

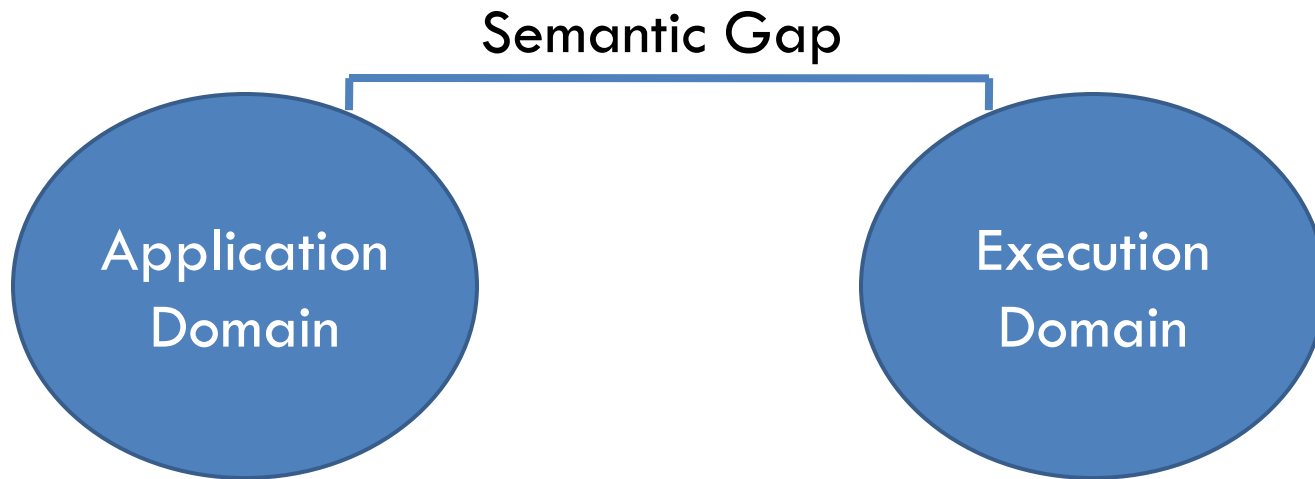
# LANGUAGE PROCESSORS

Presented By: Prof. Prakash Patel, IT Dept. GIT

# Introduction

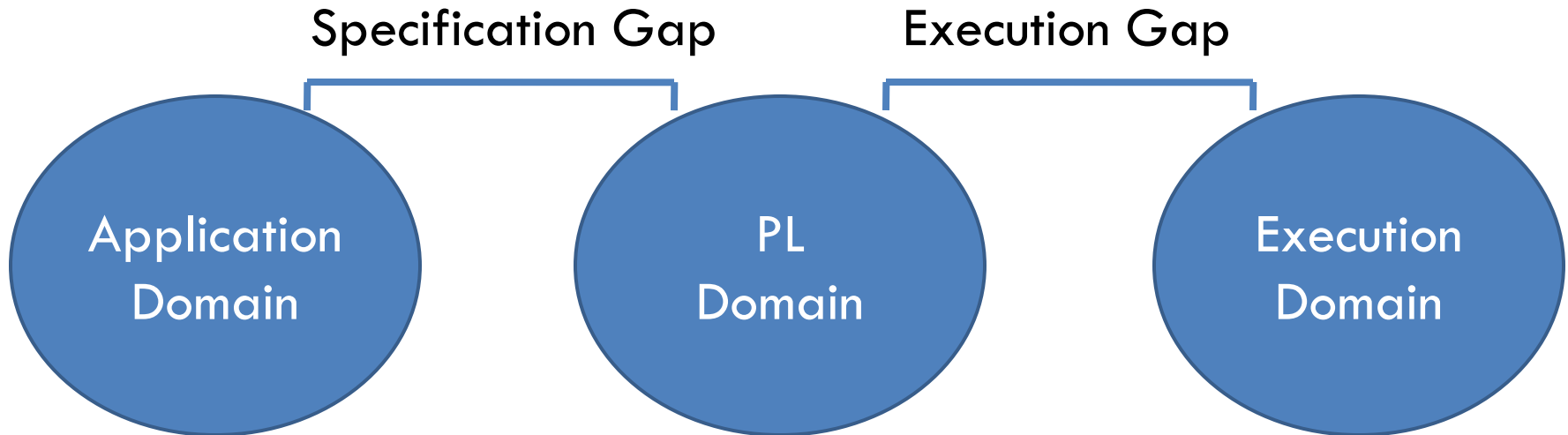
- Language Processing activities arise due to the differences between the manner in which a software designer describes ***the ideas concerning the behavior of a software*** and the manner in which ***these ideas are implemented in a computer system***.
- The designer expresses the ideas in terms related to the ***application domain*** of the software. To implement these ideas, their description has to be interpreted in terms related to the ***execution domain***.

# Semantic Gap



- ❑ Semantic Gap has many consequences
  - ▣ Large development time
  - ▣ Large development effort
  - ▣ Poor quality of software

# Specification and Execution Gaps



- ❑ The software engineering steps aimed at the use of a PL can be grouped into
  - ▣ Specification, design and coding steps
  - ▣ PL implementation steps

# Specification and Execution Gaps

- Specification Gap

- ▣ It is the semantic gap between two specifications of the same task.

- Execution Gap

- ▣ It is the gap between the semantics of programs (that perform the same task) written in different programming languages.

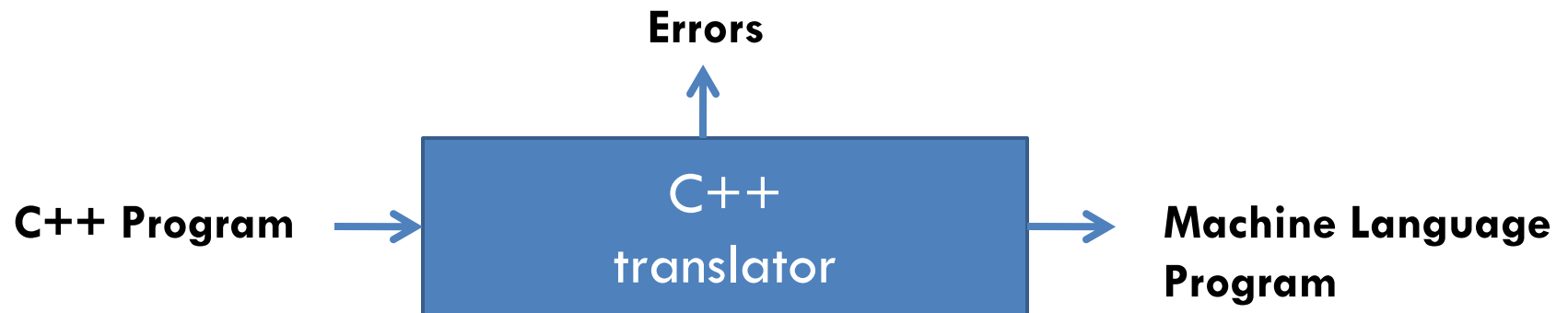
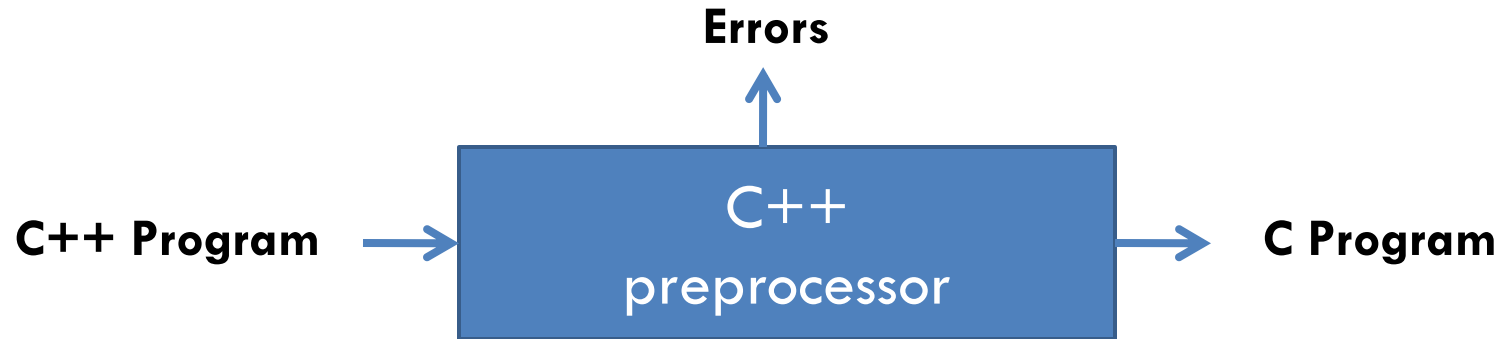
# Language Processors

- “A *language processor* is a software which bridges a *specification or execution gap*”.
- The program form input to a language processor as the *source program* and to its output as the *target program*.
- The languages in which these programs are written are called *source language* and *target language*, respectively.

# Types of Language Processors

- A **language translator** bridges an execution gap to the machine language (or assembly language) of a computer system. E.g. Assembler, Compiler.
- A **detranslator** bridges the same execution gap as the language translator, but in the reverse direction.
- A **preprocessor** is a language processor which bridges an execution gap but is not a language translator.
- A **language migrator** bridges the specification gap between two PLs.

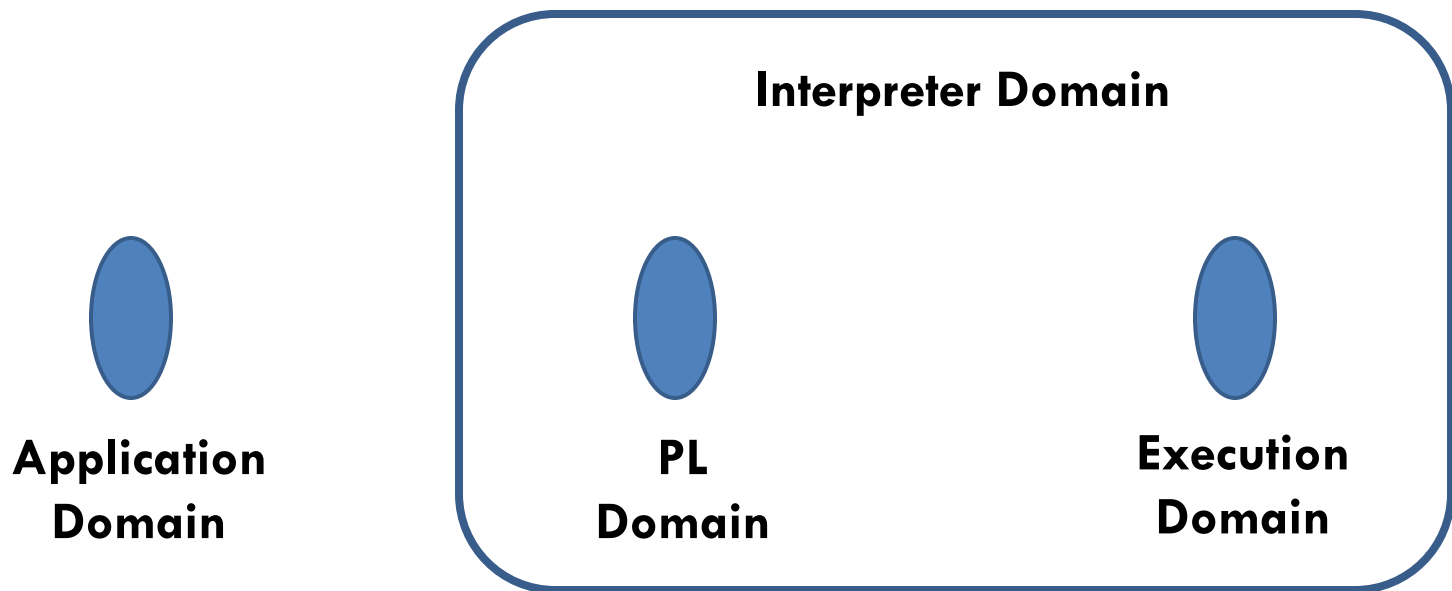
# Language Processors - Examples





# Interpreters

- An **interpreter** is a language processor which bridges an execution gap without generating a machine language program.
- An interpreter is a language translator according to classification.

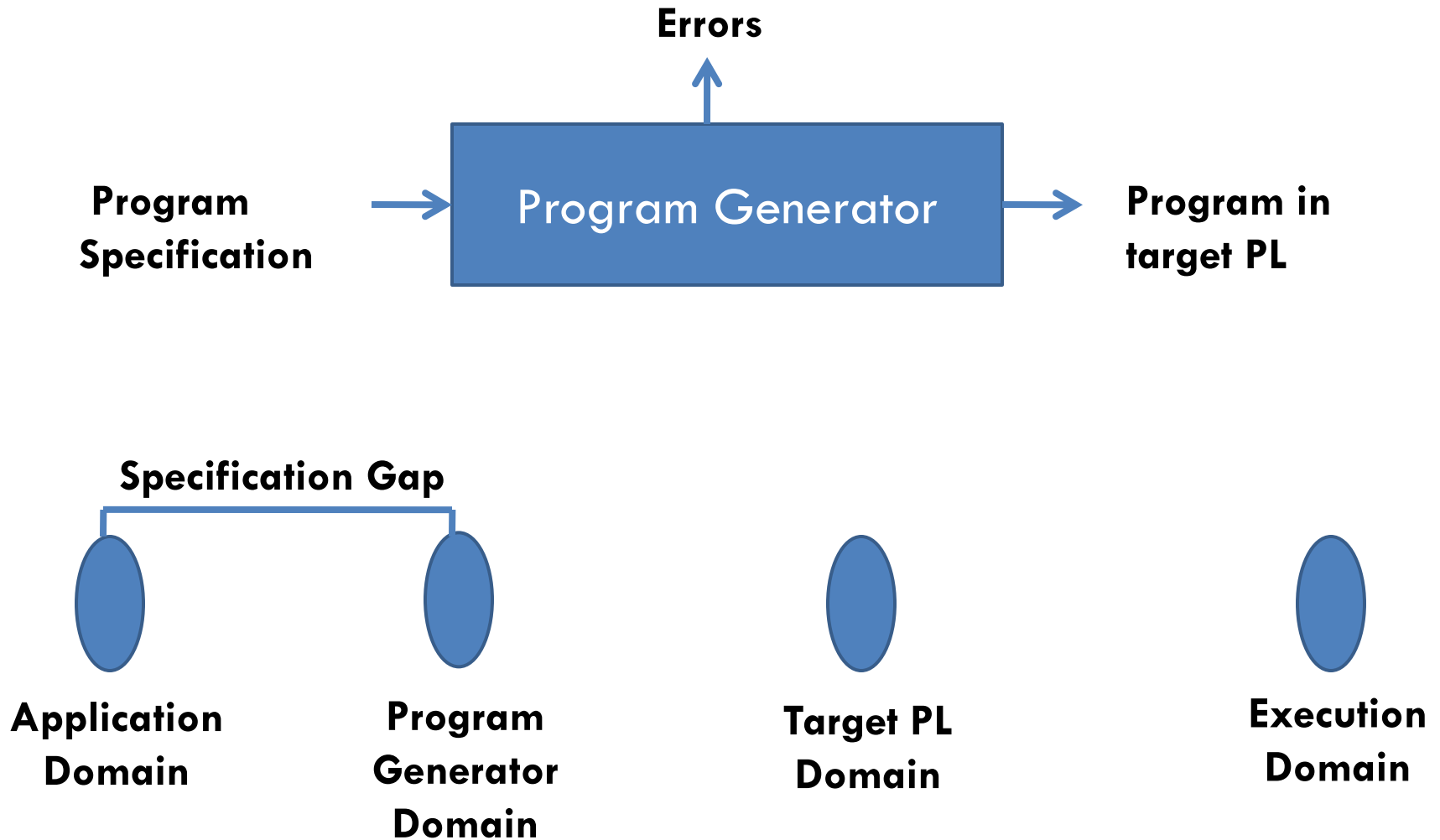


# Language Processing Activities

---

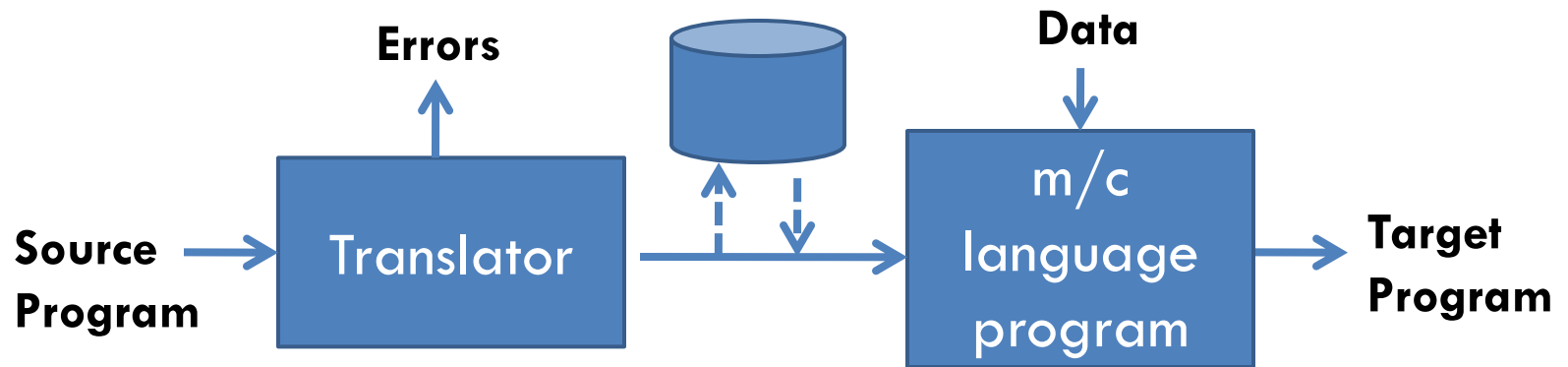
- Program Generation Activities
- Program Execution Activities

# Program Generation



# Program Execution

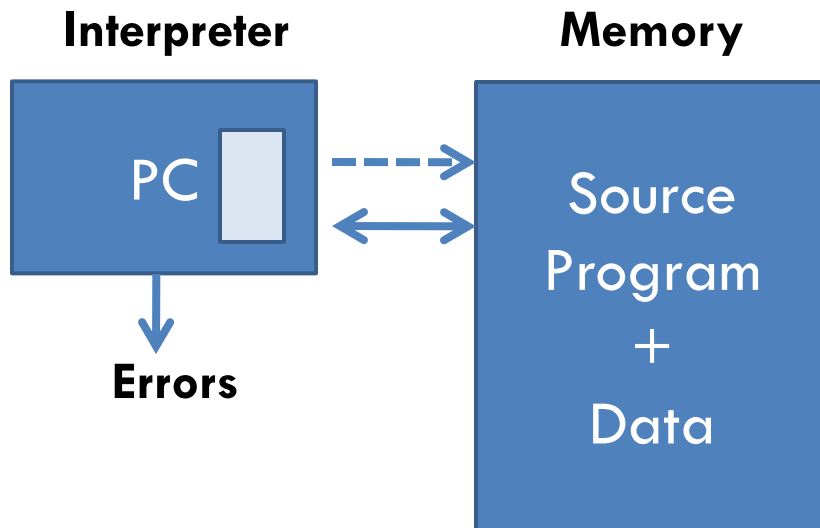
- Two popular models for program execution are translation and interpretation.
- **Program translation**



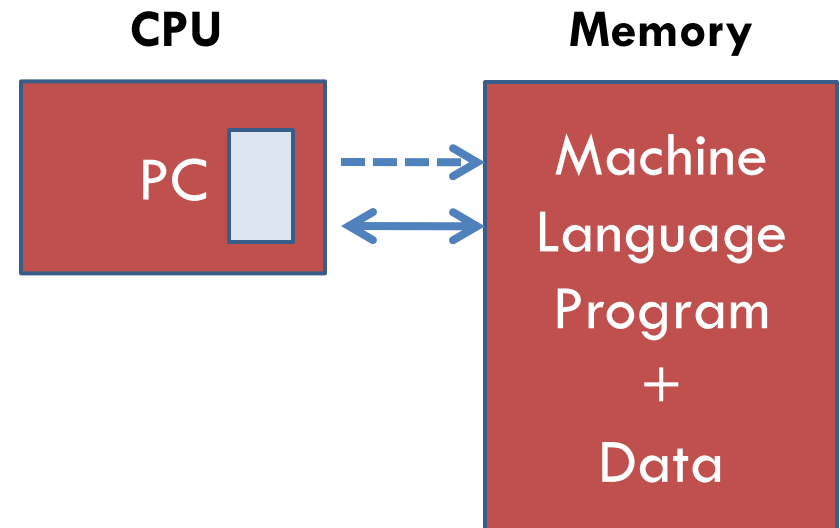
- A program must be translated before it can be executed.
- The translated program may be saved in a file. The saved program may be executed repeatedly.
- A program must be retranslated following modifications.

# Program Execution

## □ Program interpretation



**Interpretation**



**Program execution**

# Fundamentals of Language Processing

Language Processing = *Analysis of SP + Synthesis of TP*

*Collection of LP components engaged in analysis a source program as the analysis phase and components engaged in synthesizing a target program constitute the synthesis phase.*

# Analysis Phase

- The specification consists of three components:
  - ▣ **Lexical rules** which govern the formation of valid lexical units in the source language.
  - ▣ **Syntax rules** which govern the formation of valid statements in the source language.
  - ▣ **Semantic rules** which associate meaning with valid statements of the language.

- Consider the following example:

**percent\_profit = (profit \* 100) / cost\_price;**

**Lexical units** identifies =, \* and / operators, 100 as constant, and the remaining strings as identifiers.

**Syntax analysis** identifies the statement as an assignment statement with percent\_profit as the left hand side and (profit \* 100) / cost\_price as the expression on the right hand side.

**Semantic analysis** determines the meaning of the statement to be the assignment of profit X 100 / cost\_price to percent\_profit.

# Synthesis Phase

- The synthesis phase is concerned with the construction of target language statements which have the same meaning as a source statement.
- It performs two main activities:
  - ▣ Creation of data structures in the target program (**memory allocation**)
  - ▣ Generation of target code (**code generation**)
- Example

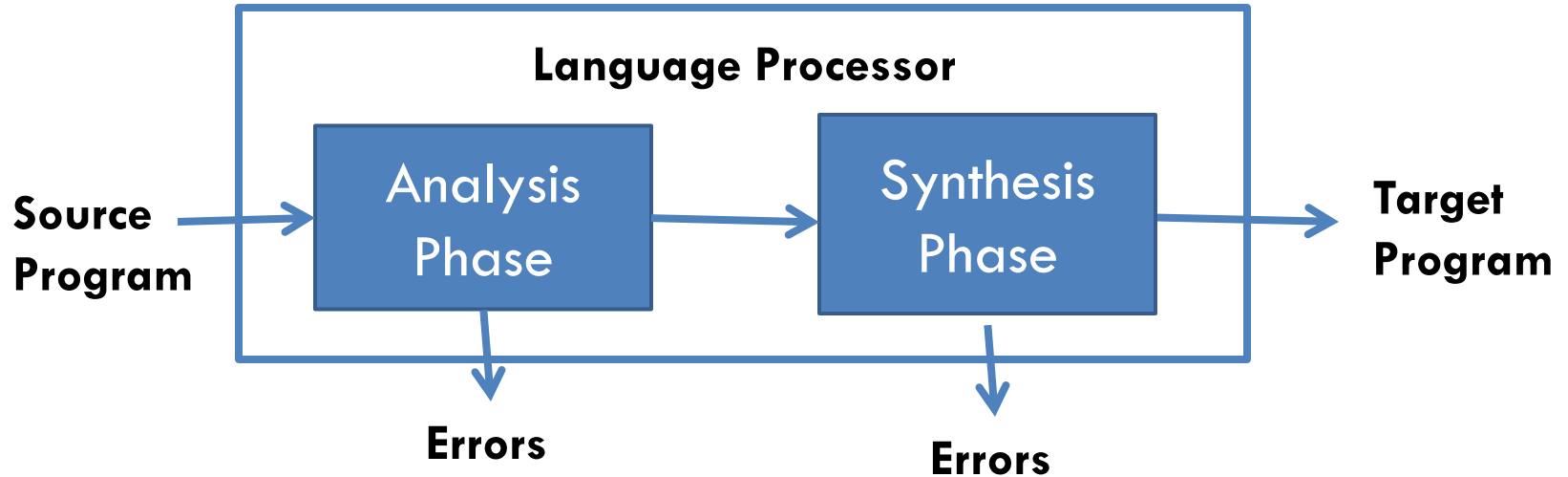
```
MOVER  AREG, PROFIT
MULT   AREG, 100
DIV    AREG, COST_PRICE
MOVEM  AREG, PERCENT_PROFIT
```

...

PERCENT_PROFIT	DW	1
PROFIT	DW	1
COST_PRICE	DW	1



# Phases and Passes of LP



- Analysis of source statements can not be immediately followed by synthesis of equivalent target statements due to following reasons:
  - ▣ Forward References
  - ▣ Issues concerning memory requirements and organization of a LP

# Lexical Analysis (Scanning)

- It identifies the lexical units in a source statements. It then classifies the units into different lexical classes, e.g. id's, constants, reserved id's, etc. and enters them into different tables.
- It builds a descriptor, called **token**, for each lexical unit. A token contains two fields – *class code* and *number in class*.
- class code identifies the class to which a lexical unit belongs. number in class is the entry number of the lexical unit in the relevant table.
- We depict a token as Code # no, e.g. Id # 10

# Lexical Analysis (Scanning) - Example

`i : integer;`  
`a, b : real;`  
`a := b + i;`

	Symbol	Type	Length	Address
1	<b>i</b>	<b>int</b>		
2	<b>a</b>	<b>real</b>		
3	<b>b</b>	<b>real</b>		
4	<b>i *</b>	<b>real</b>		
5	<b>temp</b>	<b>real</b>		

Note that int i first needed to be converted into real, that is why 4<sup>th</sup> entry is added into the table.

Addition of entry 3 and 4, gives entry 5 (temp), which is value  $b + (i *)$ .

The statement `a := b+i;` is represented as the string of tokens

**Id#2**

**Op#5**

**Id#3**

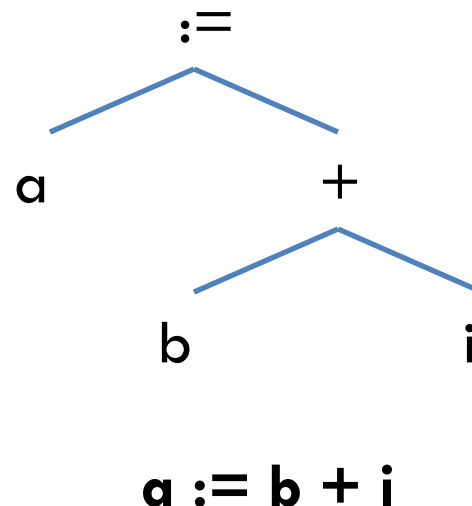
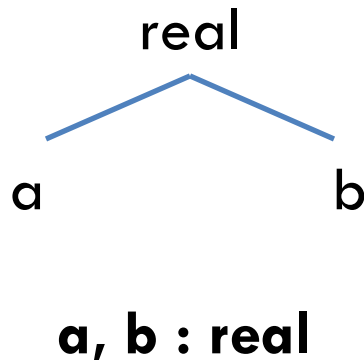
**Op#3**

**Id#1**

**Op#10**

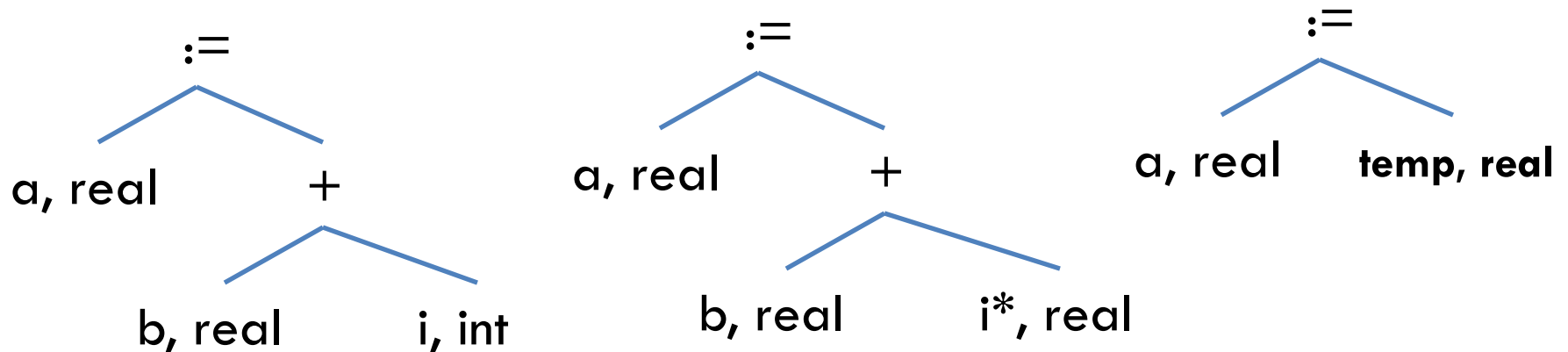
# Syntax Analysis (Parsing)

- It processes the string of tokens built by lexical analysis to determine the **statement class**, e.g. assignment statement, if statement etc.
- It then builds an IC which represents the structure of a statement. The IC is passed to semantic analysis to determine the meaning of the statement.

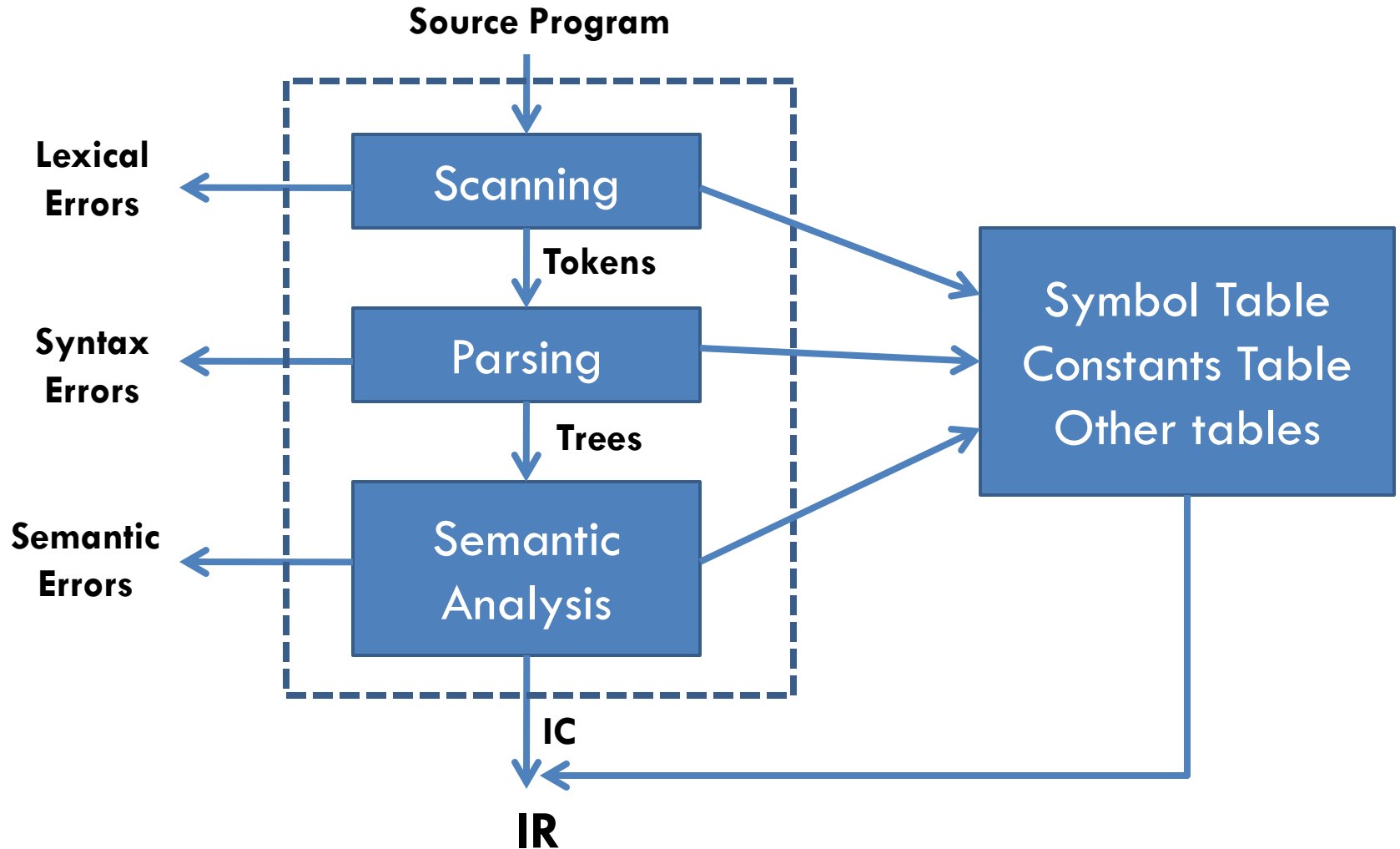


# Semantic Analysis

- It identifies the sequence of actions necessary to implement the meaning of a source statement.
- It determines the meaning of a sub tree in the IC, it adds information to a table or adds an action to the sequence of actions. The analysis ends when the tree has been completely processed.



# Analysis Phase (Front end)



# Synthesis Phase (Back end)

- It performs memory allocation and code generation.

- **Memory Allocation**

- ▣ The memory requirement of an identifier is computed from its type, length and dimensionality and memory is allocated to it.
- ▣ The address of the memory area is entered in the symbol table.

	Symbol	Type	Length	Address
1	i	int		2000
2	a	real		2001
3	b	Real		2002

# Synthesis Phase (Back end)

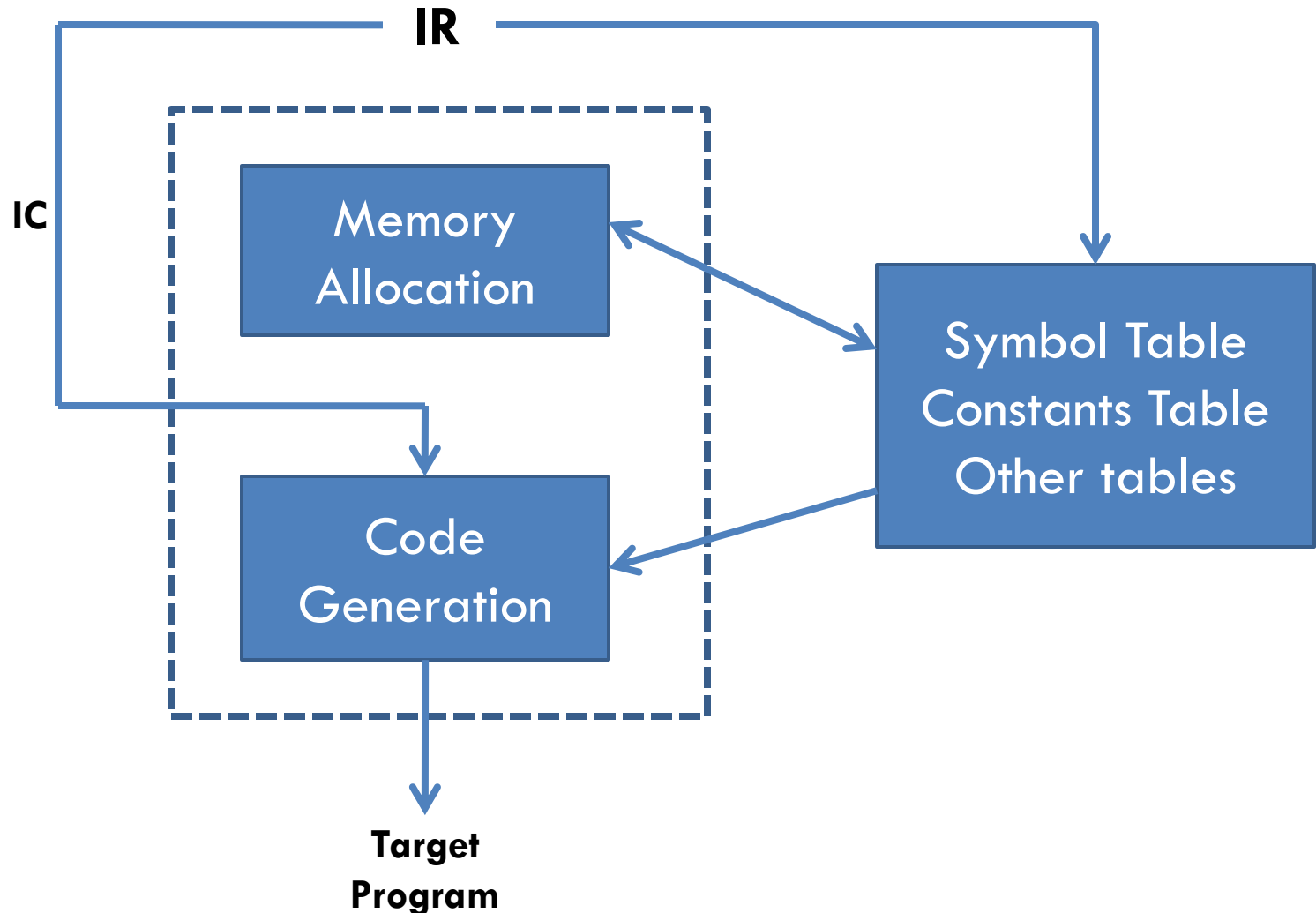
## □ Code Generation

- ▣ It uses knowledge of the target architecture, viz. knowledge of instructions and addressing modes in the target computer, to select the appropriate instructions.
- ▣ The synthesis phase may decide to hold the values of  $i^*$  and temp in machine registers and may generate the assembly code.
- ▣  $a := b + i;$

CONV_R	AREG, I
ADD_R	AREG, B
MOVEM	AREG, A



# Synthesis Phase (Back end)



# Fundamentals of Language Specification

## □ PL Grammars

- The lexical and syntactic features of a programming language are specified by its grammar.
- A language  $L$  can be considered to be a collection of valid sentences.
- Each sentence can be looked upon as a sequence of words, and each word as a sequence of letters or graphic symbols acceptable in  $L$ .
- A language specified in this manner is known as a formal language.

# Alphabet

- The alphabet of  $L$ , denoted by the Greek symbol  $\Sigma$  is the collection of symbols in its character set.
- We use lower case letters  $a, b, c$ , etc. to denote symbols in  $\Sigma$
- A symbol in the alphabet is known as a **terminal symbol** ( $T$ ) of  $L$ .
- The alphabet can be represented using mathematical notation of a set, e.g.

$$\Sigma = \{ a, b, \dots, z, 0, 1, \dots, 9 \}$$

where  $\{, ,, \}$  are called *meta symbols*.

# String

- A string is a finite sequence of symbols.
- We represent strings by Greek symbols  $\alpha$ ,  $\beta$ ,  $\gamma$ , etc.  
Thus  $\alpha = axy$  is a string over  $\Sigma$
- The length of a string is the number of symbols in it.
- Absence of any symbol is also a string, null string  $\epsilon$ .
- Example

$$\alpha = ab, \beta = axy$$

$$\alpha\beta = \alpha.\beta = abaxy \text{ [concatenation]}$$

# Nonterminal symbols

- A Nonterminal symbol (NT) is the name of a syntax category of a language, e.g. noun, verb, etc.
- An NT is written as a single capital letter, or as a name enclosed between  $\langle \dots \rangle$ , e.g. A or  $\langle \text{Noun} \rangle$ .
- It is a set of symbols not in  $\Sigma$  that represents intermediate states in the generation process.

# Productions

- A production, also called a rewriting rule, is a rule of the grammar.
- It has the form

A nonterminal symbol ::= String of Ts and NTs



L.H.S.



R.H.S.

e.g.  $\langle \text{article} \rangle ::= a \mid an \mid the$

$\langle \text{Noun} \rangle ::= boy \mid apple$

$\langle \text{Noun Phrase} \rangle ::= \langle \text{article} \rangle \langle \text{Noun} \rangle$

# Derivation, Reduction and Parse Trees

- A grammar  $G$  is used for two purposes, to generate valid strings of  $L_G$  and to 'recognize' valid strings of  $L_G$ .
- The derivation operation helps to generate valid strings while the reduction operation helps to recognize valid strings.
- A parse tree is used to depict the syntactic structure of a valid string as it emerges during a sequence of derivations or reductions.

# Derivation

- Let production  $P_1$  of grammar  $G$  be of the form

$$P_1 : A ::= \alpha$$

and let  $\beta$  be a string such that  $\beta = \gamma A \theta$ , then replacement of  $A$  by  $\alpha$  in string  $\beta$  constitutes a derivation according to production  $P_1$ .

- Example

$\langle \text{Sentence} \rangle ::= \langle \text{Noun Phrase} \rangle \langle \text{Verb Phrase} \rangle$

$\langle \text{Noun Phrase} \rangle ::= \langle \text{Article} \rangle \langle \text{Noun} \rangle$

$\langle \text{Verb Phrase} \rangle ::= \langle \text{Verb} \rangle \langle \text{Noun Phrase} \rangle$

$\langle \text{Article} \rangle ::= a \mid an \mid the$

$\langle \text{Noun} \rangle ::= boy \mid apple$


$\langle \text{Verb} \rangle ::= ate$



# Derivation

- The following strings are sentential forms of LG.

<Noun Phrase> <Verb Phrase>  
the boy <Verb Phrase>  
<Noun Phrase> ate <Noun Phrase>  
the boy ate <Noun Phrase>



sentential  
forms

the boy ate an apple ← sentence

# Reduction

Let production  $P_1$  of grammar  $G$  be of the form

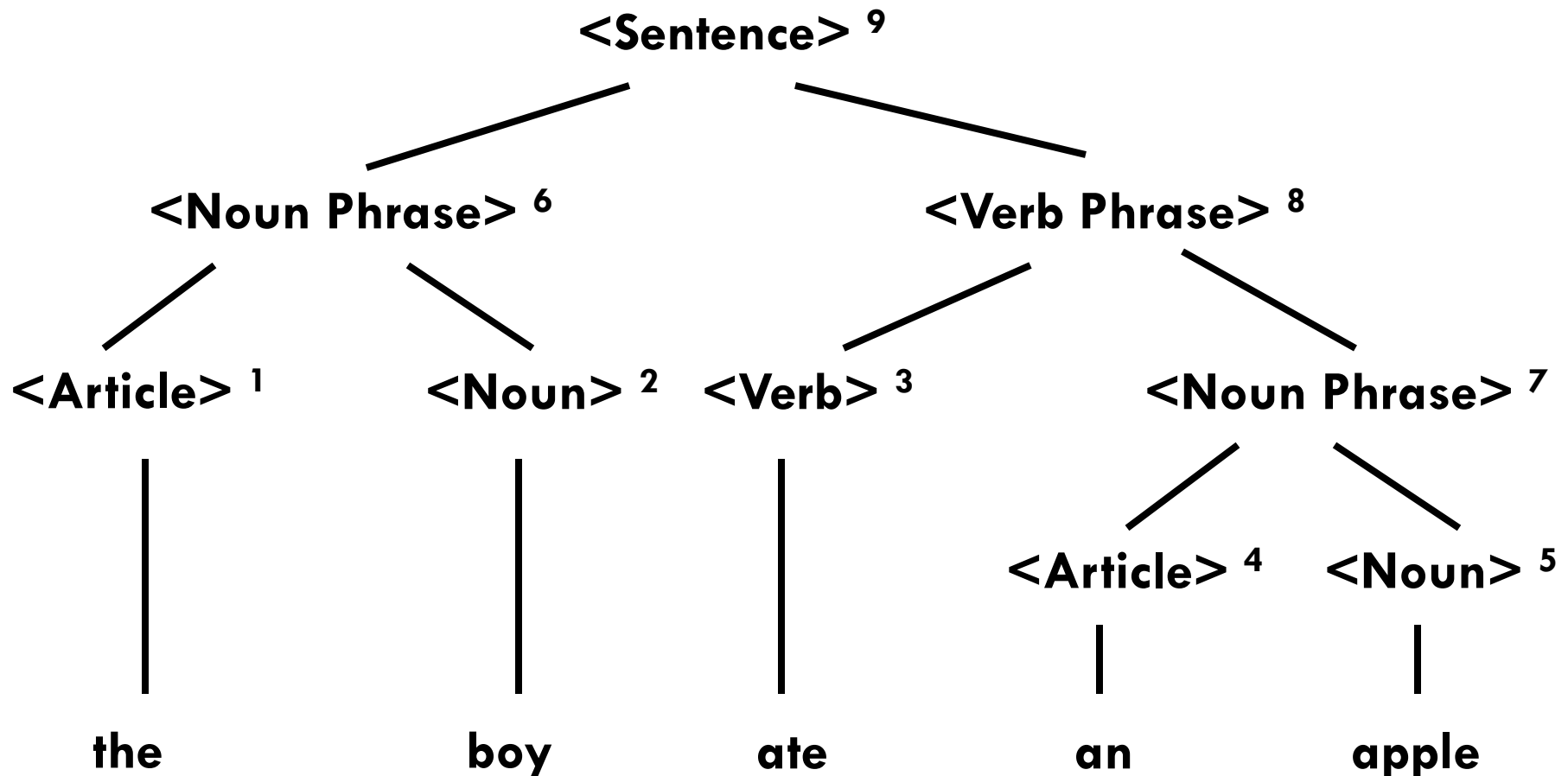
$$P_1 : A ::= \alpha$$

and let  $\sigma$  be a string such that  $\sigma = \gamma A \theta$ , then replacement of  $\alpha$  by  $A$  in string  $\sigma$  constitutes a reduction according to production  $P_1$ .

Step	String
0	<b>the boy ate an apple</b>
1	<b>&lt;Article&gt; boy ate an apple</b>
2	<b>&lt;Article&gt; &lt;Noun&gt; ate an apple</b>
3	<b>&lt;Article&gt; &lt;Noun&gt; &lt;Verb&gt; an apple</b>
4	<b>&lt;Article&gt; &lt;Noun&gt; &lt;Verb&gt; &lt;Article&gt; apple</b>
5	<b>&lt;Article&gt; &lt;Noun&gt; &lt;Verb&gt; &lt;Article&gt; &lt;Noun&gt;</b>
6	<b>&lt;Noun Phrase&gt; &lt;Verb&gt; &lt;Article&gt; &lt;Noun&gt;</b>
7	<b>&lt;Noun Phrase&gt; &lt;Verb&gt; &lt;Noun Phrase&gt;</b>
8	<b>&lt;Noun Phrase&gt; &lt;Verb Phrase&gt;</b>
9	<b>&lt;Sentence&gt;</b>

# Parse Trees

- A sequence of derivations or reductions reveals the syntactic structure of a string with respect to  $G$ , in the form of a parse tree.



# Classification of Grammars

- Type-0 grammar (Phrase Structure Grammar)  
 $\alpha ::= \beta$ , where both can be strings of Ts and NTs.  
But it is not relevant to specification of Prog. Lang.
- Type-1 grammar (Context Sensitive Grammar)  
 $\alpha A \beta ::= \alpha \pi \beta$ ,  
But it is not relevant to specification of Prog. Lang.
- **Type-2 grammar (Context Free Grammar)**  
 $A ::= \pi$ , which can be applied independent of its context.  
CFGs are ideally suited for PL specifications.
- Type-3 grammar (Linear or Regular Grammar)  
 $A ::= t B \mid t \text{ OR } A ::= B t \mid t$   
Nesting of constructs or matching of parentheses cannot be specified using such productions.

# Ambiguity in Grammatical Specification

- It implies the possibility of different interpretation of a source string.
- Existence of ambiguity at the level of the syntactic structure of a string would mean that more than one parse tree can be built for the string. So string can have more than one meaning associated with it.

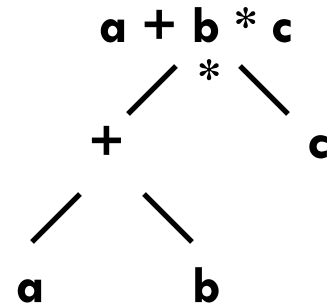
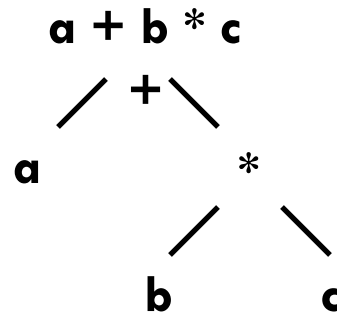
## Ambiguous Grammar

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

$\text{Id} \rightarrow a \mid b \mid c$

Assume source

string is  $a + b * c$



# Eliminating Ambiguity – An Example

## Unambiguous Grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow F \wedge P \mid P$

$P \rightarrow \text{id}$

$\text{id} \rightarrow a \mid b \mid c$

$a + b * c$

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

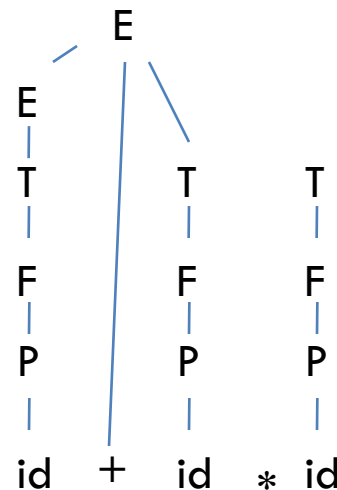
$\Rightarrow P + P * P$

$\Rightarrow F + P * P$

$\Rightarrow T + F * F$

$\Rightarrow E + T * T$

$\Rightarrow E * T$  (?? Ambiguous)



$a + b * c$

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

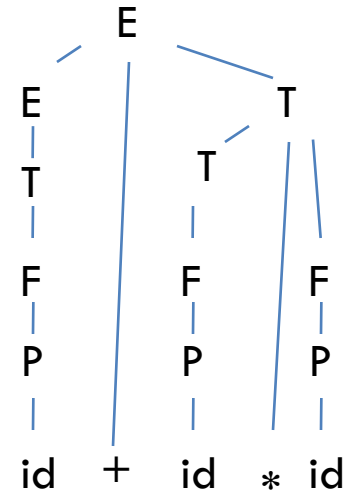
$\Rightarrow P + P * P$

$\Rightarrow F + F * P$

$\Rightarrow T + \underline{T * F}$

$\Rightarrow E + T$

$\Rightarrow E$  (Unambiguous)



# GTU Examples

- List out the unambiguous production rules (grammar) for arithmetic expression containing  $+$ ,  $-$ ,  $*$ ,  $/$  and  $^$  (exponent).

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow F ^ P \mid P$$

$$P \rightarrow (E) \mid \text{<id>}$$

Derive string  $\text{<id>} - \text{<id>} * \text{<id>} ^ \text{<id>} + \text{<id>}$

$$E \rightarrow E + T$$

**→ E - T + T**

→ T - T + T

**→ F - T + T**

**→ P - T + T**

→  $\langle id \rangle - T + T$

→ **<id> - T \* F + T**

**→ <id> - F \* F + T**

→ **<id> - P \* F + T**

**→  $\langle id \rangle - \langle id \rangle * F + T$**

→  $\langle id \rangle - \langle id \rangle * F^P + T$

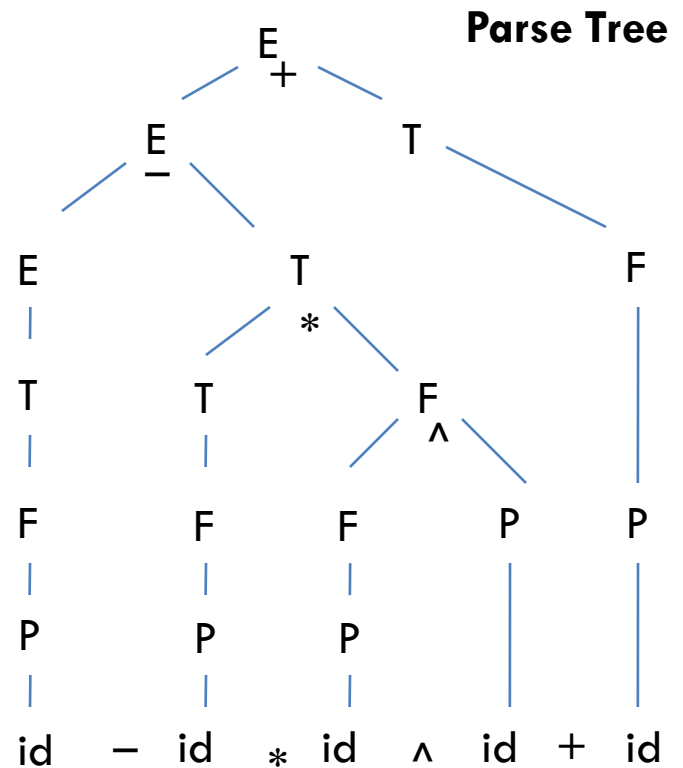
$$\rightarrow \langle id \rangle - \langle id \rangle * P^P + T$$
$$\rightarrow \langle id \rangle - \langle id \rangle * \langle id \rangle^p + T$$

→  $\langle id \rangle - \langle id \rangle * \langle id \rangle ^ \langle id \rangle + T$

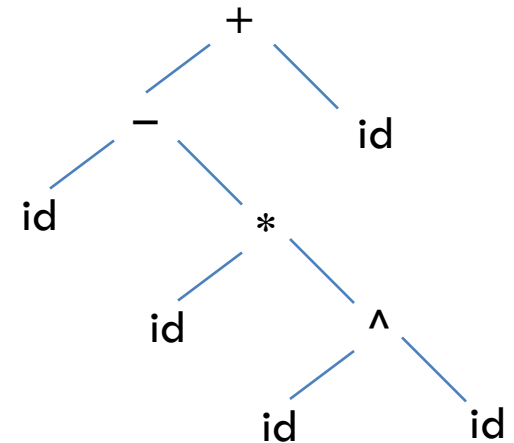
→  $\langle id \rangle - \langle id \rangle * \langle id \rangle ^ \langle id \rangle + F$

→  $\langle id \rangle - \langle id \rangle * \langle id \rangle ^ \langle id \rangle + P$

→  $\langle id \rangle - \langle id \rangle * \langle id \rangle ^ \langle id \rangle + \langle id \rangle$



## Abstract Syntax Tree





# Another Example

- Consider the following grammar:

$$S \rightarrow a S b S \mid b S a S \mid \varepsilon$$

Derive the string ***abab***. Draw corresponding parse tree. Are these rules ambiguous ? Justify.