CONTEXT-FREE GRAMMAR (CFG)

Dr. Nadeem Akhtar

Assistant Professor
Department of Computer Science & IT
The Islamia University of Bahawalpur

PhD – Formal methods in Software engineering
IRISA – University of South Brittany – FRANCE.

CONTEXT FREE GRAMMAR (CFG)

- Context-Free Grammar is a more powerful method of describing languages.

CFGs have a recursive structure, which makes them useful in a variety of applications.

- ***Study of human languages***. One way of understanding the relationship of terms such as noun, verb, and preposition and their respective phrases leads to a natural recursion because noun phrases may appear inside verb phrases and vice versa. Context-free grammars can capture important aspects of these relationships.

CONTEXT FREE GRAMMAR (CFG)

- An important application of context-free grammars occurs in the ***specification and compilation of programming languages***.

A grammar for a programming language is a reference for people trying to learn the language syntax.

- Designers of compilers and interpreters for programming languages start by obtaining a grammar for the language.

CONTEXT FREE GRAMMAR (CFG)

- ***Parser*** of a compiler or interpreter extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution.

- A number of methodologies facilitate the construction of a parser once a context-free grammar is available. Some tools even automatically generate the parser from the grammar.

CONTEXT FREE GRAMMAR (CFG)

The collection of languages associated with context-free grammars are called the **context-free languages**.

They include all the regular languages and many additional languages.

A **formal definition of context-free grammars** and study the properties of context-free languages.

Pushdown Automata, a class of machines recognizing the context-free languages. Pushdown automata allow us to gain additional insight into the power of context-free grammars.

CONTEXT FREE GRAMMAR (CFG)

A CFG has four components

1) A set of terminal symbols, sometimes referred to as “tokens”. Terminals are the elementary symbols of the language defined by the grammar.
2) A set of non-terminals sometimes called “syntactic variables”. Each non-terminal represents a set of strings of terminals

   In stmt → if (expr) stmt else stmt

   stmt and expr are non-terminals.
3) A set of productions

   Each production consists of a non-terminal, called head or left side of the production, an arrow, and a sequence of terminals and/or non-terminals, called the body or right side of the production.
4) A designation of one of the non-terminals as the start symbol (head)

Regular Languages

- Closed under Union, Concatenation and Closure ($*$)
- Recognizable by finite-state automata
- Denoted by Regular Expressions
- Generated by Regular Grammars

CONTEXT FREE GRAMMAR (CFG)

- More general productions than regular grammars

$$S \rightarrow \boldsymbol{w}$$

where $\boldsymbol{w}$ is any string of terminals and non-terminals

- What languages do these grammars generate?

$\mathbf{S \rightarrow (A)}$ $\mathbf{A \rightarrow \varepsilon \mid aA \mid ASA}$	$\mathbf{S \rightarrow \varepsilon \mid aSb}$

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular
  - ▸ but it *is* context-free

- Why are they called "context-free"?

▸ Context-sensitive grammars allow more than one symbol on the left-hand side of productions

  – xAy → x(S)y can only be applied to the non-terminal A when it is in the *context* of x and y

Example derivation in a Grammar

- Grammar: start symbol is *A*

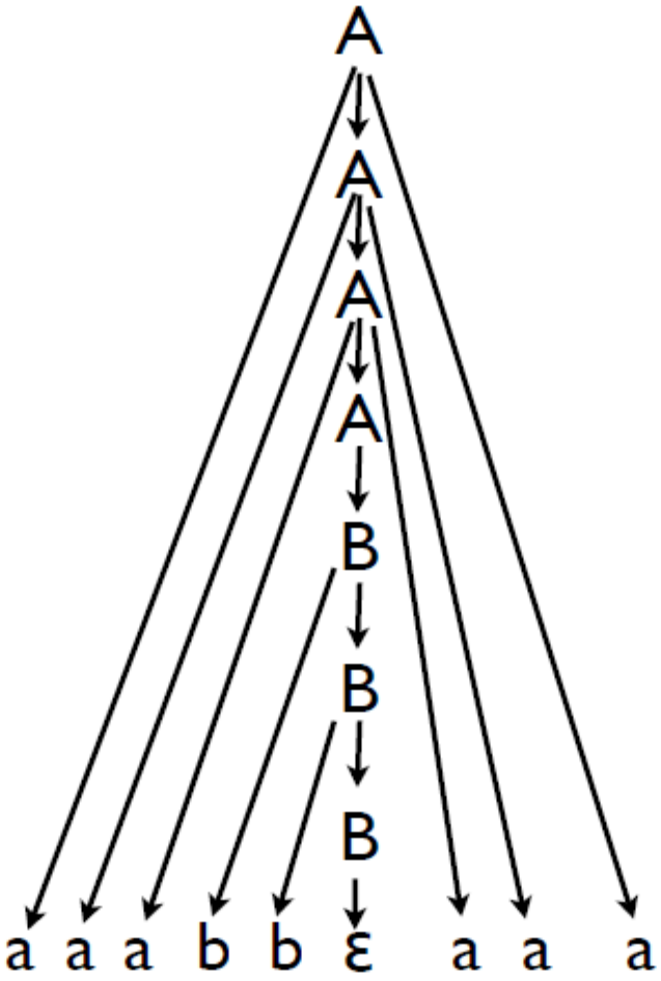
$$A \rightarrow \mathrm{a}A\mathrm{a}$$
$$A \rightarrow B$$
$$B \rightarrow \mathrm{b}B$$
$$B \rightarrow \varepsilon$$

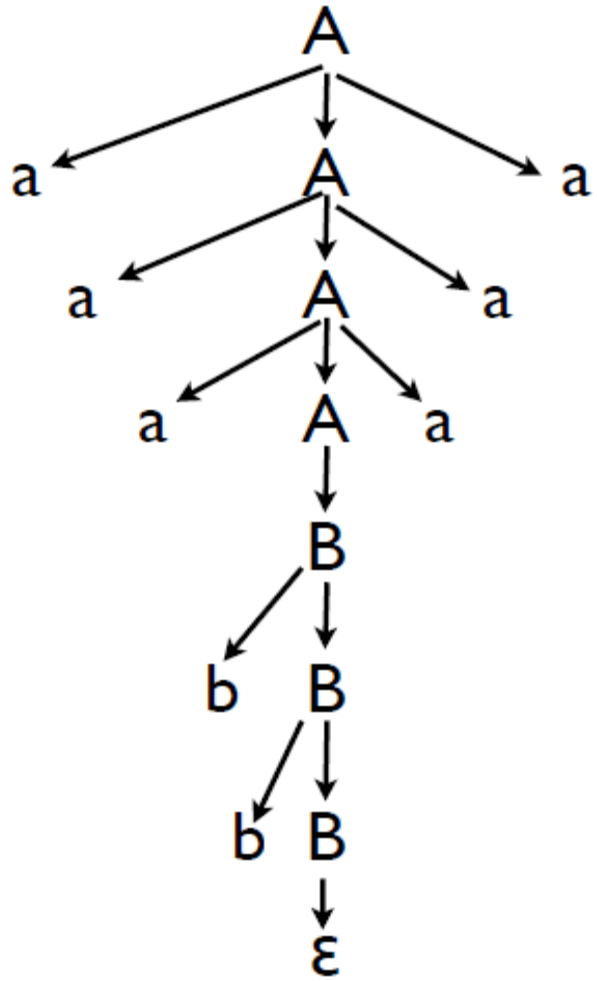
- Sample Derivation:

$\underline{\mathrm{A}} \Rightarrow \mathrm{a}\underline{\mathrm{A}}\mathrm{a} \Rightarrow \mathrm{aa}\underline{\mathrm{A}}\mathrm{aa} \Rightarrow \mathrm{aaa}\underline{\mathrm{A}}\mathrm{aaa} \Rightarrow \mathrm{aaa}\underline{\mathrm{B}}\mathrm{aaa} \Rightarrow \mathrm{aaab}\underline{\mathrm{B}}\mathrm{aaa}$
$\Rightarrow \mathrm{aaabb}\underline{\mathrm{B}}\mathrm{aaa} \Rightarrow \mathrm{aaabbaaa}$

- Language?

Derivations in Tree Form





Example CFG

- An example of a context-free grammar, which we call $\mathbf{G_1}$.

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

Collection of substitution rules, called **productions**. Each rule appears as a line in the grammar, comprising a symbol and a **string** separated by an arrow. The symbol is called a **variable**.

The string consists of **variables** and **terminals**.

The **variable** symbols often are represented by capital letters.

The **terminals** are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols.

One variable is designated as the **start variable**. It usually occurs on the left-hand side of the topmost rule.

Example CFG

- An example of a context-free grammar, which we call $\mathbf{G_1}$.

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

Grammar $\mathbf{G_1}$ contains three rules.

$\mathbf{G_1}$'s variables are A and B, where A is the start variable.

Its terminals are 0, 1, and #.

Example CFG

Grammar is used to describe a language by generating each string of that language in the following manner.

1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
3. Repeat step 2 until no variables remain.

Example-1 CFG

Context-free grammar, $\mathbf{G_1}$.

$$A \rightarrow 0A1$$

$$A \rightarrow B$$

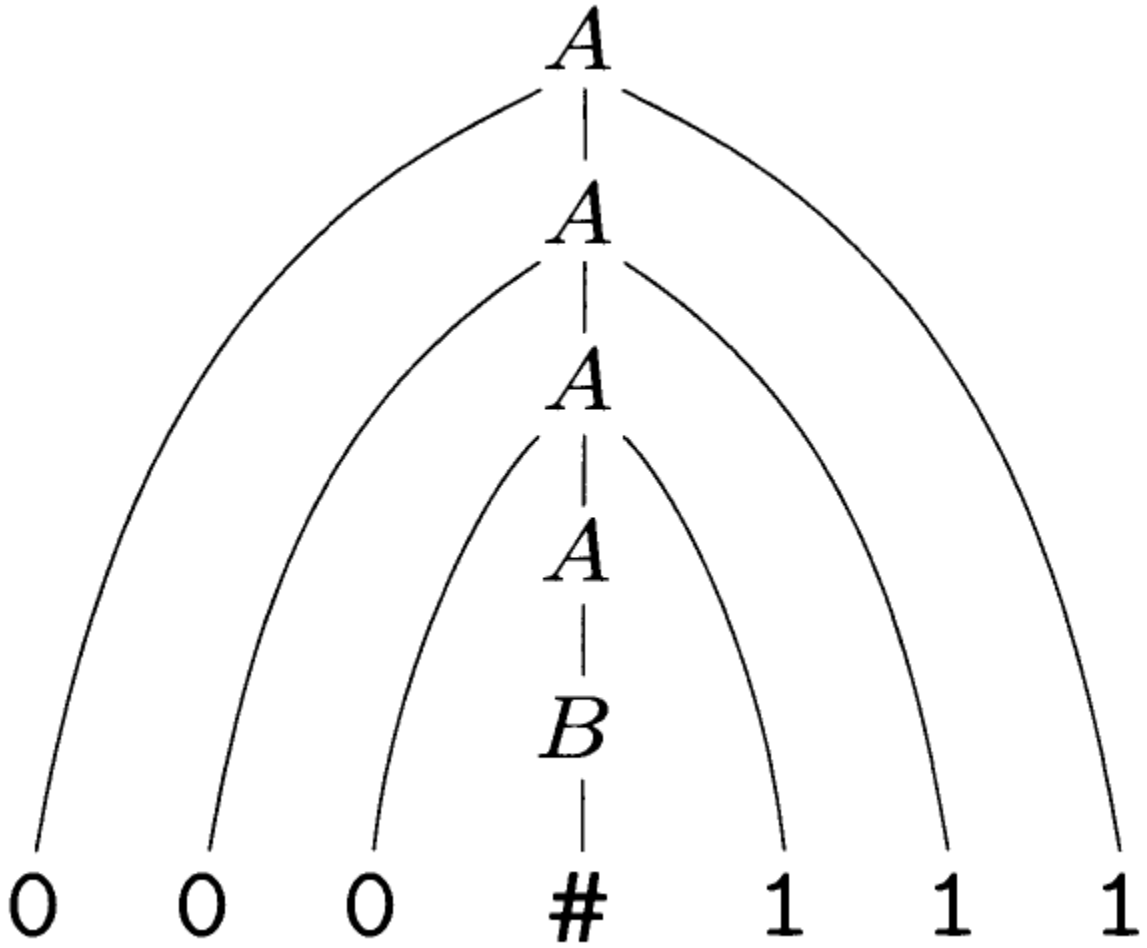
$$B \rightarrow \#$$

- Grammar G 1 generates the string 000#111.

The sequence of substitutions to obtain a string is called a derivation. A derivation of string 000#111 in grammar G 1 is

A ⇒ OA1 ⇒ 00A11 ⇒ 000A111 ⇒ 000B111 ⇒ 000#111

Parse tree for 000#111 in G_1



Example CFG

- All strings generated in this way constitute the ***language of the grammar***.
- We write **$L(G_1)$** for the language of grammar **G_1**.

Grammar G_1 shows that **$L(G_1)$** is $\{0^n\#1^n \mid n > 0\}$.

Any language that can be generated by some context-free grammar is called a Context-Free Language (CFL).

FORMAL DEFINITION OF A CONTEXT FREE GRAMMARS (CFG)

A context-free grammar is a 4-tuple (V, Σ, R, S), where

1. **V** is a finite set called the **variables**
2. **Σ** is a finite set, disjoint from V, called the ***terminals***
3. **R** is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. **S** ∈ **V** is the start variable.

Example-1 CFG

Context-free grammar, $\mathbf{G_1}$.

$$A \rightarrow 0A1$$

$$A \rightarrow B$$

$$B \rightarrow \#$$

In grammar $\mathbf{G_1}$, $V = \{A, B\}$, $\Sigma = \{0, 1, \#\}$, $S = A$, and R is the collection of the three rules

Example-2 CFG

- Consider grammar $\mathbf{G_2}$ = ({***S***}, { a, b }, ***R***, ***S***).

The set of rules, R, is S $\rightarrow$ a*S*b I *SS* I ε

This grammar generates strings such as abab, aaabbb, and aababb.

You can see more easily what this language is if you think of a as a left parenthesis" (" and b as a right parenthesis") ". Viewed in this way, L($\mathbf{G_2}$) is the language of all strings of properly nested parentheses.

Example-3 CFG

Consider grammar $G_3 = (V, \Sigma, R, \langle EXPR \rangle)$.

V is {<EXPR>, <TERM>, <FACTOR>} and

Σ is {a, +, x, (,)}.

The rules are

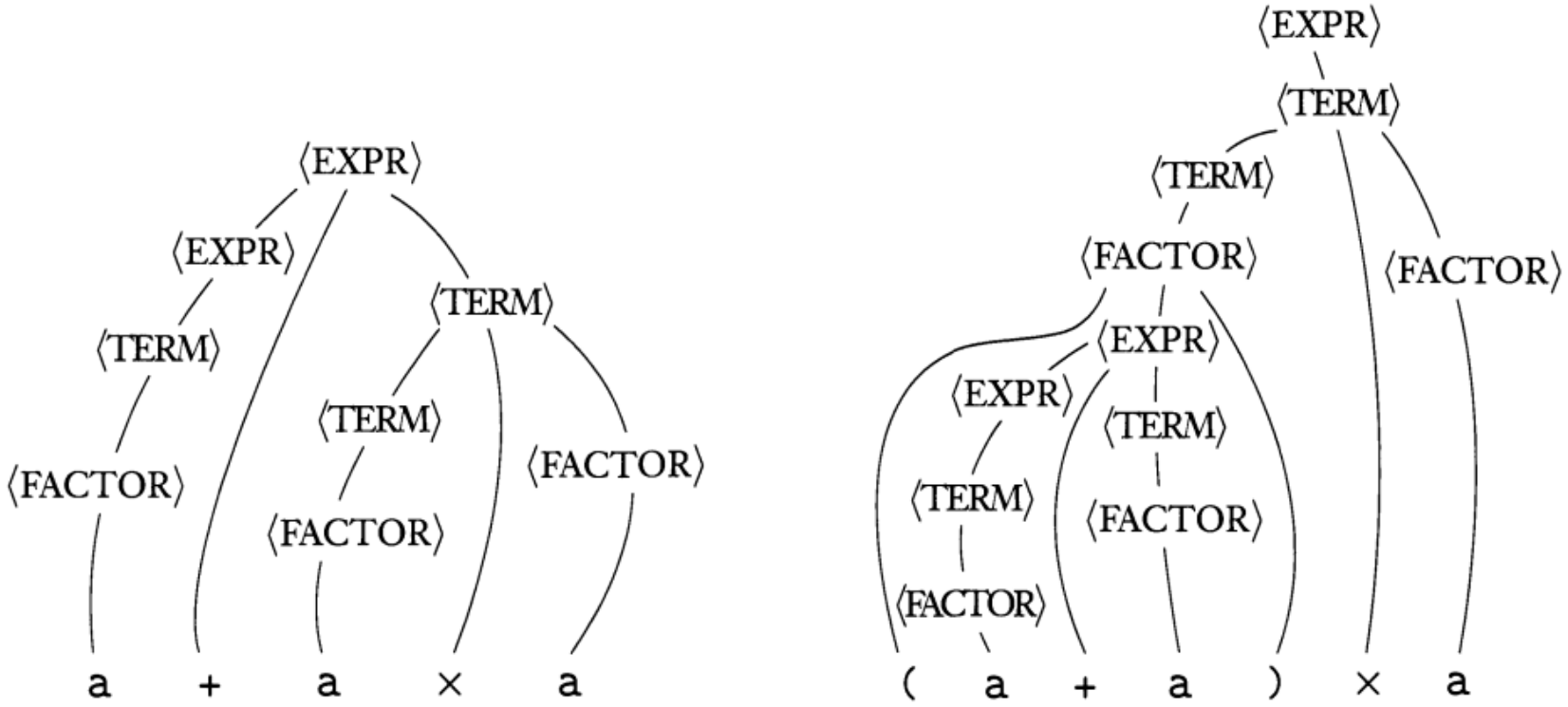
<EXPR) → <EXPR> **+** <TERM> | <TERM>

<TERM> → <TERM> **x** <FACTOR> | <FACTOR>

<FACTOR> → (<EXPR>) | a

<EXPR) → <EXPR> + <TERM> | <TERM>
<TERM> → <TERM> x <FACTOR> | <FACTOR>
<FACTOR> → (<EXPR>) | a

The two strings **a+axa** and **(a+a)xa** can be generated with grammar G_3.
The parse trees are shown in the following figure.



Example – Designing CFGs

- Grammar for the language

$\{0^n 1^n \mid n \geq 0\}$ U $\{1^n 0^n \mid n \geq 0\}$

First construct the grammar

$\mathbf{S_1} \rightarrow 0\ \mathbf{S_1}\ 1 \mid \varepsilon$

for the language $\{0^n 1^n \mid n \geq 0\}$ and the grammar

$\mathbf{S_2} \rightarrow 1\ \mathbf{S_2}\ 0 \mid \varepsilon$

for the language $\{1^n 0^n \mid n \geq 0\}$

And then add the rule $\mathbf{S} \rightarrow \mathbf{S_1} \mid \mathbf{S_2}$

Grammar:

$\mathbf{S} \rightarrow \mathbf{S_1} \mid \mathbf{S_2}$

$\mathbf{S_1} \rightarrow 0\ \mathbf{S_1}\ 1 \mid \varepsilon$

$\mathbf{S_2} \rightarrow 1\ \mathbf{S_2}\ 0 \mid \varepsilon$

Ambiguous Grammar

Sometimes a grammar can generate the same string in several different ways. Such a string will have several different parse trees and thus several different meanings.

In a programming language, its important that a a given program should have a unique interpretation.

Ambiguous Grammar

If a grammar generates the same string in several different ways, then that string is derived ambiguously in that grammar.

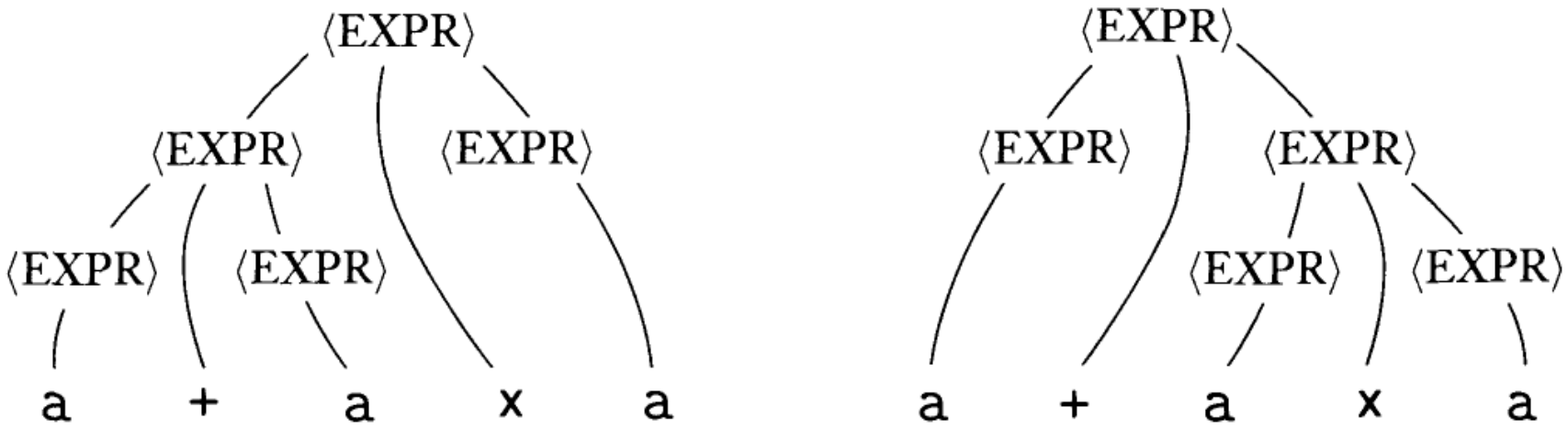
If a grammar generates some string ambiguously we say that the grammar is ambiguous.

For example, consider the following grammar:

<EXPR> → <EXPR>+<EXPR> | <EXPR> x <EXPR> | (<EXPR>) | a

This grammar generates the string **a+axa** ambiguously.

The following figure shows the two different parse trees.



AMBIGUOUS GRAMMAR

A string *w* is derived *ambiguously* in context-free grammar G if it has two or more different leftmost derivations.

Grammar G is *ambiguous* if it generates some string ambiguously.

AMBIGUOUS GRAMMAR

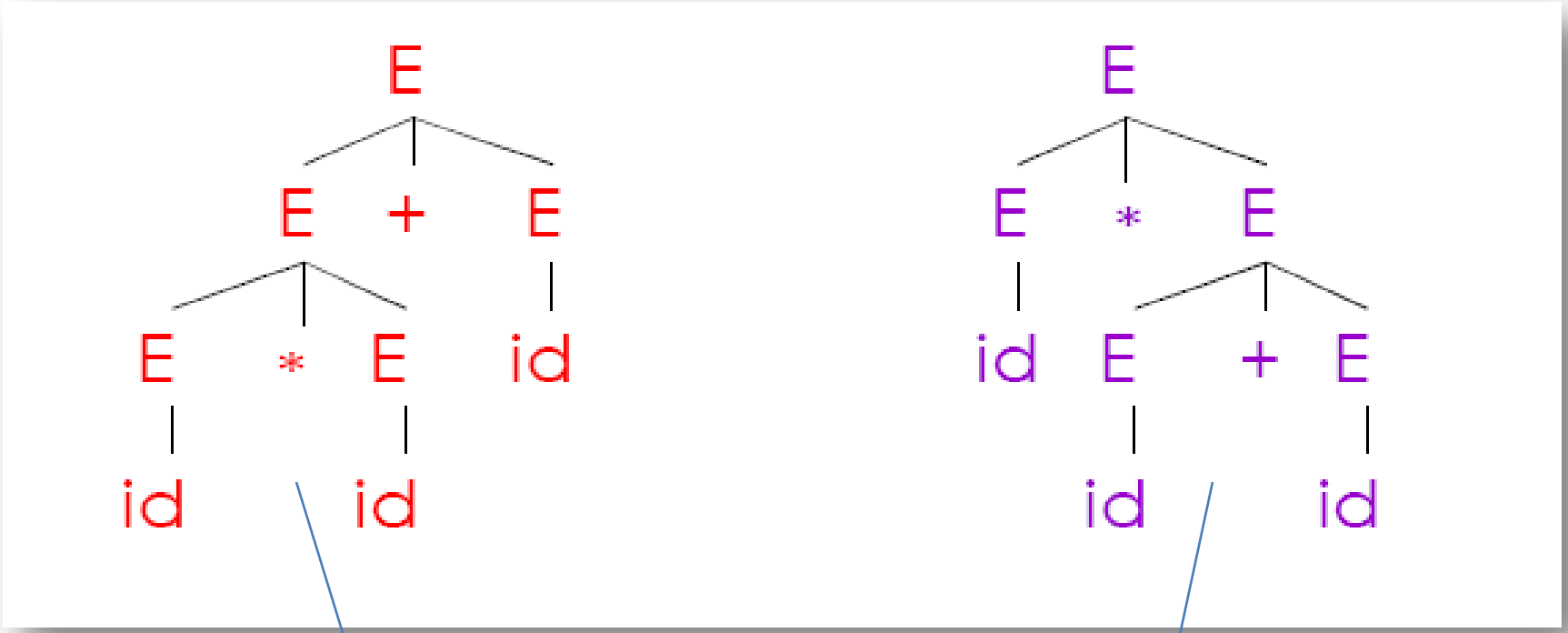
- A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be ambiguous

- Grammar is ambiguous, a terminal string that yield of more than one parse tree.

AMBIGUITY

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

String id * id + id has the following two parse trees



Enforces precedence of * over +

Doesn't enforce this precedence

Dealing with Ambiguity

- The most direct way is to re-write the grammar unambiguously

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

$$E \rightarrow E' + E \mid E'$$

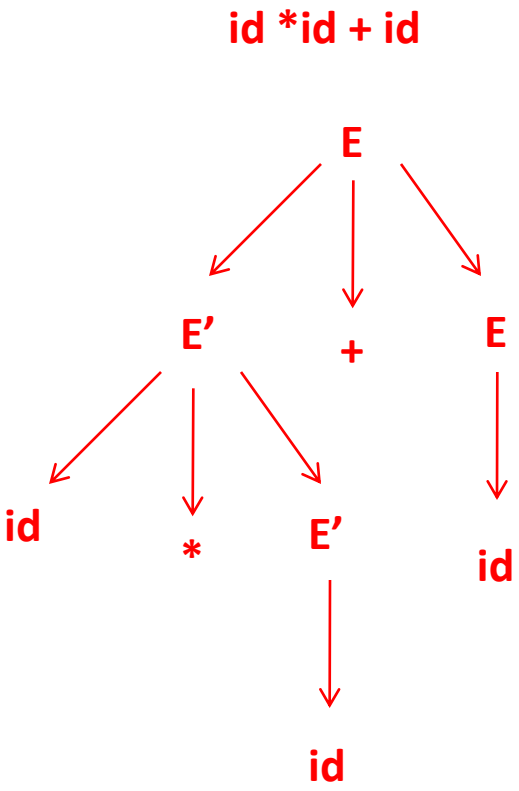
$$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$$

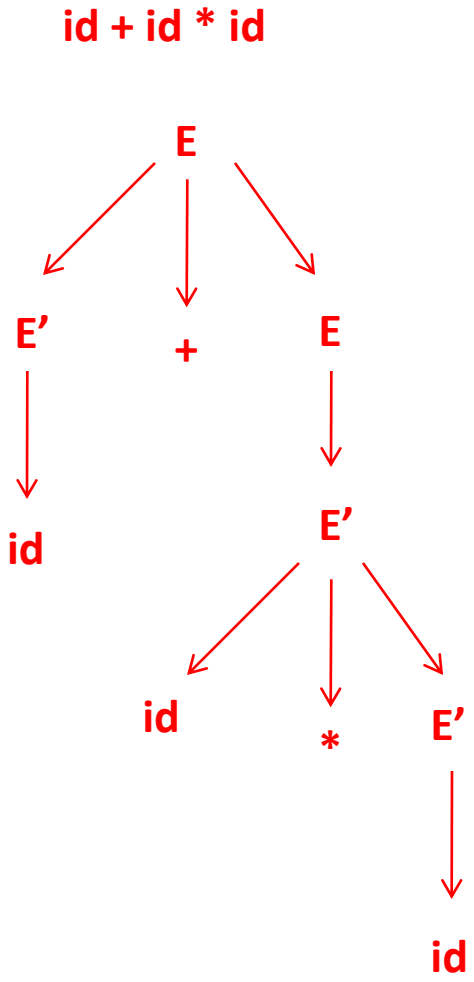
Enforces precedence of * over +

EXAMPLE

$$E \rightarrow E'+E \mid E'$$

$$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$$





EXAMPLE

Another Ambiguous Grammar

- S → x
- S → y
- S → z
- S → S + S
- S → S – S
- S → S * S
- S → S / S
- S → (S)

Generates two parse trees for x + y * z

Rewrite it as:

- T → x
- T → y
- T → z
- S → S + T
- S → S – T
- S → S * T
- S → S / T
- T → (S)
- S → T

Enforces precedence of * over +

TRY DIFFERENT INPUTS AT HOME

CONTEXT FREE GRAMMAR (CFG)

Example

Java if-else statement

If (expression) statement else statement

`stmt → if (expr) stmt else stmt`

The arrow may be read as "can have the form". Such a rule is called a production

CONTEXT FREE GRAMMAR (CFG)

Example

```
list → list + digit
list → list - digit
list → digit
digit → 0|1|2|3|4|5|6|7|8|9
```

```
list → list + digit | list – digit | digit
```

The terminals are + - 0 1 2 3 4 5 6 7 8 9

CONTEXT FREE GRAMMAR (CFG)

Example

Function call

```
call → id(optparams)
optparams → params | ϵ
params → params, param | param
```

CONTEXT FREE GRAMMAR (CFG)

- **Example**
- Operators on the same line have the same associativity and precedence

  left-associative: + -

  left-associative: * /

Two non-terminals expr and term for the two levels of precedence, non-terminal factor for generating basic units in expressions

CONTEXT FREE GRAMMAR (CFG)

```
factor → digit | (expr)
```

Binary operators * and / have the highest precedence

```
term →  term * factor
            | term / factor
            | factor
Similarly, expr
expr →      expr + term
            | expr - term
            | term
```

The resulting grammar is therefore

```
expr → expr + term | expr – term | term
term → term * factor | term / factor | factor
factor → digit | (expr)
```

CONTEXT FREE GRAMMAR (CFG)

Example:

A grammar for a subset of Java statement

```
stmt → id = expression;
     | if(expression) stmt
     | if(expression) stmt else stmt
     | while(expression) stmt
     | do stmt while (expression);
     | {stmts}


stmts →  stmts stmt
         | ϵ
```

CONTEXT FREE GRAMMAR (CFG)

Example:

Grammar for statement blocks and conditional statements:

```
stmt → if expr then stmt else stmt
     | if stmt then stmt
     | begin stmtList end

stmtList → stmt; stmtList | stmt
```

CONTEXT FREE GRAMMAR (CFG)

- **Exercise**

Consider the context-free grammar

$$S \rightarrow SS+ \mid SS* \mid a$$

(a) Show how the string aa+a* can be generated by this grammar

(b) Construct a parse tree for this string

(c) What language does this grammar generate? Justify your answer.

CFG vs. Regular Expression

- **CFGs are a more powerful notation than regexes**
  - Every construct that can be described by a regex can also be described by the CFG, but not vice-versa
  - Every regular language is a context-free language, but not vice versa.

CFG vs. Regular Expression

Regex: (a|b)*abb

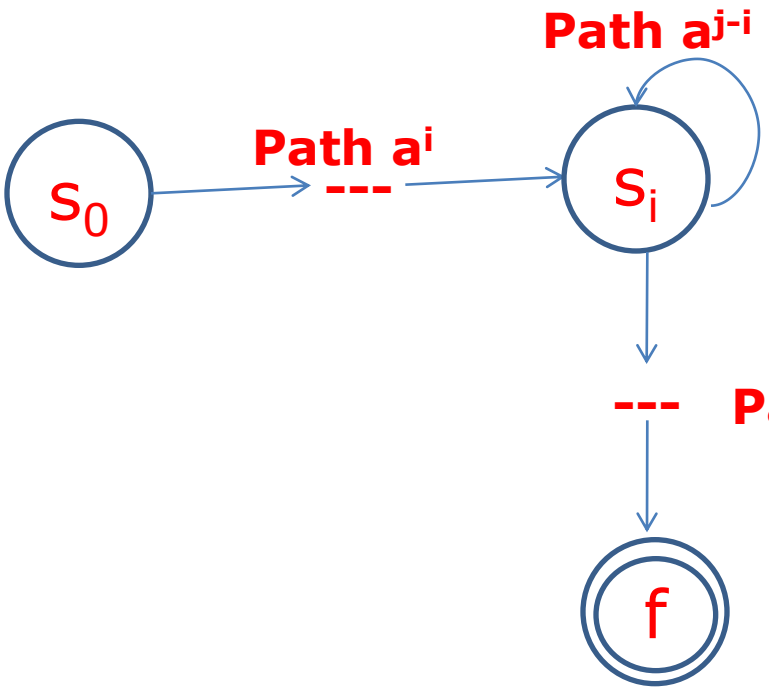
Describe the same language: the set of strings of a's and b's ending with abb

Grammar:

$$A \rightarrow aA_0 \mid bA_0 \mid aA_1$$
$$\rightarrow bA_2$$
$$\rightarrow bA_3$$
$$\rightarrow \in$$

CFG vs. Regular Expression

- Language L = {$a^n b^n$ | n>=1} can be described by a grammar but not by a regex
- Suppose L was defined by some regex
  - We could construct a DFA with a finite number of states, say k, to accept L



State s_i: For an input beginning with more than *k* a's

$a^i b^i$ is in the language: A path b^i from s_i to state *f*

Path $a^j b^i$ is also possible

This DFA accepts both $a^i b^i$ and $a^j b^i$

DFA cannot count, i.e., keep track of the number of a's before it sees the b's

Context-free grammars are widely used for programming languages

- From the definition of Algol-60:

procedure_identifier::= identifier.
actual_parameter::= string_literal | expression | array_identifier | switch_identifier | procedure_identifier.
letter_string::= letter | letter_string letter.
parameter_delimiter::= "," | ")" letter_string ":" "(".
actual_parameter_list::= actual_parameter | actual_parameter_list parameter_delimiter actual_parameter.
actual_parameter_part::= empty | "(" actual_parameter_list} ")".
function_designator::= procedure_identifier actual_parameter_part.

EXAMPLE

adding_operator::= "+" | "−" .
multiplying_operator::= "×" | "/" | "÷" .
primary::= unsigned_number | variable | function_designator | "(" arithmetic_expression ")".
factor::= primary | factor | factor power primary.
term::= factor | term multiplying_operator factor.
simple_arithmetic_expression::= term | adding_operator term |
simple_arithmetic_expression adding_operator term.
if_clause::= if Boolean_expression then.
arithmetic_expression::= simple_arithmetic_expression |
if_clause simple_arithmetic_expression else arithmetic_expression.

if a < 0 then U+V else if a * b < 17 then U/V else if k <> y then V/U else 0

NFA To CFG Conversion

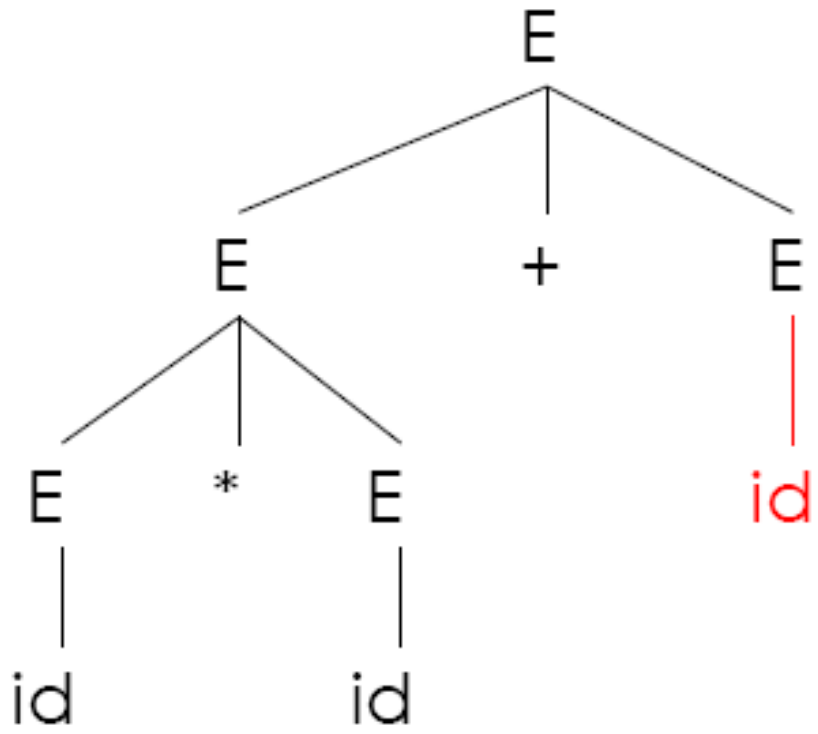
- **We can mechanically construct the CFG from an NFA**
- Converting the NFA for (a|b)*abb into CFG
  - For each state *i* of the NFA, create a non-terminal A_i
  - If *i* has a transition to *j* on input *a*, add $A_i \rightarrow aA_j$
  - If *i* has a transition to *j* on input ε, add $A_i \rightarrow A_j$
  - If *i* is an accepting state, add $A_i \rightarrow \in$
  - If *i* is the start state, make A_i the start symbol of the grammar.

BNF: Meta-Syntax for CFGs

- <postal-address> **::=** <name-part> <street-address>
  <zip-part>
- <name-part> **::=** <personal-part> <last-name>
  <opt-jr-part> <EOL>
  | <personal-part> <name-part>
- <personal-part> **::=** <first-name> **|** <initial> "."
- <street-address> **::=** <house-num> <street-name>
  <opt-apt-num> <EOL>
- <zip-part> **::=** <town-name> "," <state-code>
  <ZIP-code> <EOL>
- <opt-jr-part> **::=** "Sr." **|** "Jr." **|** <roman-numeral> **|** ""

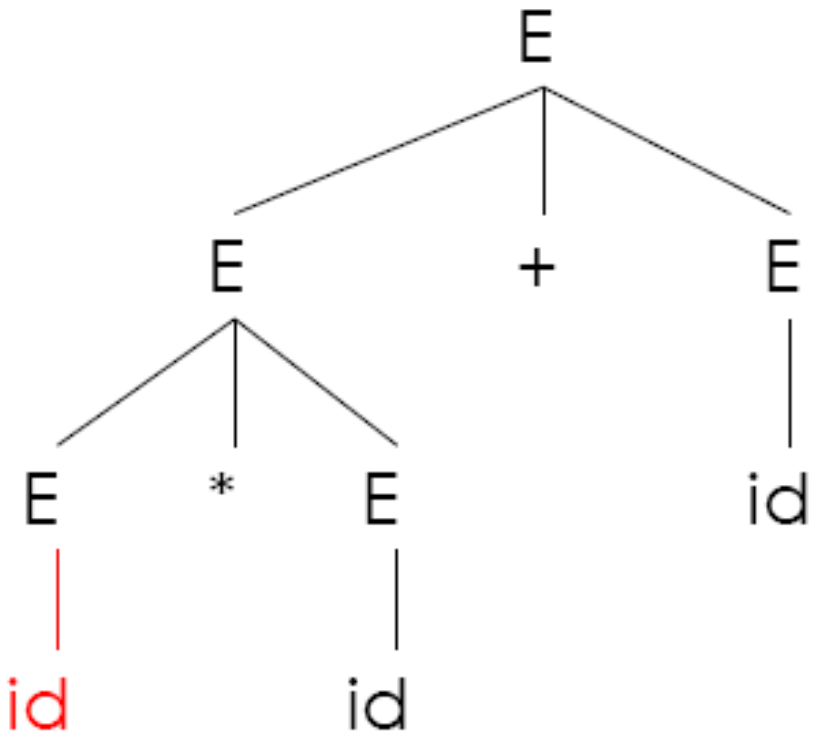
Left-Most Derivation Parse Tree

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Right-Most Derivation Parse Tree

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E*E+id \\
\rightarrow\ & E*id+id \\
\rightarrow\ & id*id+id
\end{aligned}
$$



Ambiguity

- **There are no general techniques for handling ambiguity**
- **It is impossible to automatically convert an ambiguous grammar into an unambiguous one**
- **If used sensibly, ambiguity can simplify the grammar**
- **Disambiguation Rules**: Instead of re-writing the grammar, we can
  - Use the ambiguous grammar
  - Along with disambiguation rules.

Associativity of Operators

- The operator + associates to the left

An operator with + signs on both sides of it belongs to the operator to its left.

In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are left associative.

RIGHT ASSOCIATIVE OPERATOR

- The operator = associates to the right

```
right → letter = right | letter
letter → a | b |…| z
```

Parse tree for 9 – 5 – 2 grows down towards the left, whereas parse tree for a=b=c grows down towards the right

PRECEDENCE OF OPERATORS

- Associativity rules for + and * apply to occurrences of the same operator
- **Rule**

* has the highest precedence than + if * takes its operands before + does

- Multiplication and division have higher precedence than addition and subtraction.
- 9 + 5 * 2 and 9 * 5 + 2 equivalent to 9 + (5 * 2) and (9 * 2) + 2

QUESTIONS

- CFG representing the Regular Expression a^+

$$\mathbf{A \rightarrow aA \mid a}$$

- CFG representing the Regular Expression b*

$$\mathbf{B \rightarrow}$$

$$\mathbf{B \rightarrow bB}$$

QUESTIONS

- CFG representing the Regular Expression a^*b^+

  (i.e. start with any number of a's followed by non-zero numbers of b)

  $$\mathbf{S \rightarrow R \mid aS}$$
  $$\mathbf{R \rightarrow b \mid bR}$$

- A CFG representing the Regular Expression ab^+a

  (i.e. start with a followed by non-zero numbers of b's and ends with a)

  $$\mathbf{S \rightarrow aRa}$$
  $$\mathbf{R \rightarrow b \mid bR}$$

QUESTIONS

Every construct described by a Regular Expression can also be described by a CFG. Consider the following regular expression:

(a|b)*abb where Σ = {a, b}
Create an equivalent CFG of the above regular expression

$$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$$
$$A_1 \rightarrow bA_2$$
$$A_2 \rightarrow bA_3$$
$$A_3 \rightarrow \epsilon$$