derivation and parse tree

- In an unambiguous grammar, leftmost derivation will be unique; and rightmost derivation will be unique;
- How about ambiguous grammar?

$$
\begin{aligned}
&E \\
&\Rightarrow_{lm} E+E \\
&\Rightarrow_{lm} D+E \\
&\Rightarrow_{lm} 9+E \\
&\Rightarrow_{lm} 9+E*E \\
&\Rightarrow_{lm} 9+D*E \\
&\Rightarrow_{lm} 9+5*E \\
&\Rightarrow_{lm} 9+5*D \\
&\Rightarrow_{lm} 9+5*2
\end{aligned}
$$

$$
\begin{aligned}
&E \\
&\Rightarrow_{lm} E*E \\
&\Rightarrow_{lm} E+E*E \\
&\Rightarrow_{lm} D+E*E \\
&\Rightarrow_{lm} 9+E*E \\
&\Rightarrow_{lm} 9+D*E \\
&\Rightarrow_{lm} 9+5*E \\
&\Rightarrow_{lm} 9+5*D \\
&\Rightarrow_{lm} 9+5*2
\end{aligned}
$$

- A string has two parser trees iff it has two distinct leftmost derivations (or two rightmost derivations).

Remove ambiguity

- Some theoretical results (bad news)
  - Is there an algorithm to remove the ambiguity in CFG?
    - the answer is no
  - Is there an algorithm to tell us whether a CFG is ambiguous?
    - The answer is also no.
  - There are CFLs that have nothing but ambiguous CFGs.
    - That kind of language is called ambiguous language;
    - If a language has one unambiguous grammar, then it is called unambiguous language.
- In practice, there are well-known techniques to remove ambiguity
- Two causes of the ambiguity in the expr grammar
  - the precedence of operator is not respected. "*" should be grouped before "+";
  - a sequence of identical operator can be grouped either from left or from right. 3+4+5 can be grouped either as (3+4)+5 or 3+(4+5).

Remove ambiguity

- Enforcing precedence by introducing several different variables, each represents those expressions that share a level of binding strength
  - factor: digit is a factor
  - term: factor * factor*factor .... is a term
  - expression: term+term+term ... is an expression
- So we have a new grammar:

  E→T | E+T

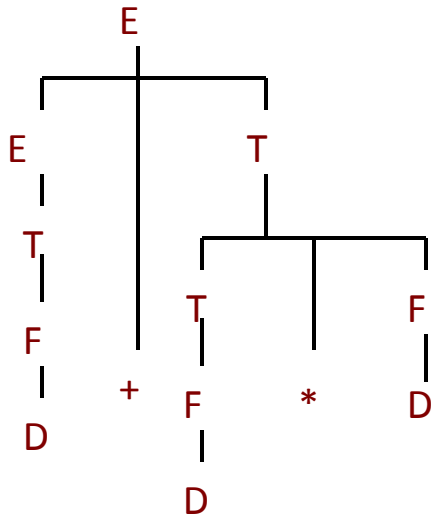
  T→F | T*F

  F→D
- Compare the original grammar:

  E→E+E

  E→E*E

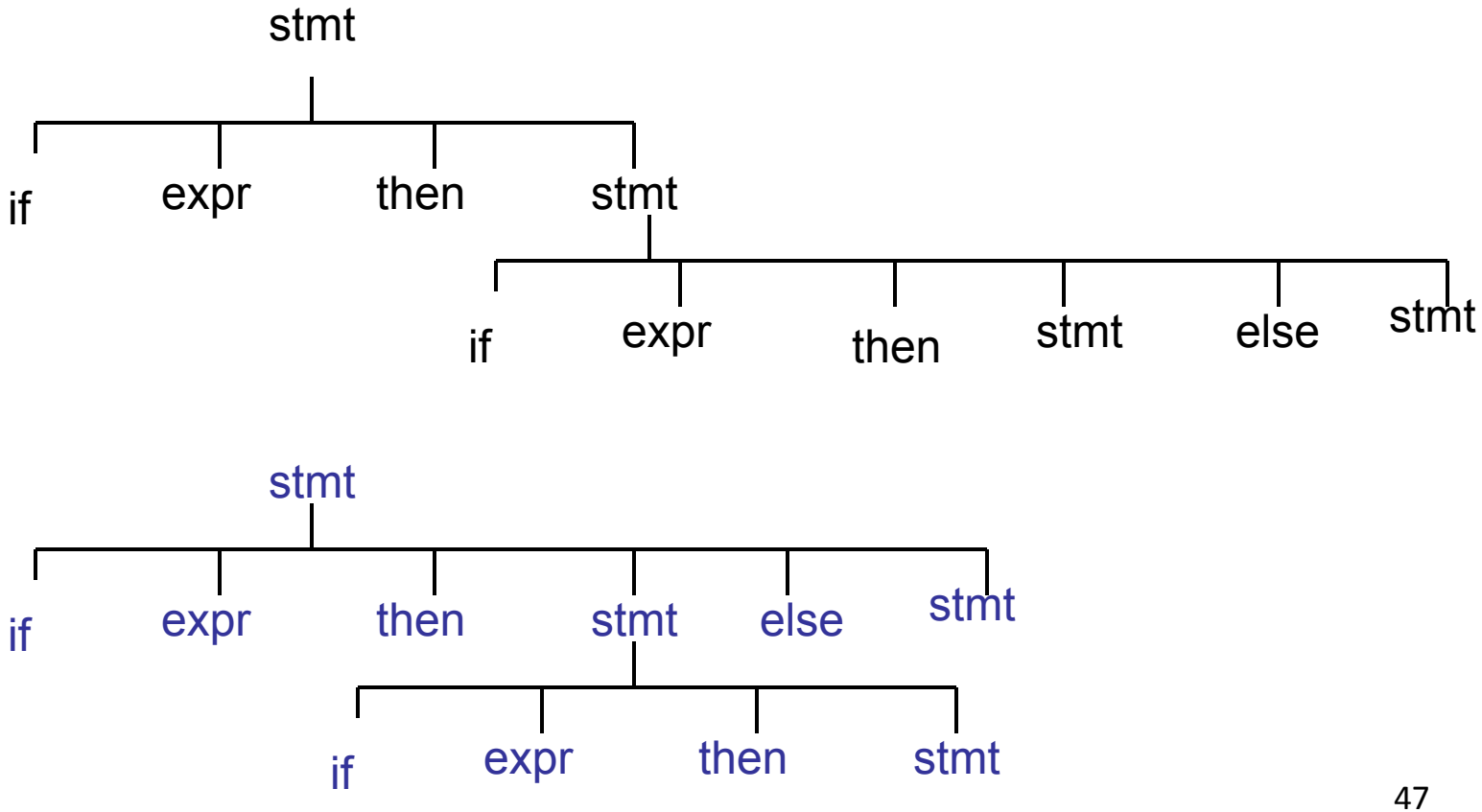
  E→D
- The parser tree for D+D*D is:



Another ambiguous grammar

Stmt → if expr then stmt
| if expr then stmt else stmt
| other

If E1 then if E2 then S1 else S2



Remove the ambiguity

- Match “else” with closest previous unmatched “then”
- How to incorporate this rule into the grammar?

Stmt → if expr then stmt
| if expr then stmt else stmt
| other

stmt→matched_stmt | unmatched_stmt
matched_stmt→if expr then matched_stmt else matched_stmt
|other
unmatched_stmt→if expr then stmt
| if expr then matched_stmt else unmatched_stmt

The parse tree of if-stmt example