



Context-Free Grammar (CFG)

Dr. Nadeem Akhtar

Assistant Professor

Department of Computer Science & IT

The Islamia University of Bahawalpur

PhD – Computer Science

IRISA – University of South Brittany – Bretagne - FRANCE.

Introduction

There are four important components in a grammatical description of a language:

1. There is a finite set of symbols that form the strings of the language being defined. We call this alphabet the *terminals*, or *terminal symbols*.
2. There is a finite set of variables, also called sometimes *non-terminals* or *syntactic categories*. Each variable represents a language; i.e., a set of strings.
3. One of the variables represents the language being defined; it is called the start symbol. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol.

Introduction

4. There is a finite set of *productions* or *rules* that represent the recursive definition of a language. Each production consists of:
 - (a) A variable that is being (partially) defined by the production. This variable is often called the *head* of the production.
 - (b) The production symbol \rightarrow
 - (c) A string of zero or more terminals and variables. This string, called the body of the production, represents one way to form strings in the language of the variable of the head. In so doing, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of that variable.

The four components just described form a context-free grammar, or just grammar, or CFG. We shall represent a CFG by its four components, that is, $G = (V, T, P, S)$, where V is the set of variables, T the terminals, P the set of productions, and S the start symbol.

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 \rightarrow 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)

Formal definition of Context-Free Grammar

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. $S \in V$ is the start variable.

Regular Languages

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

Context-Free Grammars

- More general productions than regular grammars

$$S \rightarrow w$$

where w is any string of terminals and non-terminals

- What languages do these grammars generate?

$$S \rightarrow (A)$$

$$A \rightarrow \epsilon \mid aA \mid ASA$$

$$S \rightarrow \epsilon \mid aSb$$

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 *LHS* of productions
 - $xAy \rightarrow x(S)y$ can only be applied to the non-terminal A when it is in the *context* of x and y

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 ::= "x" | "/" | "÷" .

primary ::= unsigned_number | variable | function_designator | "(" arithmetic_expression ")".

factor ::= primary | factor | factor power primary.

term ::= factor | term multiplying_operator factor.

$$\text{simple_arithmetic_expression} ::= \text{term} \mid \text{adding_operator term} \mid$$

simple_arithmetic_expression adding_operator term.

if_clause ::= if Boolean_expression then.

$$\text{arithmetic_expression} ::= \text{simple_arithmetic_expression} \mid$$

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
```

Example derivation in a Grammar

- Grammar: start symbol is A

$$A \rightarrow aAa$$

$$A \rightarrow B$$

$$B \rightarrow bB$$

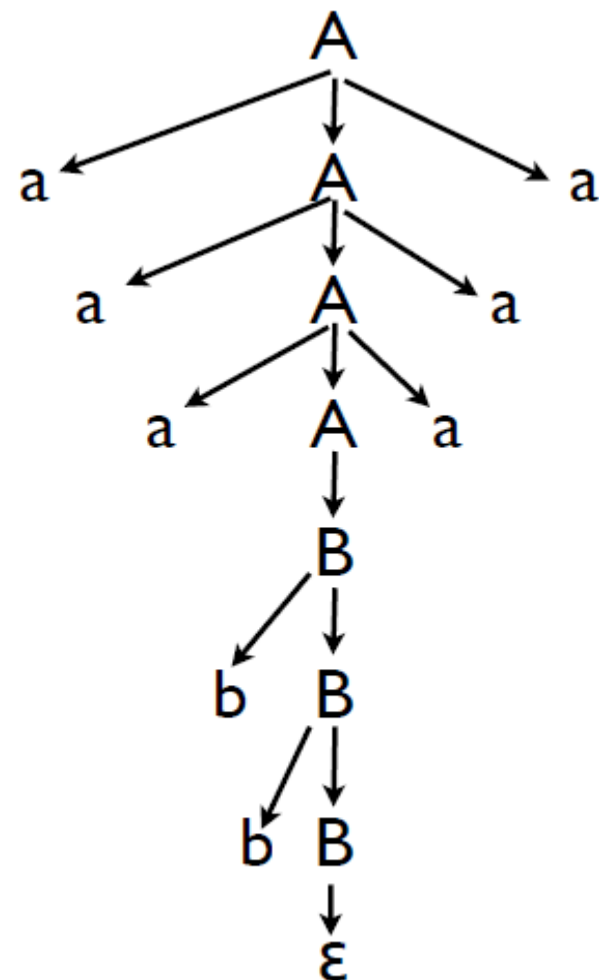
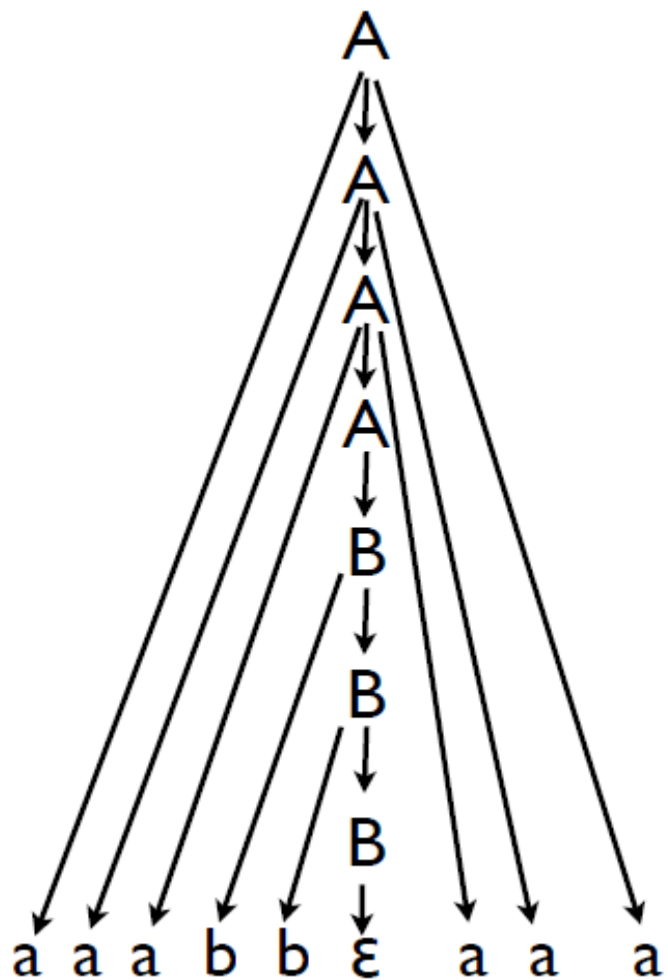
$$B \rightarrow \varepsilon$$

- Sample Derivation:

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

- Language?

Derivations in Tree Form



Example CFG

- An example of a context-free grammar, which we call 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.

Grammar G_1 contains three rules. 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 CFG

Context-free grammar, G_1 .

$$A \rightarrow 0A1$$

$$A \rightarrow B$$

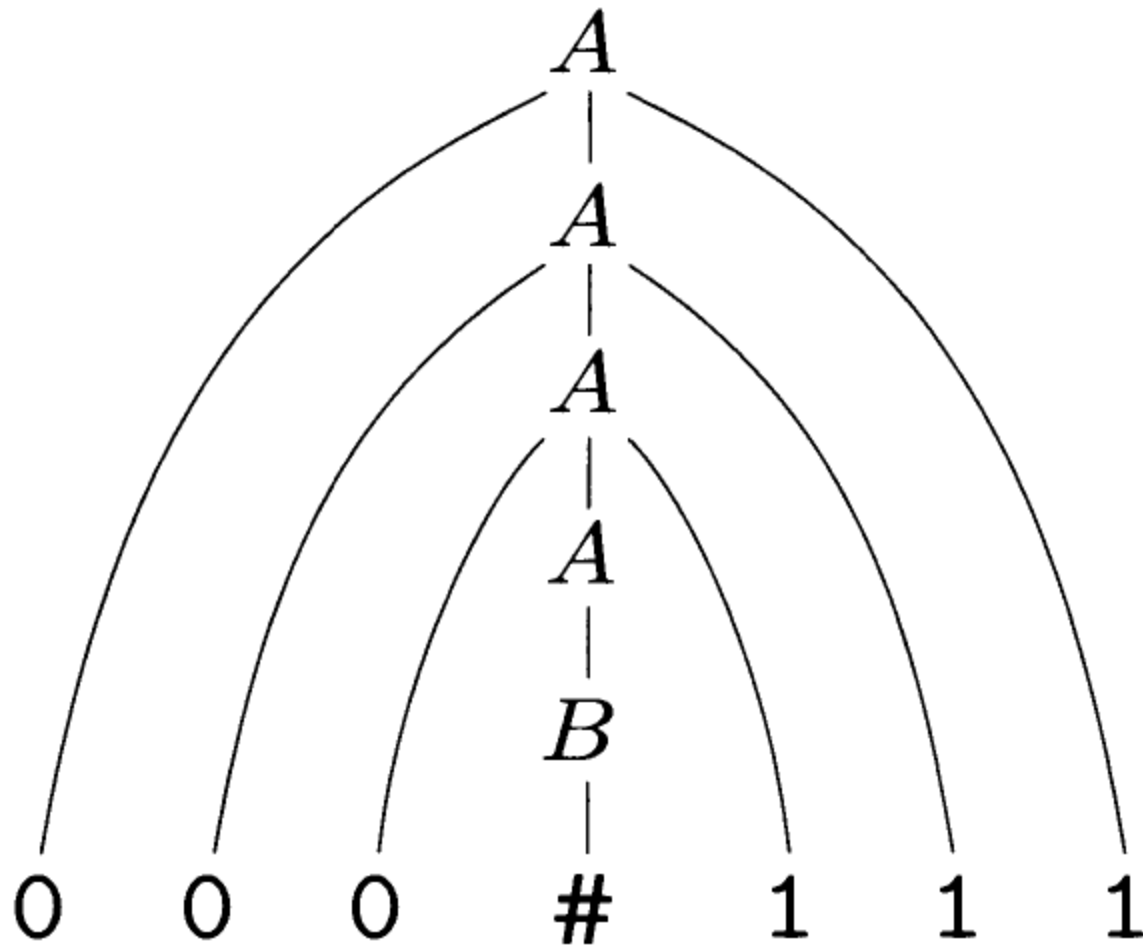
$$B \rightarrow \#$$

- 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

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

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).

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` \rightarrow `list + digit`

`list` \rightarrow `list - digit`

`list` \rightarrow `digit`

`digit` \rightarrow `0|1|2|3|4|5|6|7|8|9`

`list` \rightarrow `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` \rightarrow `id(optparams)`

`optparams` \rightarrow `params` | ϵ

`params` \rightarrow `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)

$\text{factor} \rightarrow \text{digit} \mid (\text{expr})$

Binary operators * and / have the highest precedence

$\text{term} \rightarrow \text{term} * \text{factor}$
 $\quad \quad \quad \mid \text{term} / \text{factor}$
 $\quad \quad \quad \mid \text{factor}$

Similarly, expr

$\text{expr} \rightarrow \text{expr} + \text{term}$
 $\quad \quad \quad \mid \text{expr} - \text{term}$
 $\quad \quad \quad \mid \text{term}$

The resulting grammar is therefore

$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$

$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$

$\text{factor} \rightarrow \text{digit} \mid (\text{expr})$

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

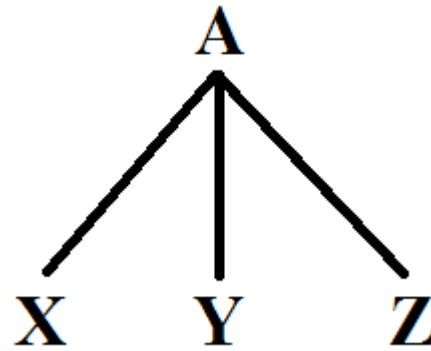
$$S \rightarrow SS^+ \mid SS^* \mid a$$

- (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.

Parse Trees

$A \rightarrow XYZ$

then a parse tree may have an interior node labelled A with three children labeled X , Y , and Z



Parse Trees

Formally, given a context-free grammar, a parse-tree according to the grammar is a tree with the following properties:

- (1) The root is labelled by the start symbol
- (2) Each leaf is labelled by a terminal or by ϵ
- (3) Each interior node is labelled by a non-terminal
- (4) If A is the non-terminal labelling some interior node and X_1, X_2, \dots, X_n are the labels of the children of that node from left to right, then there must be a production

$$A \rightarrow X_1 X_2 \dots X_n$$

Here, $X_1 X_2 \dots X_n$ each stand for a symbol that is either a terminal or a non-terminal.

As a special case, if $A \rightarrow \epsilon$ is a production, then a node labelled A may have a single child labelled ϵ

CFG vs. Regex

- 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. Regex

Regex: $(a|b)^*abb$

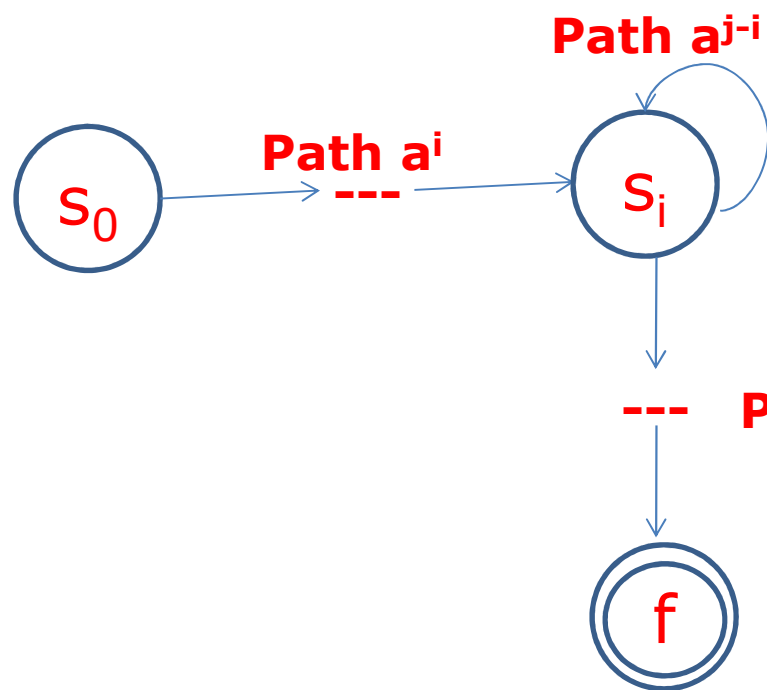
**Describe the same
language: the set of
strings of a's and b's
ending with abb**

Grammar:

$$\begin{aligned} A &\rightarrow aA_0 \mid bA_0 \mid aA_1 \\ &\rightarrow bA_2 \\ &\rightarrow bA_3 \\ &\rightarrow \epsilon \end{aligned}$$

CFG vs. Regex

- Language $L = \{a^n b^n \mid n \geq 1\}$ 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

NFA To CFG Conversion

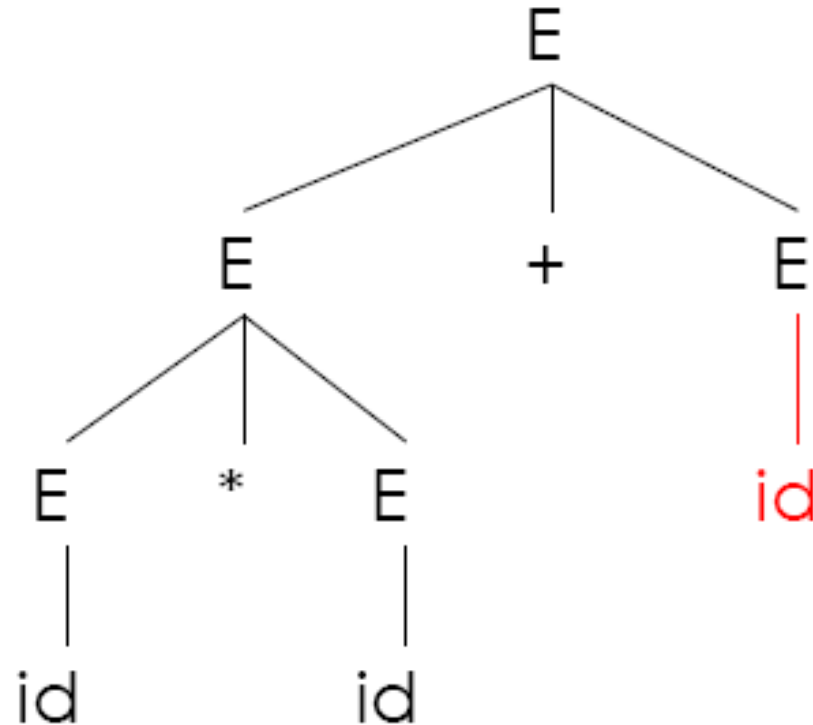
- We can mechanically construct the CFG from an NFA
- Converting the NFA for $(a|b)^*abb$ into CFG
 - For each state i of the NFA, create a non-terminal A_i
 - If i has a transition to j on input a , add $A_i \rightarrow aA_j$
 - If i has a transition to j on input ϵ , add $A_i \rightarrow A_j$
 - If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

BNF: Meta-Syntax for CFGs

- $\langle \text{postal-address} \rangle ::= \langle \text{name-part} \rangle \langle \text{street-address} \rangle$
 $\langle \text{zip-part} \rangle$
- $\langle \text{name-part} \rangle ::= \langle \text{personal-part} \rangle \langle \text{last-name} \rangle$
 $\langle \text{opt-jr-part} \rangle \langle \text{EOL} \rangle$
 $| \langle \text{personal-part} \rangle \langle \text{name-part} \rangle$
- $\langle \text{personal-part} \rangle ::= \langle \text{first-name} \rangle | \langle \text{initial} \rangle "."$
- $\langle \text{street-address} \rangle ::= \langle \text{house-num} \rangle \langle \text{street-name} \rangle$
 $\langle \text{opt-apt-num} \rangle \langle \text{EOL} \rangle$
- $\langle \text{zip-part} \rangle ::= \langle \text{town-name} \rangle ", " \langle \text{state-code} \rangle$
 $\langle \text{ZIP-code} \rangle \langle \text{EOL} \rangle$
- $\langle \text{opt-jr-part} \rangle ::= "Sr." | "Jr." | \langle \text{roman-numeral} \rangle | ""$

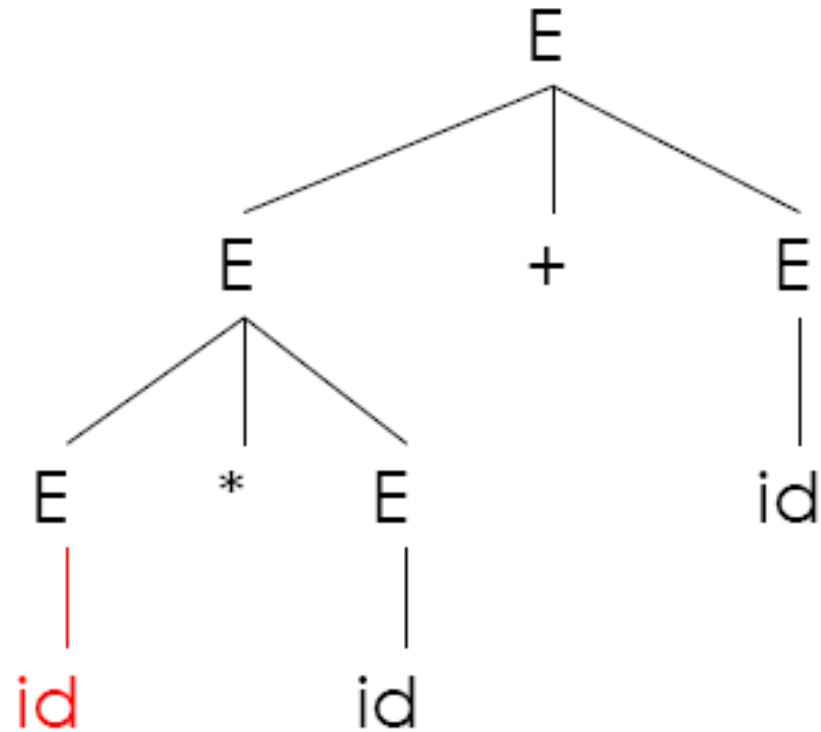
Left-Most Derivation Parse Tree

E
 $\rightarrow E + E$
 $\rightarrow E * E + E$
 $\rightarrow id * E + E$
 $\rightarrow id * id + E$
 $\rightarrow id * id + id$



Right-Most Derivation Parse Tree

E
 $\rightarrow E + E$
 $\rightarrow E + id$
 $\rightarrow E * E + id$
 $\rightarrow E * id + id$
 $\rightarrow id * id + id$



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.

Ambiguous Grammar

If a grammar generates the same string in several different ways, we say that the string is derived ambiguously in that grammar. If a grammar generates some string ambiguously we say that the grammar is ambiguous.

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

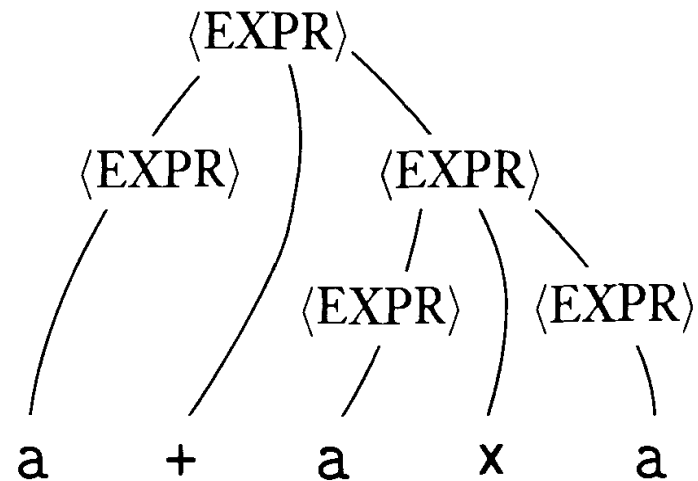
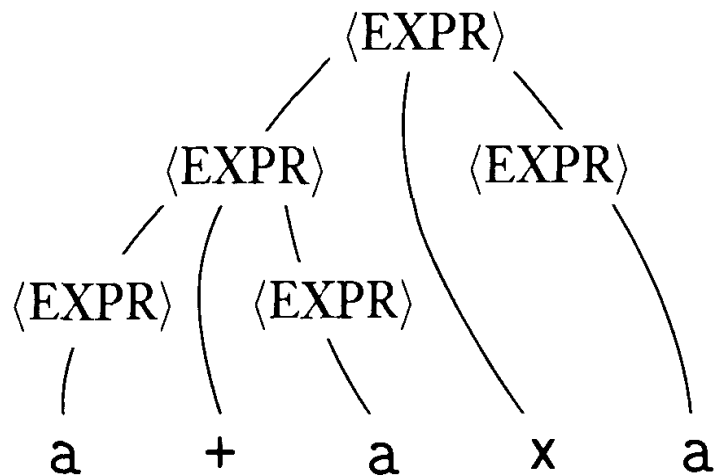
Consider grammar G_2 :

$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle$
 $\mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle$
 $\mid (\langle \text{EXPR} \rangle) \mid a$

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

Ambiguous Grammar

The two parse trees for the string a+axa in grammar G_2



Ambiguity

- **Example**

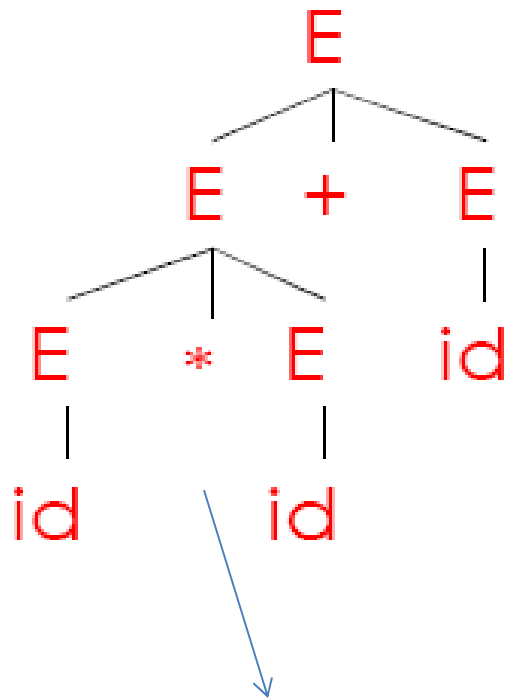
String \rightarrow String + String | String – String

|0|1|2|3|4|5|6|7|8|9

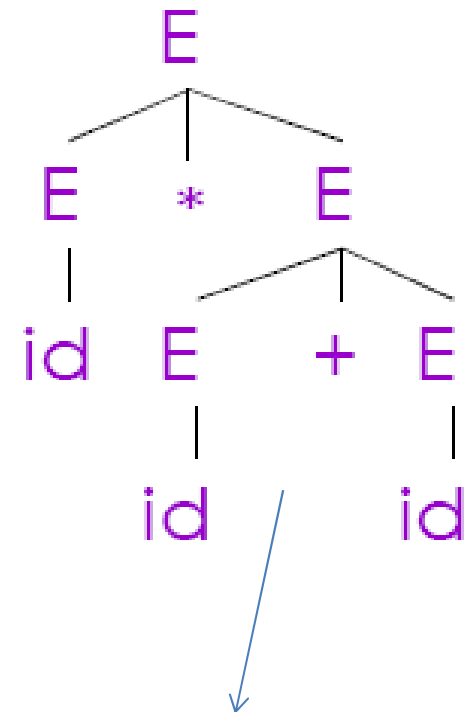
Ambiguity

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

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

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



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

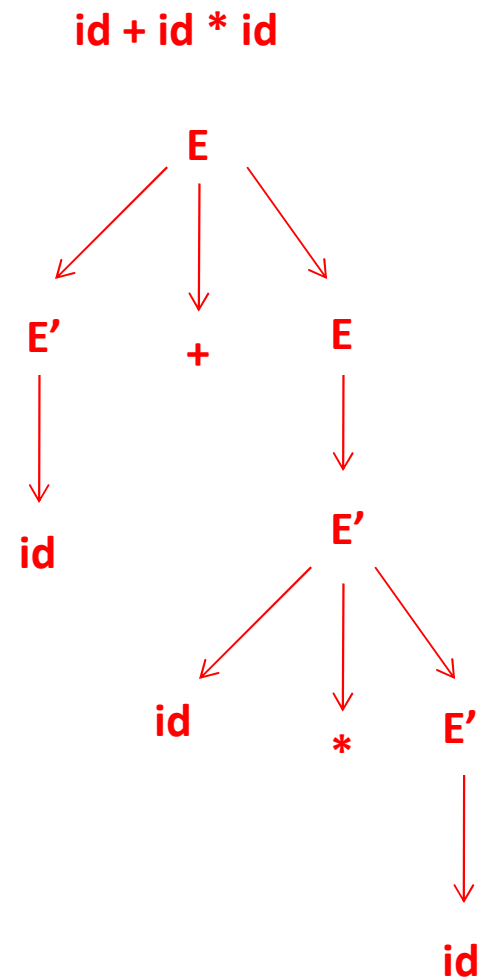
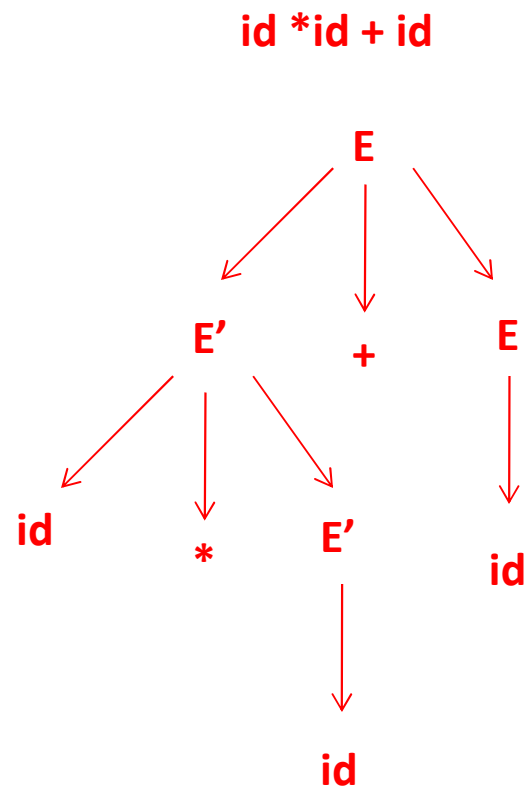
$$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$$

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

- $S \rightarrow x$
- $S \rightarrow y$
- $S \rightarrow z$
- $S \rightarrow S + S$
- $S \rightarrow S - S$
- $S \rightarrow S * S$
- $S \rightarrow S / S$
- $S \rightarrow (S)$

Rewrite it as:

- $T \rightarrow x$
- $T \rightarrow y$
- $T \rightarrow z$
- $S \rightarrow S + T$
- $S \rightarrow S - T$
- $S \rightarrow S * T$
- $S \rightarrow S / T$
- $T \rightarrow (S)$
- $S \rightarrow T$

Generates two parse trees for $x + y * z$

Enforces precedence of $*$ over $+$

TRY DIFFERENT INPUTS AT HOME

Ambiguity – The Dangling Else

$$\begin{aligned} E \rightarrow & \text{if } E \text{ then } E \\ & | \text{if } E \text{ then } E \text{ else } E \\ & | \text{OTHER} \end{aligned}$$

- This is an ambiguous grammar

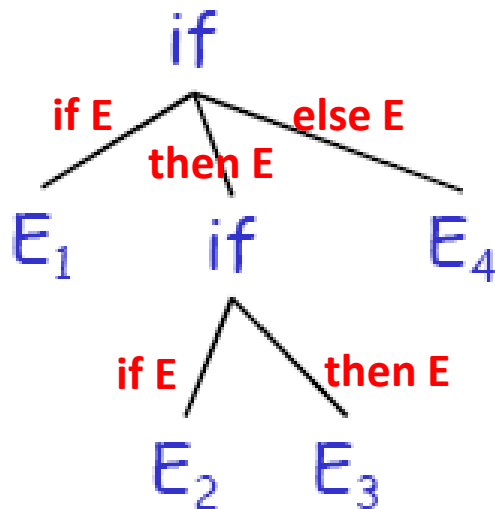
Dangling Else

$$E \rightarrow \begin{array}{l} \text{if } E \text{ then } E \\ | \text{if } E \text{ then } E \text{ else } E \text{ } | \text{ OTHER} \end{array}$$

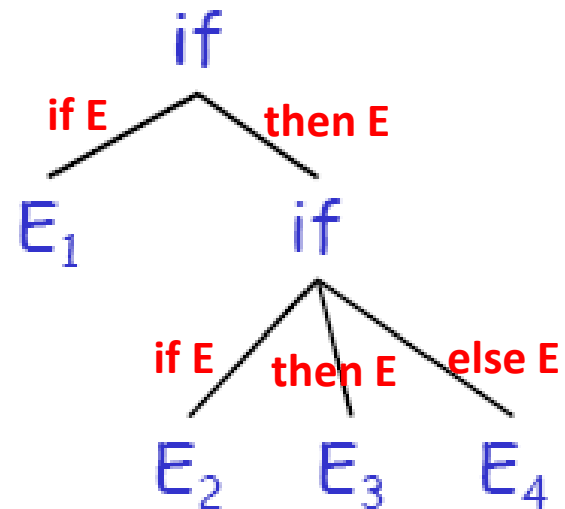
- The expression

if E_1 then if E_2 then E_3 else E_4

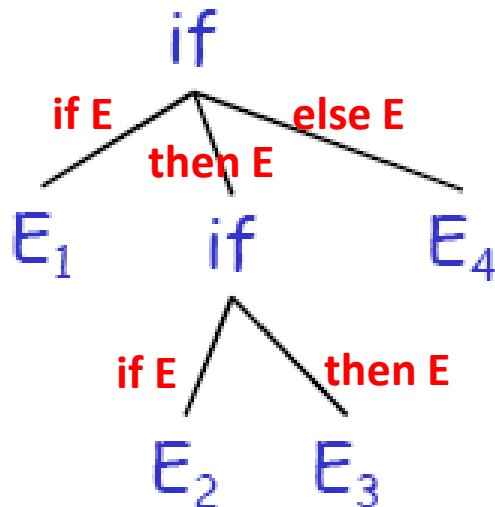
has two parse trees



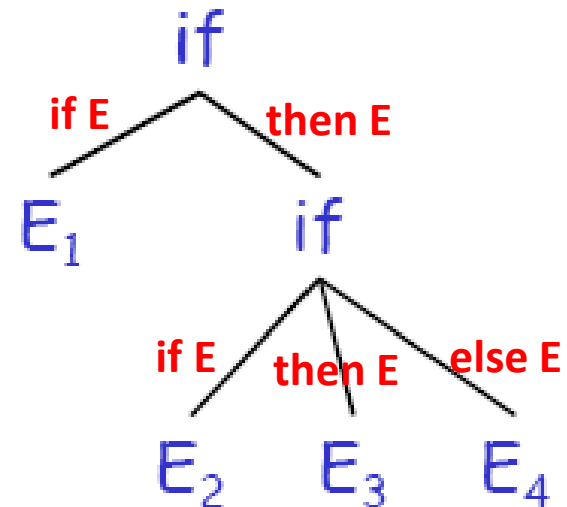
The
'ELSE'
should
be
consider
ed with
which
'THEN'



Dangling Else



The 'ELSE' should be considered with which 'THEN'



Typically we want this parse tree

'Else' matches the closest unmatched 'Then'

Dangling Else

$E \rightarrow \text{matchedIF} \quad // \text{all THEN are matched}$

$\quad | \text{unmatchedF} \quad // \text{some THEN is unmatched}$

$\text{matchedIF} \rightarrow \text{if } E \text{ then matchedIF else matchedIF}$

$\quad | \text{OTHER}$

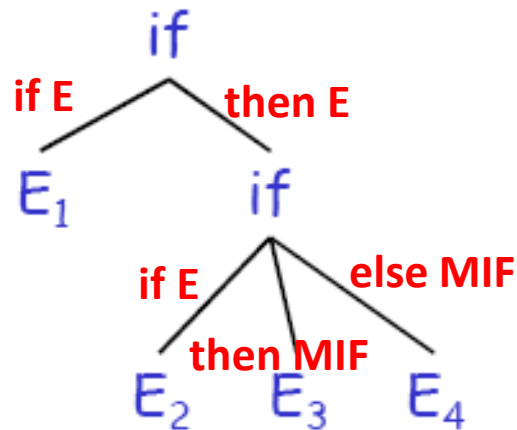
$\text{unmatchedIF} \rightarrow \text{if } E \text{ then } E$

$\quad | \text{if } E \text{ then matchedIF else unmatchedF}$

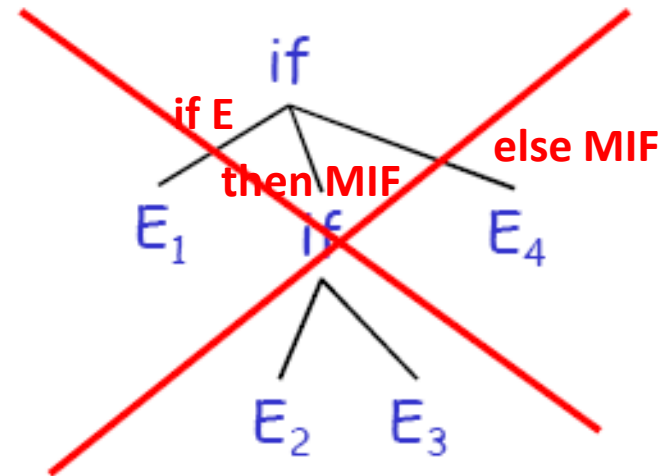
Dangling Else

- Consider again the expression

if E_1 then if E_2 then E_3 else E_4



A Valid Parse Tree for an unmatchedIF



Not Valid because the THEN is not a matchedIF

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.

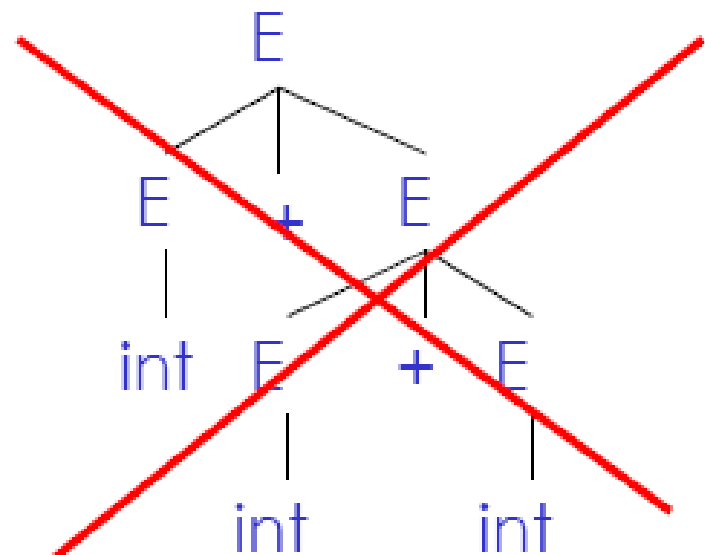
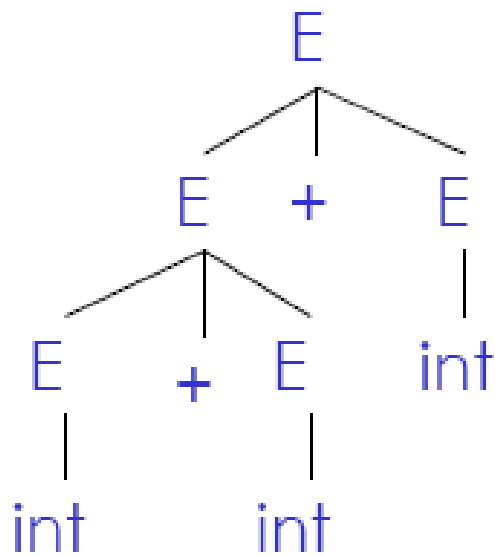
Disambiguation Rules

- Precedence and Associativity Declarations
- %left: all tokens following this declaration are left-associative
- %right: all tokens following this declaration are right-associative
- Precedence is established by the order of the %left and %right declarations
- %left '+' '-'
- %right '*' '/'
 - '*' has a higher precedence than '+', so '1+2*3' would be evaluated as '1+(2*3)'
- %nonassoc: the specified operators may not be used together, e.g., %nonassoc '>' '<'.

Associativity Example

$E \rightarrow E + E \mid \text{int}$

Input int + int + int

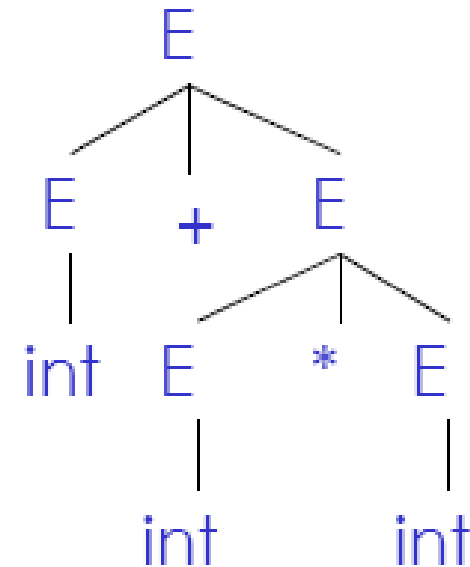
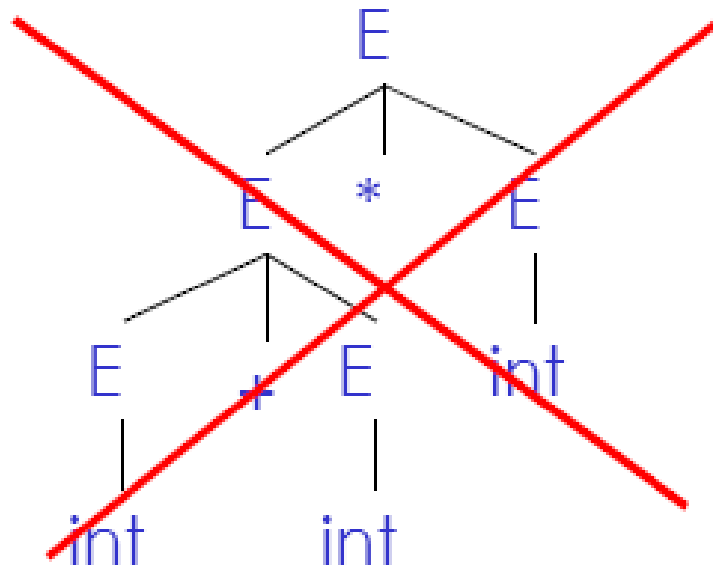


%left +

Precedence Example

$E \rightarrow E + E \mid E * E \mid \text{int}$

Input int + int * int



%left +

% left *

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` \rightarrow `letter` = `right` | `letter`
`letter` \rightarrow `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 $(9 * 2) + 2$

Example

Grammar that defines simple arithmetic expressions

In this grammar the terminal symbols are id + - * ? ()

The non-terminal symbols are expression, term and factor; and expression is the start symbol

expression \rightarrow **expression** + **term**

expression \rightarrow **expression** - **term**

expression \rightarrow **term**

term \rightarrow **term** * **factor**

term \rightarrow **term** / **factor**

term \rightarrow **factor**

factor \rightarrow (**expression**)

factor \rightarrow id

Example

The above grammar can be written concisely as:

$$E \rightarrow E + T \mid E - T \mid T$$
$$E \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid id$$

Derivations Using a Grammar

We apply the productions of a CFG to infer that certain strings are in the language of a certain variable.

There are two approaches to this inference.

The more conventional approach is to use the rules from body to head. That is, we take strings known to be in the language of each of the variables of the body, concatenate them, in the proper order, with any terminals appearing in the body, and infer that the resulting string is in the language of the variable in the head. This procedure is called *Recursive inference*.

Derivations Using a Grammar

- There is another approach to defining the language of a grammar, in which we use the productions from head to body. We expand the start symbol using one of its productions (i.e., using a production whose head is the start symbol). We further expand the resulting string by replacing one of the variables by the body of one of its productions, and so on, until we derive a string consisting entirely of terminals. The language of the grammar is all strings of terminals that we can obtain in this way. This use of grammars is called *derivation*.

Example

Let us explore a more complex CFG that represents (a simplification of) expressions in a typical programming language. First we shall limit ourselves to the operators $+$ and $*$, representing addition and multiplication. We shall allow arguments to be identifiers, but instead of allowing the full set of typical identifiers (letters followed by zero or more letters and digits), we shall allow only the letters a and b and the digits 0 and 1. Every identifier must begin with a or b , which may be followed by any string in $\{a, b, 0, 1\}^*$.

Example

We need two variables in this grammar. One, which we call E , represents expressions. It is the start symbol and represents the language of expressions we are defining. The other variable, I , represents identifiers. Its language is actually regular; it is the language of the regular expression

$$(a \mid b)(a \mid b \mid 0 \mid 1)^*$$

However, we shall not use regular expressions directly in grammars. Rather we use a set of productions that say essentially the same thing as this regular expression.

Example

A context-free grammar for simple expressions

- 1. $E \rightarrow I$**
- 2. $E \rightarrow E + E$**
- 3. $E \rightarrow E * E$**
- 4. $E \rightarrow (E)$**

- 5. $I \rightarrow a$**
- 6. $I \rightarrow b$**
- 7. $I \rightarrow Ia$**
- 8. $I \rightarrow Ib$**
- 9. $I \rightarrow I0$**
- 10. $I \rightarrow I1$**

Example

The grammar for expressions is stated formally as $G = (\{E, I\}, T, P, E)$, where T is the set of symbols $\{+, *, (,), a, b, 0, 1\}$ and P is the set of productions shown above.

Tree Traversal

- ***Tree traversals*** are used for describing attribute evaluation and for specifying the execution of code fragments in a translation scheme.
- A traversal of a tree starts at the root and visits each node of the tree in some order.

Depth-First Traversal

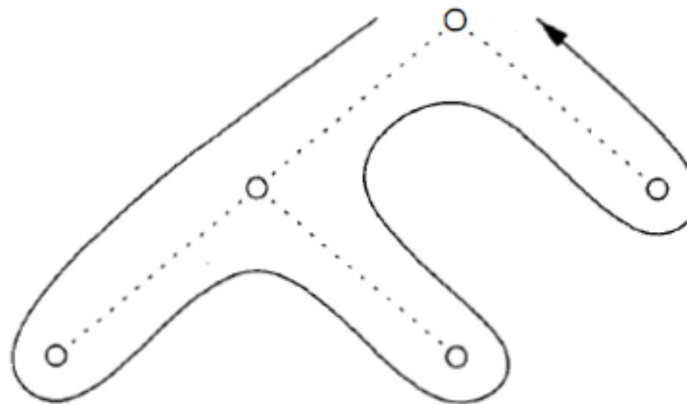
- ***Depth-first traversal*** starts at the root and recursively visits the children of each node in any order, not necessarily from left to right . It is called "***depth-first***" because it visits an unvisited child of a node whenever it can, so it visits nodes as far away from the root (as "***deep***") as quickly as it can.

Depth-First Traversal

- The procedure ***visit(N)*** in Fig. is a depth-first traversal that visits the children of a node in left-to-right order, as shown in Fig.

Depth-First Traversal

```
procedure visit(node  $N$ ) {  
    for ( each child  $C$  of  $N$ , from left to right ) {  
        visit( $C$ );  
    }  
    evaluate semantic rules at node  $N$ ;  
}
```



Depth-First Traversal

- Synthesized attributes can be evaluated during any bottom-up traversal, that is, a traversal that evaluates attributes at a node after having evaluated attributes at its children.
- In general, with both synthesized and inherited attributes, the matter of evaluation order is quite complex

Questions

- CFG representing the Regular Expression a^+

$$A \rightarrow aA$$

- CFG representing the Regular Expression b^*

$$B \rightarrow$$

$$B \rightarrow bB$$

Questions

- CFG representing the Regular Expression a^*b^+
(i.e. start with any number of a's followed by non-zero numbers of b)

$$S \rightarrow R \mid aS$$

$$R \rightarrow b \mid bR$$

- 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)

$$S \rightarrow aRa$$

$$R \rightarrow b \mid bR$$

Questions

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

$(a|b)^*abb$ where $\Sigma = \{a, b\}$

Create an equivalent CFG of the above regular expression

$$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$$

$$A_1 \rightarrow bA_2$$

$$A_2 \rightarrow bA_3$$

$$A_3 \rightarrow \epsilon$$