# System Programming

Dr. Wassim Ahmad

# Book:

- 1.Computer Systems_ A Programmer's Perspective 3rd Edition
- By: Brayant O'Hallaron

- 2. System Programming with C and Unix
- By: Adam Hoover

- 3. System Programming
- By: D. M. Dhamdhere

# System Software

- **Computer software**, or simply **software**, refers to the non-tangible components of computers, known as computer programs. The term is used to contrast with computer hardware, which denotes the physical tangible components of computers.

# Software classification

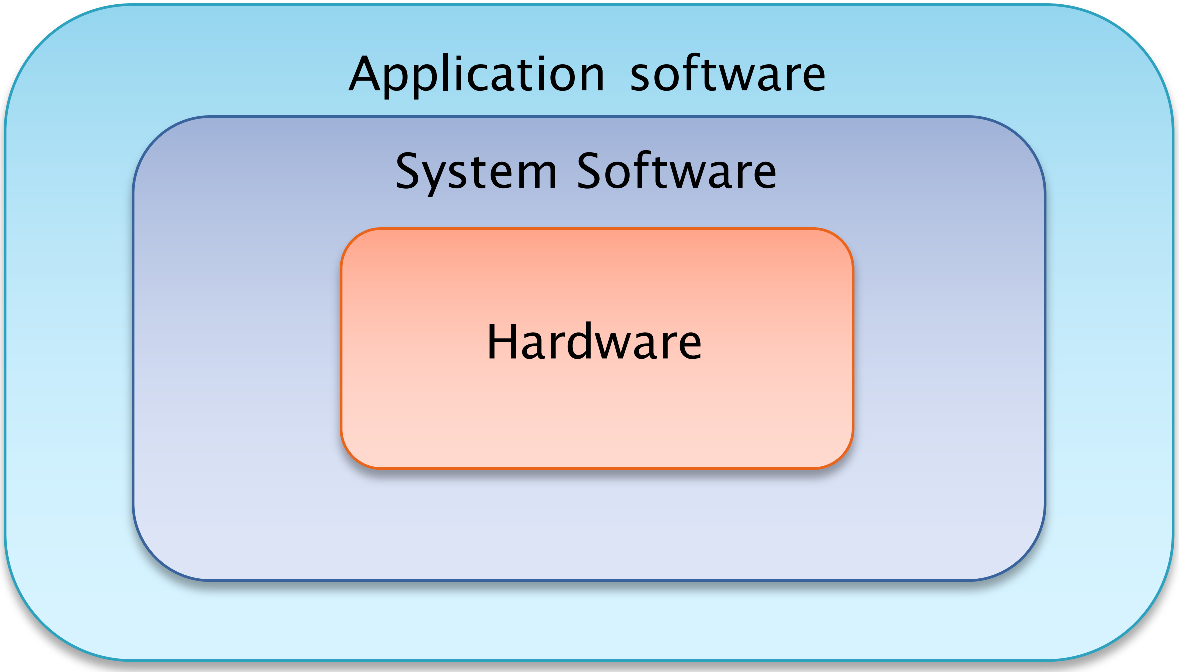
- Software can be classified into
  - System software:
  - **System software** (or **systems software**) is computer software designed to operate and control the computer hardware and to provide a platform for running application software.

    - System software is collection of software program that perform a variety of functions like IO management, storage management, generation and execution of programs etc.
      - Operating Systems
      - Compiler / Assembler (utility software)
      - Device Drivers

  - Application software:
    - Application software is kind of software which is designed for fulfillment specialized user requirement.
      - MS Office
      - Adobe Photoshop

# Cont.

- The system software work as middleware between application software and hardware.

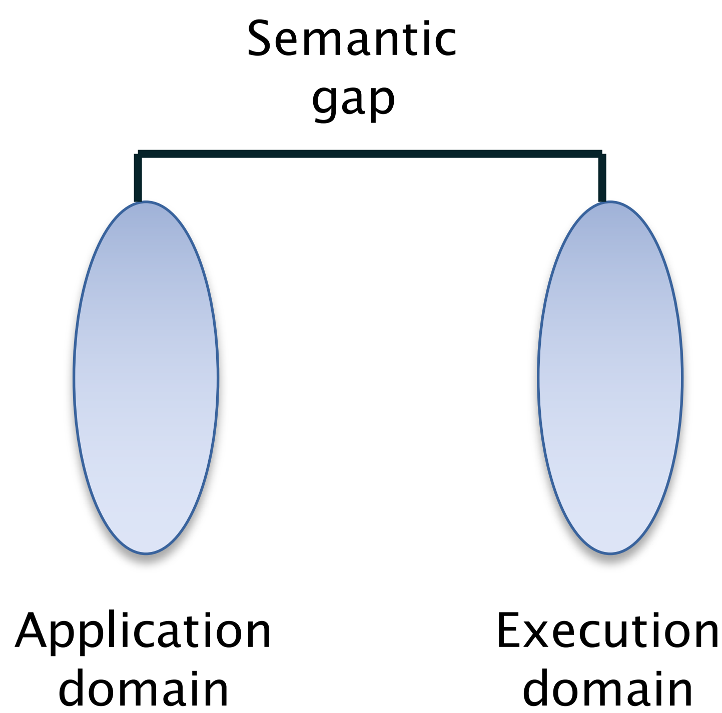


# Chapter– 1
# Language Processors

# System Software

- Language processors (Why?)
  - Language processing activities arise due to the differences between the manner in which a software designer describes the ideas concerning the behavior of software and the manner in which these ideas are implemented in computer system.

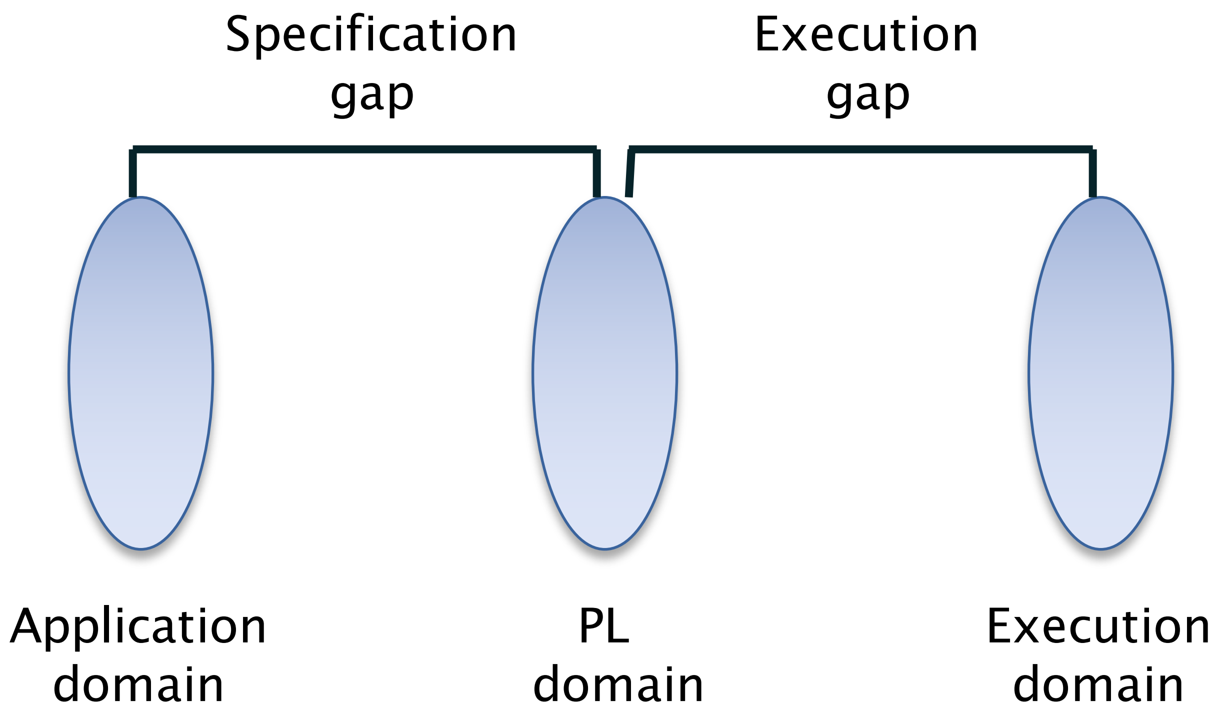
  - The designer expresses the ideas in terms related to the application domain of the software.
  - To implement these ideas, their description has to be interpreted in terms related to the execution domain.




- The term semantics to represent the rules of meaning of a domain, and the term semantic gap to represent difference between the semantics of two domains.

- The semantic gap has many consequences, some of the important are
  - Large development times
  - Large development effort
  - Poor quality software.

- these issues are tackled by software engneering thru’ use of methodologies and programming languages.

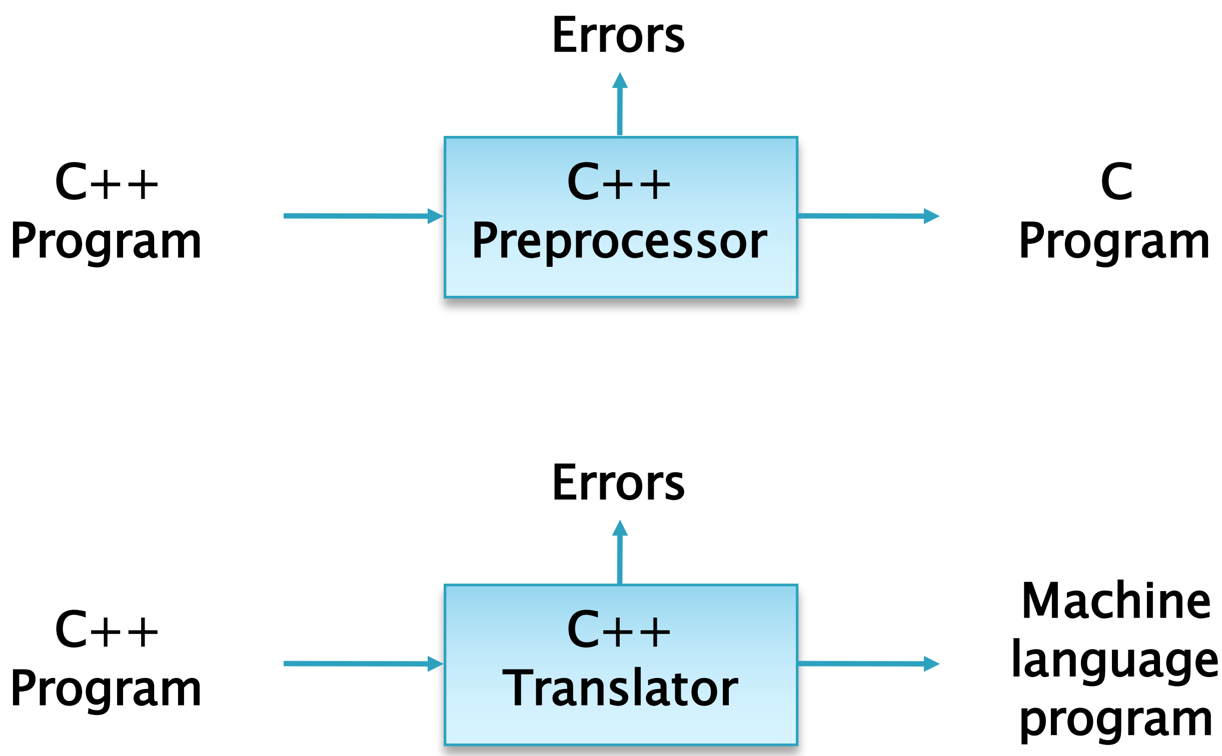



- s/w development team
- Programing language processor

# Cont.

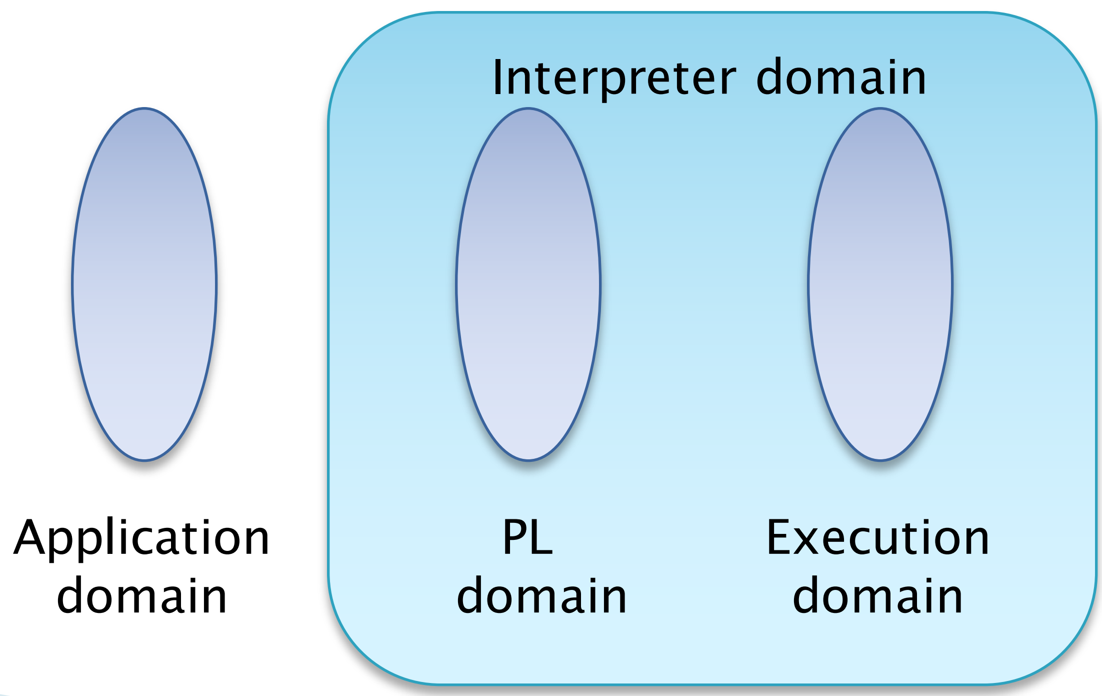
- Language processor: A language processor is software which bridge a specification or execution gap.
  - A Language Translator
  - De-translator
  - Preprocessor
  - Language migrator

# Example


Errors
C++
Program
C++
Preprocessor
C
Program
Errors
C++
Program
C++
Translator
Machine
language
program

# Interpreter

- An interpreter is language processor which bridges an execution gap without generating a machine language program that means the execution gap vanishes totally.

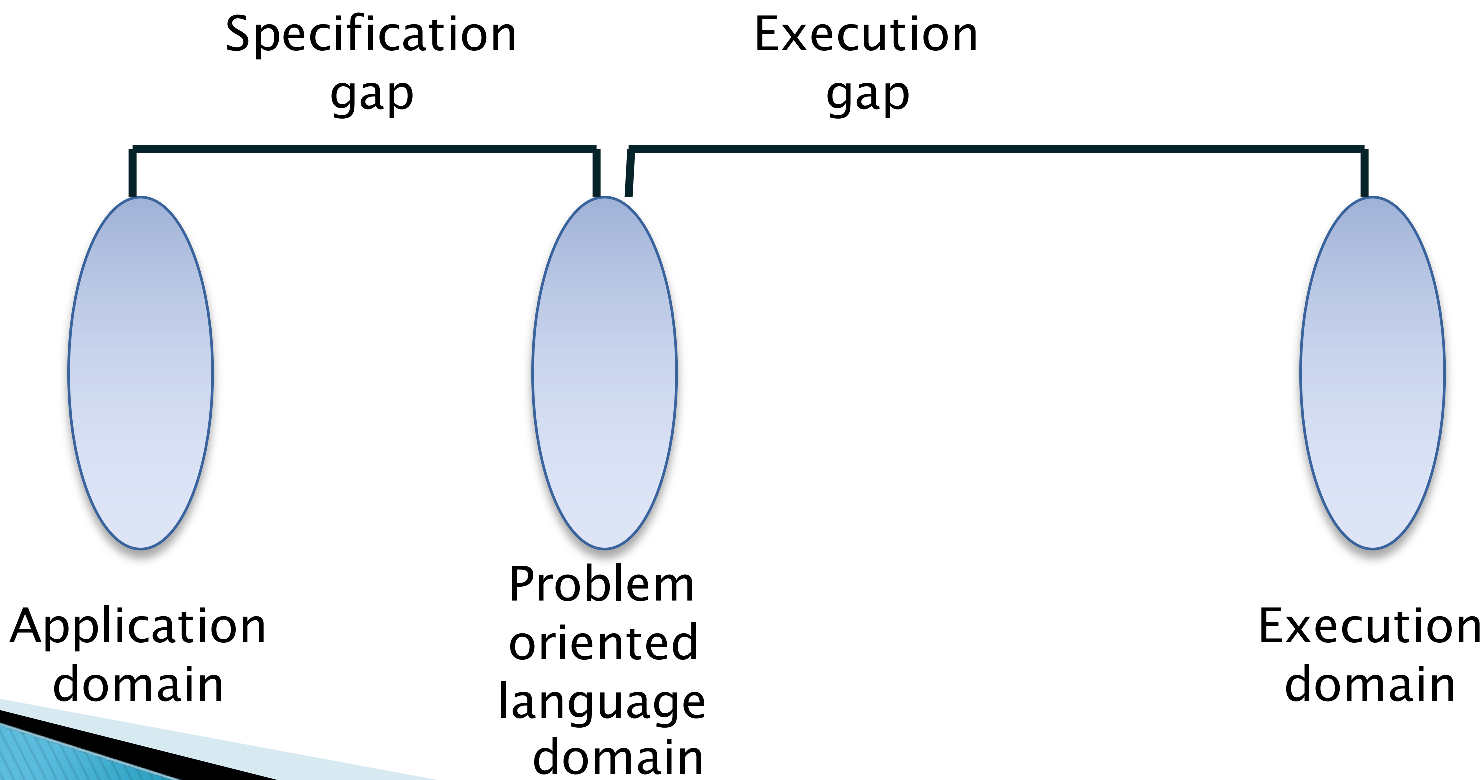


# Problem oriented language

- Three consequences of the semantic gap are in fact the consequences of specification gap.
- A classical solution is to develop a PL such that the PL domain is very close or identical to the application domain.
- Such PLs can only used for specific applications, they are problem oriented languages.

# Procedure oriented language

- A procedure oriented language provides general purpose facilities required in most application domains.



# Language Processing Activities

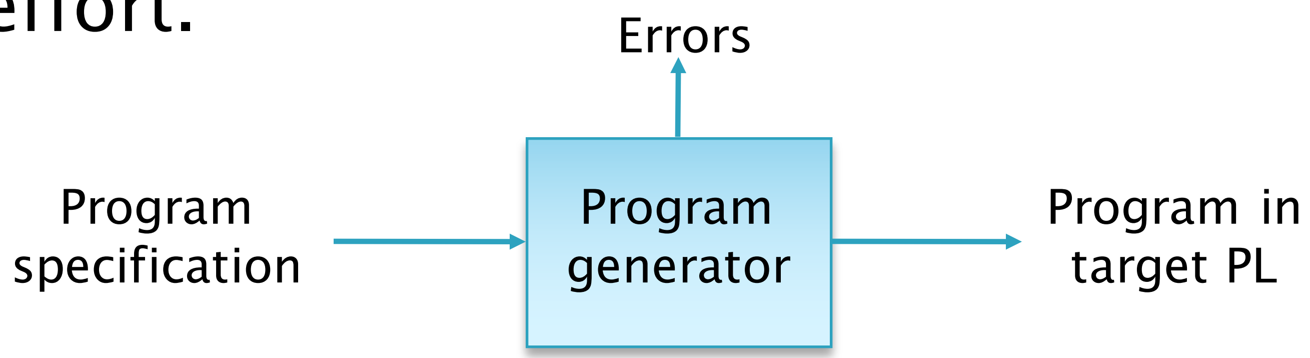
- Fundamental activities divided into those that bridge the specification gap and execution gap.
  - Program generation activities
  - Program execution activities

# Cont.

- Program generation activities
  - A program generation activity aims at automatic generation of a program.
  - A source language is a specification language of an application domain and the target language is procedure oriented PL.
  - Program generator introduces a new domain between the application and PL domain , call this the program generator domain.
  - Specification gap now between Application domain and program generation domain, reduction in the specification gap increases the reliability of the generated program.

# Cont.

- This arrangement also reduces the testing effort.

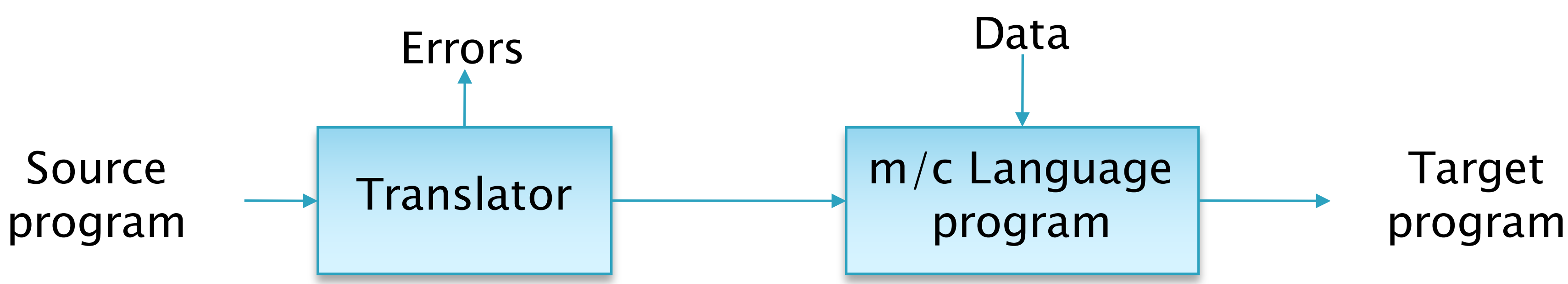


# Cont.

- Program Execution
  - Two popular model
    - Program Translation
    - Program Interpretation

# Cont.

- The program translation model bridges the execution gap by translating a sources program into program in the machine or assembly language of the computer system, called target program.



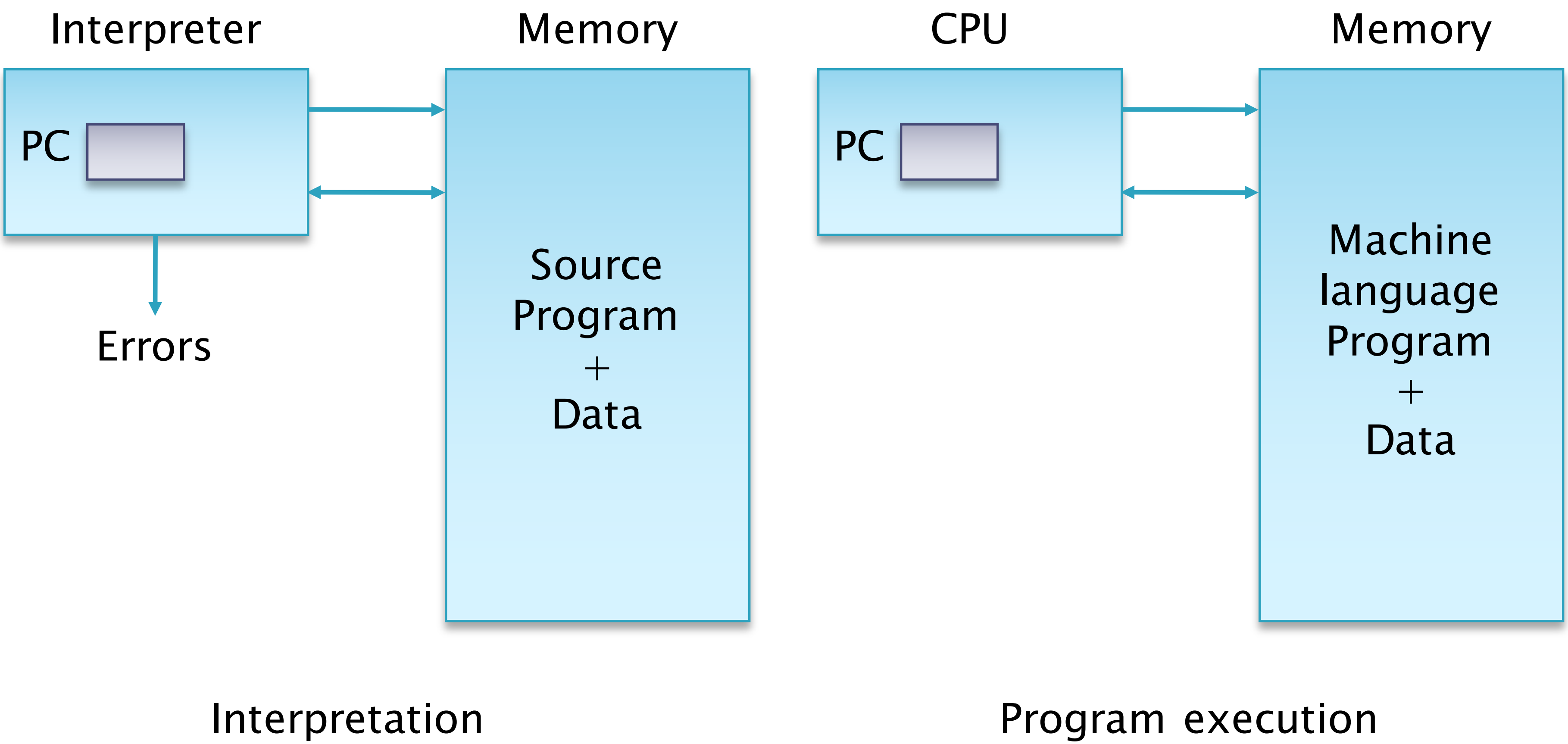
# Cont.

- Characteristics of the program translation model:
  - 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.

# Cont.

- **Program interpretation:** during interpretation interpreter takes source program statement, determines its meaning and performs actions which implement it.
- The function of an interpreter is same as the execution of machine language program by CPU.

# Cont.


Interpreter
Memory
CPU
Memory
PC
PC
Source
Program
+
Data
Machine
language
Program
+
Data
Errors
Interpretation
Program execution

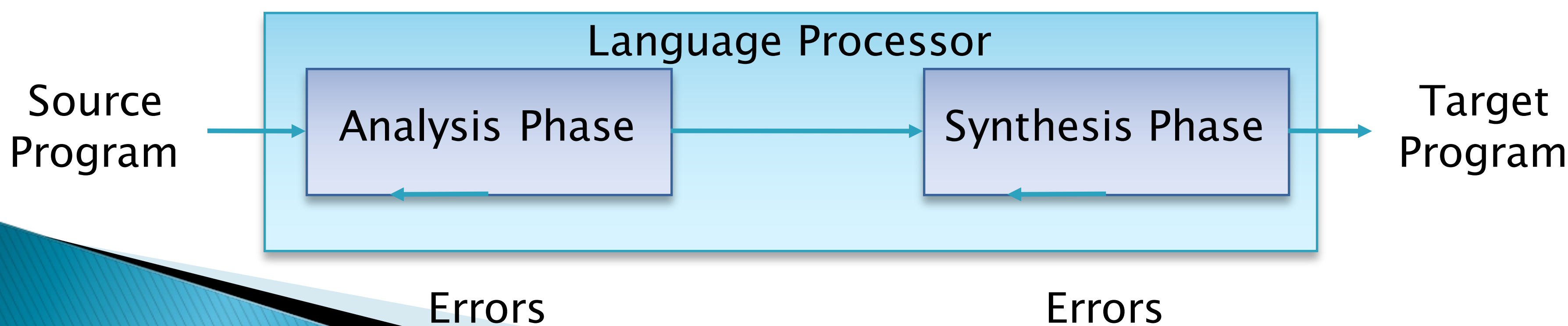
# Cont.

- Characteristics
  - The source program is retained in the source form itself, no target program form exists,
  - A statement is analyzed during its interpretation.
- Comparison
  - In translator whole program is translated into target and if modified the source program, whole source program is translated irrespective to size of modification.
  - That not the in case of interpreter, interpretation is slower than execution of m/c language program.

# Fundamental of Language Processoing

- Language Processing = Analysis of SP + Synthesis of TP.
- Analysis phase of Language processing
- **Lexical rules** which govern the formation of valid lexical units in the source language.
- **Syntax rules** which govern the formation of the valid statements in the source language.
- **Semantic rules** which associate meaning with the valid statements of the language.

- The synthesis phase is concerned with the construction of target language statement which have same meaning as a source statement.
  - Creation of data structures in the target program(**memory allocation**)
  - Generation of target code.(**Code generation**)



```
percent_profit  = (profit*100) / cost_price;
```

Lexical Analysis

Syntax Analysis

Semantic Analysis

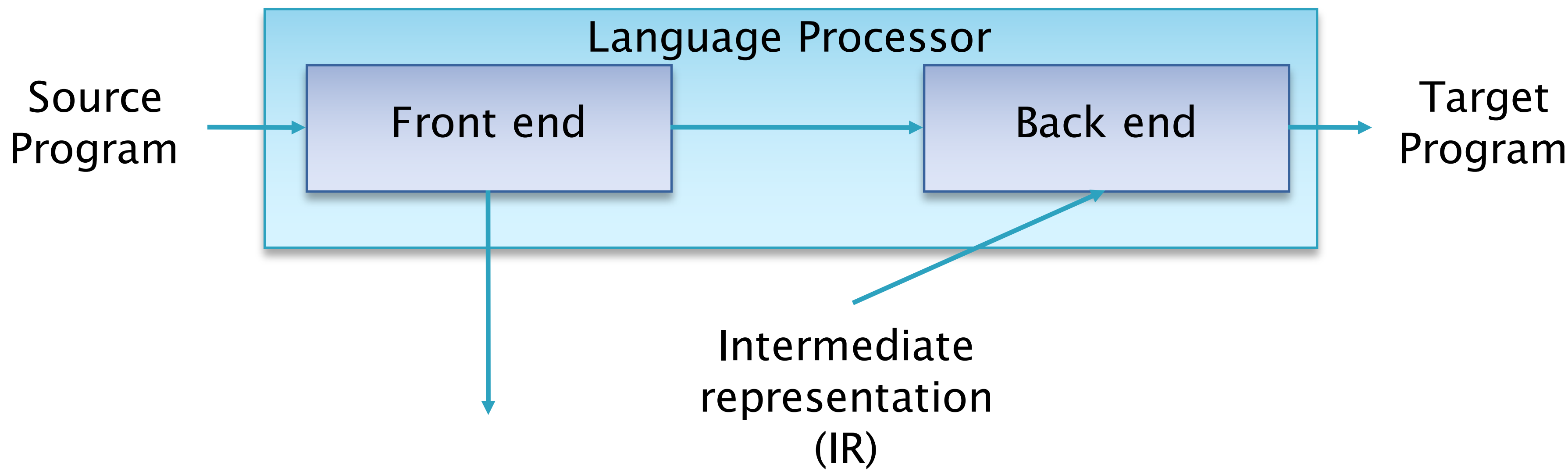
# Cont.

- **Forward references**: for reducing execution gap the language processor can performed on a statement by statement basis.
- Analysis of source statement can be immediately followed by synthesis of equivalent target statements. But this may not feasible due to :**Forward reference**
- "A forward reference of a program entity is a reference to the entity which precedes its definition in the program."

# Cont.

- **Language processor pass**: "A language processor pass is the processing of every statement in a source program, or its equivalent representation, to perform a language processing function."
- **Intermediate representation(IR)**: "An intermediate representation is a representation of a source program which reflects the effect of some, **but not all**, analysis and synthesis tasks performed during language processing."

# Cont.

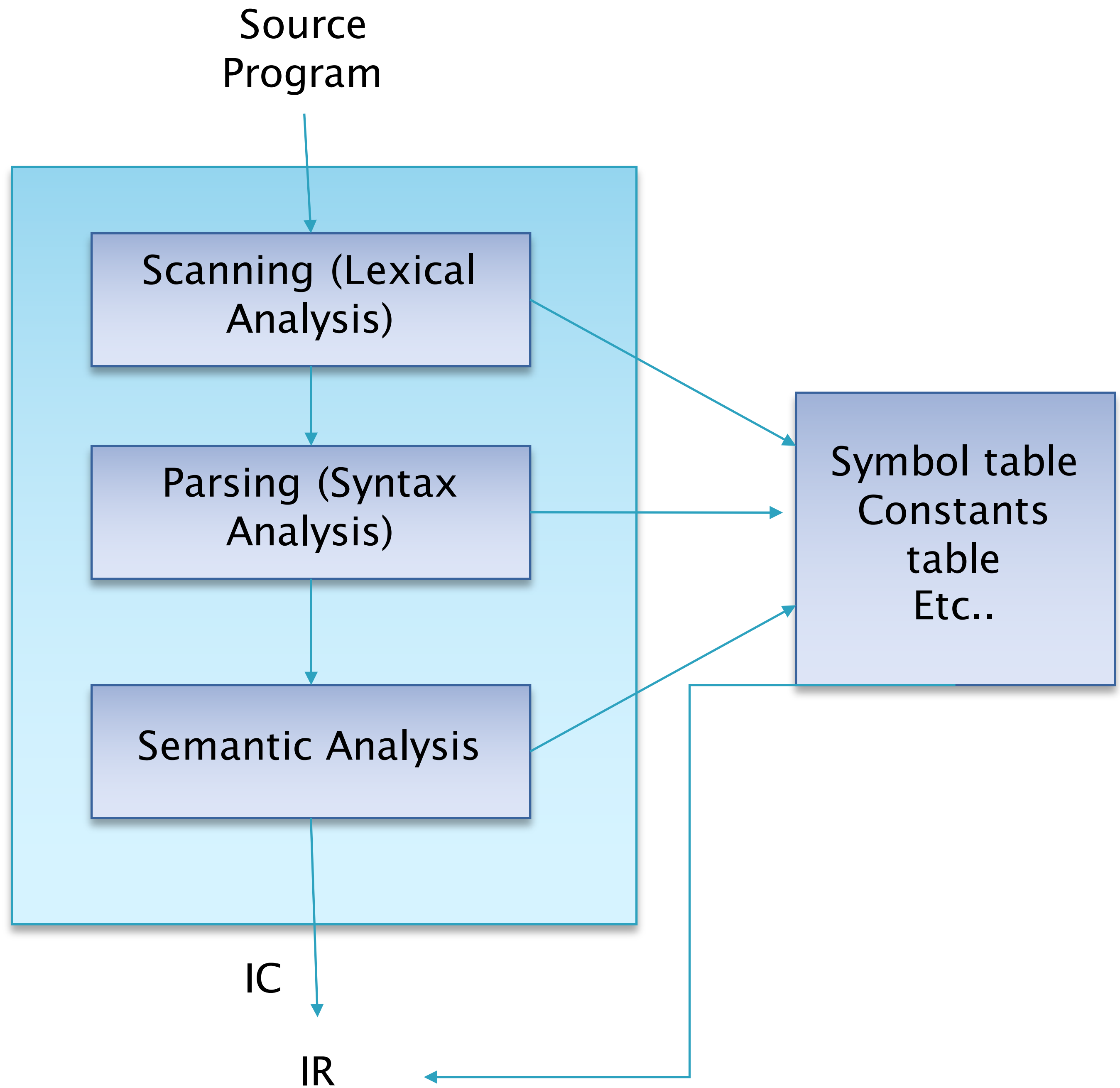

Language Processor
Source Program
Front end
Back end
Target Program
Intermediate representation (IR)

# Cont.

- **Semantic Action:** “All the actions performed by the front end, except lexical and syntax analysis, are called semantic action.
  - Checking semantic validity of constructs in SP
  - Determining the meaning of SP
  - Constructing an IR

# Toy Compiler

- The Front End
  - The front end performs lexical, syntax and semantic analysis of the source program, each kind of analysis involves the following functions:
    - Determine validity of source statement from the view point of the analysis.
    - Determine the 'content' of a source statement
      - For lexical, the lexical class to which each lexical unit belongs.
      - Syntax analysis it is syntactic structure of source program.
      - Semantic analysis the content is the meaning of a statement.
    - Construct a suitable representation of source statement for use by subsequent analysis function/synthesis phase.


Source
Program
Scanning (Lexical
Analysis)
Parsing (Syntax
Analysis)
Semantic Analysis
Symbol table
Constants
table
Etc..
IC
IR

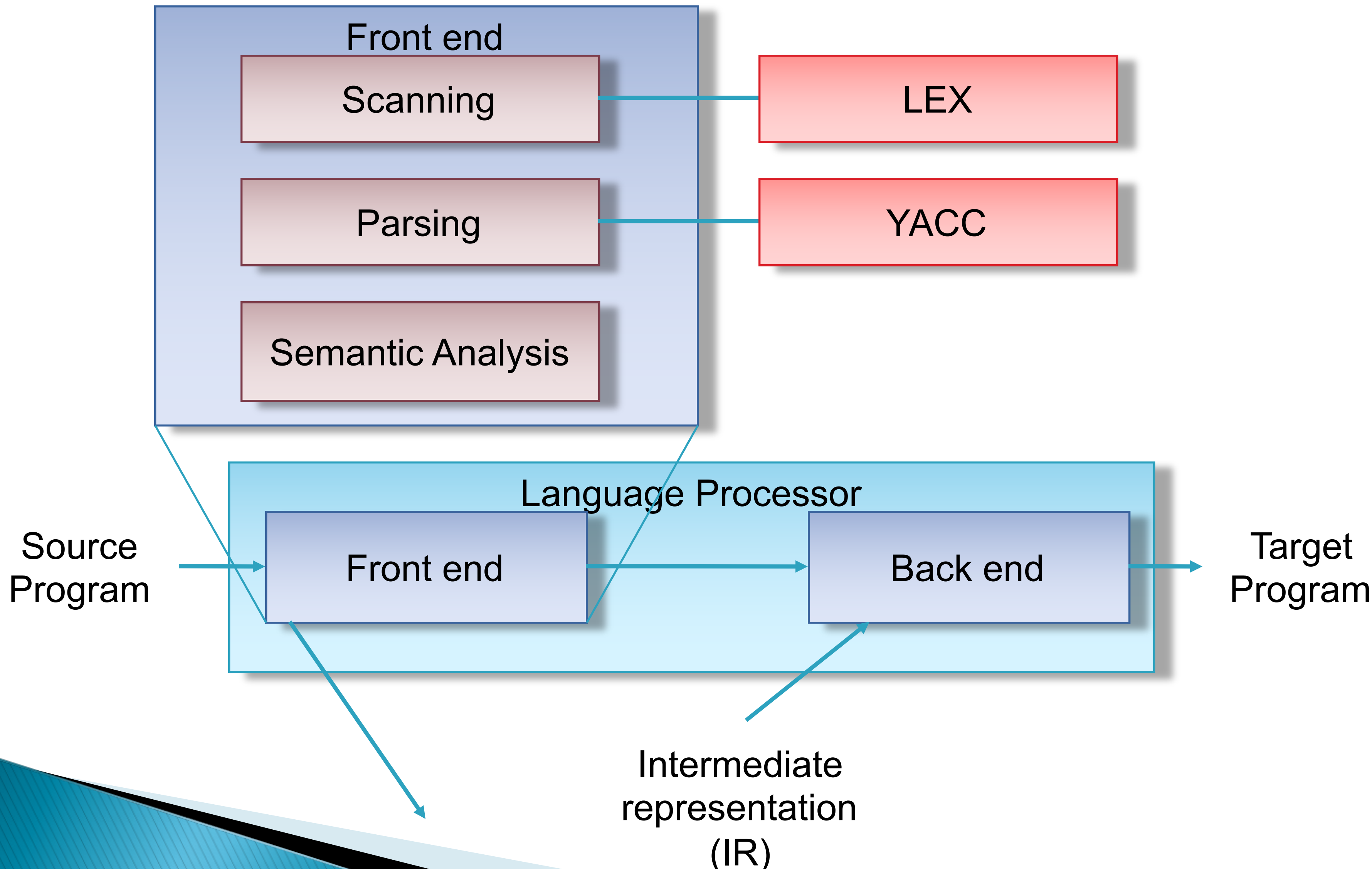
- Out put of front end produced two components: (IR)
  - Table of information
    - The symbol table which contain information concerning all identifier used in the source program.
  - An intermediate code (IC) which is a description of the source program.
    - The IC is a sequence of IC units, each IC unit representing the meaning of one action in SP. IC units may contain references to the information in various table.

# Cont.

- Lexical Analysis (Scanning):
  - Lexical analysis identifies the lexical units in source statement
  - it then classifies the unit into different classes
  - Ex. Id's, Constant reserved id's etc. and enters them into different tables.
  - This classification may be based on the nature of a string or on the specification of the source language.
  - Lexical analysis build descriptor called token, for each lexical unit. It contain two fields class code and number in class
    - Class code: identifies the class to which a lexical unit belongs.
    - Number in class: entry number of lexical unit in the relevant table.

- Syntax Analysis(Parsing)
  - Syntax analysis process the string token built by lexical analysis to determine the statement class e.g. assignment statement, if statement, etc.
  - Syntax Analysis builds an IC which represents the structure of the statement.
  - IC is passed to semantic analysis to determine the meaning of the statement.

- Semantic analysis
  - Semantic analysis of declaration statements differs from the semantic analysis of imperative statements.
  - The former results in addition of information to the symbol table e.g. Type, length and dimensionality of variables.
  - The latter identifies the sequence of action necessary to implement the meaning of a source statement.
  - When semantic analysis determines the meaning of a sub tree in the IC, it adds information to a table or adds an action to sequence of the action.
  - The analysis ends when the tree has been completely processed. The update tables and the sequence of action constitute the IR produced by the analysis phase.



Front end
Scanning
Parsing
Semantic Analysis
LEX
YACC
Language Processor
Source Program
Front end
Back end
Target Program
Intermediate representation (IR)

# LEX

Lex accepts an input specification which consist of three components.

1. Definations
2. Rules
3. User Code

This components are seprated by %% symbol.

# Defination Section

- It contains declaration of simple name defination to simplify scanner specification or in simple words it contains the variables to hold regular expressions.

- For example, if you want to define D as a numerical digit, you would write the following: D  [0-9]

# Rules Section

- Once you have defined your terms, you can write the rules section. It contains strings and expressions to be matched by the **yylex**subroutine, and C commands to execute when a match is made.
- This section is required, and it must be preceded by the delimiter %%(double percent signs), whether or not you have a definitions section. The **lex** command does not recognize your rules without this delimiter.

# Defining Patterns in Lex

- *x*

  match the character `*x*‘
- .

  any character except newline.
- [xyz]

  a "character class"; in this case, the pattern matches either an `x', a `y', or a `z‘.
- *r**

  zero or more *r*'s, where *r* is any regular expression
- *r*+

  one or more *r*'s

# User Code Section

- This section can contain any C/C++ program code that user want to execute.
- Yylex() function is used to flex compiler , which is embeded in this section so user need to include this function.

# Sample flex program

```
%{
 #include <iostream>
%}
 %%
[ \t] ;
 [0-9]+\.[0-9]+ { cout << "Found a floating-point number:" << yytext << endl;
     }
[0-9]+ { cout << "Found an integer:" << yytext << endl; }
[a-zA-Z0-9]+ { cout << "Found a string: " << yytext << endl; }
 %%
main() {
// lex through the input:
 yylex();
 }
```

# YACC

Each string specification in the input to yacc resembles a grammar production.

The parser generated by yacc performs reductions according to this grammar.

The action associated with a string specification are executed when a reduction is made according to specification.

# Finite Automata

- A recognizer for a language is a program that takes a string x as an input and answers "yes" if x is a sentence of the language and "no" otherwise.
- One can compile any regular expression into a recognizer by constructing a generalized transition diagram called a finite automation.
- A finite automation can be deterministic means that more than one transition out of a state may be possible on a same input symbol.
- Both automata are capable of recognizing what regular expression can denote.

# Nondeterministic Finite Automata (NFA)

- A nondeterministic finite automation is a mathematical model consists of :

1. a set of states S;
2. a set of input symbol, $\Sigma$, called the input symbols alphabet.
3. a transition function move that maps state-symbol pairs to sets of states.
4. a state so called the initial or the start state.
5. a set of states F called the accepting or final state

# Deterministic Finite Automata (DFA)

- A deterministic finite automation is a special case of a non-deterministic finite automation (NFA) in which

1. no state has an -transition
2. for each state s and input symbol a, there is at most one edge labeled a leaving s.

- A DFA has st most one transition from each state on any input. It means that each entry on any input. It means that each entry in the transition table is a single state (as oppose to set of states in NFA).

# Syntax Analysis

- During the first Scanning phase i.e Lexical Analysis Phase of the compiler,symbol table is created by the compiler which contain the list of leximes or tokens.

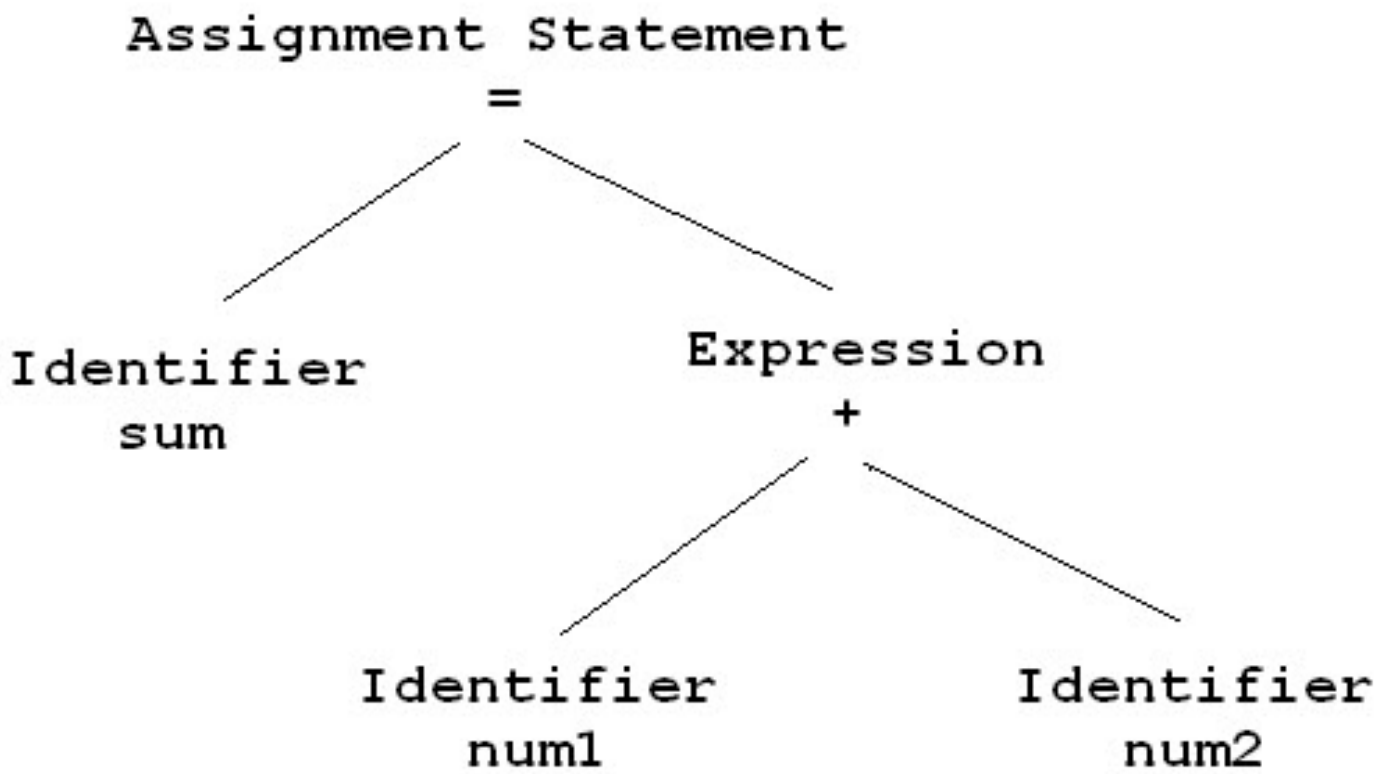
- It is also Called as Hierarchical Analysis or Parsing.

- It Groups Tokens of source Program into **Grammatical Production**

- In Syntax Analysis System Generates **Parse Tree**

# Parse Tree Generation :

sum = num1 + num2



# Explanation : Syntax Analysis

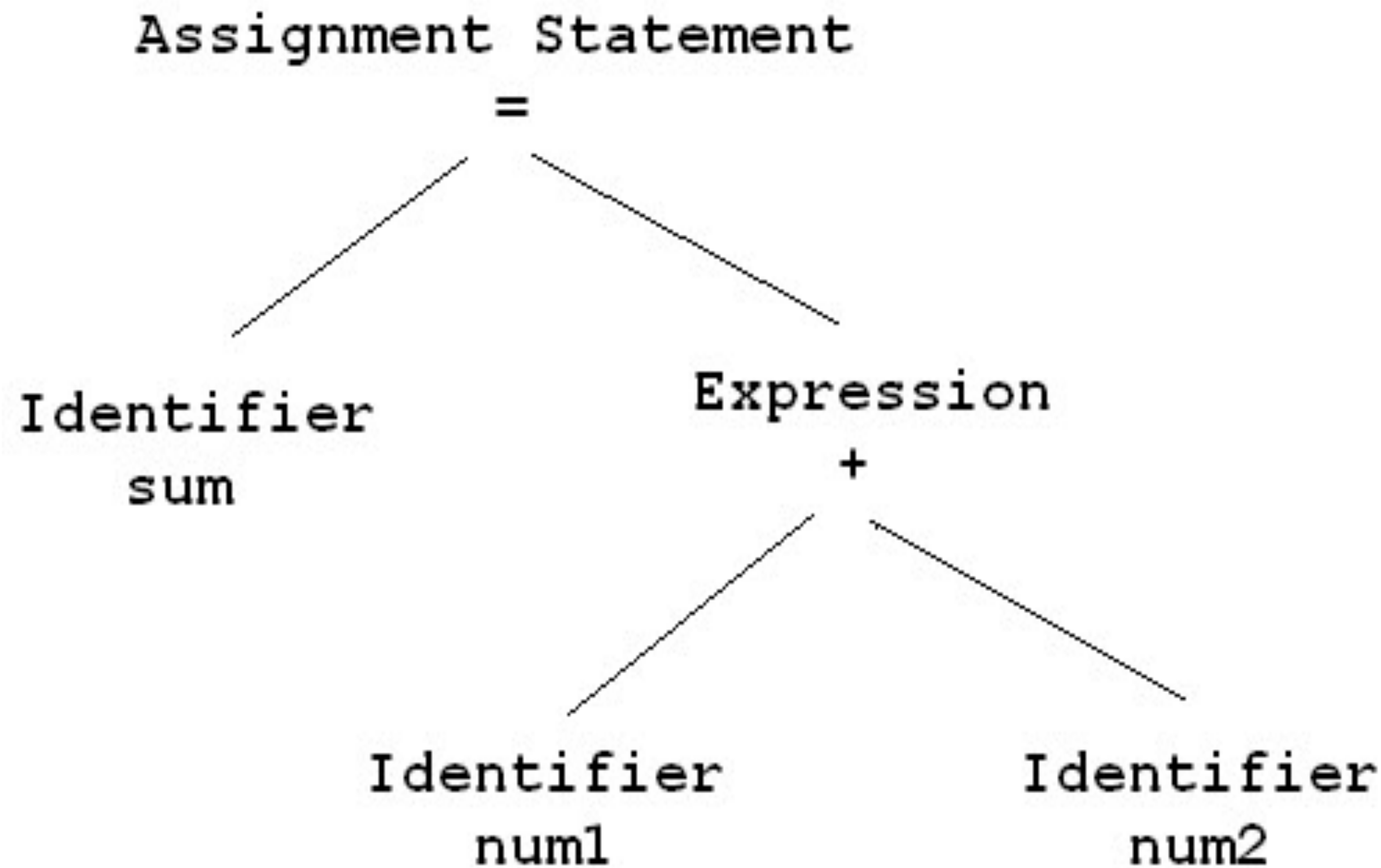
- We know , Addition operator plus ('+') operates on two Operands
- Syntax analyzer will just check whether plus operator has two operands or not . It does not checks the type of operands.
- Suppose One of the Operand is String and other is Integer then it does not throw error as it only checks whether there are two operands associated with '+' or not .
- So this Phase is also called Hierarchical Analysis as it generates Parse Tree Representation of the Tokens generated by Lexical Analyzer

# Semantic Analysis

- Syntax analyzer will just create parse tree. Semantic Analyzer will check actual meaning of the statement parsed in parse tree. Semantic analysis can compare information in one part of a parse tree to that in another part (e.g., compare reference to variable agrees with its declaration, or that parameters to a function call match the function definition).

# Semantic Analysis is used for the following –

1. **Maintaining the Symbol Table** for each block.
2. Check **Source Program for Semantic Errors**.
3. **Collect Type Information** for Code Generation.
4. **Reporting compile-time errors** in the code (except syntactic errors, which are caught by syntactic analysis)
5. **Generating the object code** (e.g., assembler or intermediate code)


Assignment Statement
=
Identifier
sum
Expression
+
Identifier
num1
Identifier
num2

# Now In the Semantic Analysis Compiler Will Check –

1. Data Type of First Operand
2. Data Type of Second Operand
3. Check Whether + is Binary or Unary.
4. Check for Number of Operands Supplied to Operator Depending on Type of Operator (Unary | Binary | Ternary)

# Fundamentals of Lang. Specification

**Terminal Symbol**

$\Sigma$ Denotes character set with all symbols.

$\{:, ;, `, ', \ldots\}$ this all are **metasymbols.**

Differentiate from terminal symbol

String – finite sequence of symbols

Nonterminal symbol– name of syntax category of symbol

denoted by single capital letter

eg. Noun,verb …

- Productions

  A productions also called a rewriting rule, is a rule of of grammar.

  A production has the form

  A Nonterminal symbol = String of Ts and NTs

  Example

  <Noun Phrase> ::= <Article> <Noun>

  <Article> ::= a | an | the

  <Noun>::= boy | apple

- Distinguished symbol/start NT of grammer

# Programming Language Grammars

- Grammar (G)

  A grammar G of a language $L_G$ is a Quadruple ($\Sigma$, SNT, S, P) where

  $\Sigma$ = is the set of Ts

  SNT= is the set of NTs

  S = is the distinguished symbols /starting symbol

  P= is the set of productions

# Derivation

A grammar G is used for two purpose

To generate valid strings of $L_G$

To recognized valid strings of $L_G$

The derivation operations helps to **generate valid strings.**

## Derivation -- Example

<Noun Phrase> ::= <Article> <Noun>

<Article> ::= a | an | the

<Noun>::= boy | apple

**Suppose we want to derivate strings “the boy”**

“$\Rightarrow$” denote direct derivation.

<Noun Phrase> ⇒<Article> <Noun>
⇒the <Noun>
⇒the boy → Leftmost Derivation

<Noun Phrase> ⇒<Article> <Noun>
⇒<Article> boy
⇒the boy → Rightmost Derivation

| | | |
|---|---|---|
| *< Sentence >* | ::= | *< Noun Phrase >< Verb Phrase >* |
| *< Noun Phrase >* | ::= | *< Article >< Noun >* |
| *< Verb Phrase >* | ::= | *< Verb >< Noun Phrase >* |
| *< Article >* | ::= | a \| an \| the |
| *< Noun >* | ::= | boy \| apple |
| *< Verb >* | ::= | ate |

<Sentence> ⇒<Noun Phrase> <Verb Phrase>

⇒<Article> <Noun><Verb Phrase>

⇒the <Noun><Verb Phrase>

⇒the boy<Verb Phrase>

⇒the boy<Verb><Noun Phrase>

⇒the boy ate <Noun Phrase>

⇒the boy ate <Article> <Noun>

⇒the boy ate an <Noun>

⇒the boy ate an apple

Reductions

The reductions operation helps to recognize valid strings.

| | | |
|---|---|---|
| *< Sentence >* | ::= | *< Noun Phrase >< Verb Phrase >* |
| *< Noun Phrase >* | ::= | *< Article >< Noun >* |
| *< Verb Phrase >* | ::= | *< Verb >< Noun Phrase >* |
| *< Article >* | ::= | `a` \| `an` \| `the` |
| *< Noun >* | ::= | `boy` \| `apple` |
| *< Verb >* | ::= | `ate` |

| *Step* | *String* |
|---|---|
| 0 | `the boy ate an apple` |
| 1 | *< Article >* `boy ate an apple` |
| 2 | *< Article > < Noun >* `ate an apple` |
| 3 | *< Article > < Noun > < Verb >* `an apple` |
| 4 | *< Article > < Noun > < Verb > < Article >* `apple` |
| 5 | *< Article > < Noun > < Verb > < Article > < Noun >* |
| 6 | *< Noun Phrase > < Verb > < Article > < Noun >* |
| 7 | *< Noun Phrase > < Verb > < Noun Phrase >* |
| 8 | *< Noun Phrase > < Verb Phrase >* |
| 9 | *< Sentence >* |

## Parse tree

- A parse tree is used to depict syntactic structure of a valid string as it emerges during a sequence of derivations or reductions

## Recursive Specification

- **A grammar is in recursive specification, if NT being defining in a production, itself occurs in a RHS string of the production, e..g. X::=AXB**
- The RHS alternative employing recursion is called recursive rules.

# Recursive Specification

Consider the grammar G

<exp> ::= <exp> + <term> | <term>

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

<factor> ::= <factor> ↑ <primary> | <primary>

<primary> ::= <id> | <constant> | (<exp>)

<id> ::= <letter> | <id> [<letter> | <digit>]

<const> ::= [+ | −] <digit> | <const> <digit>

<letter> ::= a | b | c | ... | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

# Recursive Specification

[..] denotes an optional specification

$$
\begin{aligned}
<id> \quad &::= \quad <letter> \mid <id> [<letter> \mid <digit>] \\
<const> \quad &::= \quad [+ \mid -] <digit> \mid <const><digit>
\end{aligned}
$$

$$
\begin{aligned}
<id> \quad &::= \quad <letter> \mid <id><letter> \mid <id><digit> \\
<const> \quad &::= \quad + <digit> \mid - <digit> \\
&\quad\;\; \mid <const><digit>
\end{aligned}
$$

## Recursive Specification

- Two types of recursive rules
- Left recursive rule → NT appears on the extreme left in the recursive rule
- Right recursive rule → NT appears on the extreme right in the recursive rule

## Recursive Specification

### Indirect recursion

Occurs when two or more NTs are defined in terms of one another.

Such recursion is useful for specifying nested constructs in a language

$$
\begin{array}{rcl}
<exp> & ::= & <exp> + <term> \mid <term> \\
<term> & ::= & <term> * <factor> \mid <factor> \\
<factor> & ::= & <factor> \uparrow <primary> \mid <primary> \\
<primary> & ::= & <id> \mid <constant> \mid (<exp>)
\end{array}
$$

# Recursive Specification

Direct recursion is not useful in situations where a limited number of occurrences is required. For example, the recursive specification

$$<id> ::= <letter> | <id> [<letter> | <digit>]$$

permits an identifier string to contain an unbounded number of characters, which is not correct. In such cases, controlled recurrence may be specified as

$$<id> ::= <letter> \{<letter> | <digit>\}_0^{15}$$

where the notation $\{\ldots\}_0^{15}$ indicates 0 to 15 occurrences of the enclosed specification.

# Grammars are classified as

- **Type-0 (Phrase structure grammar)**

  $\alpha = \beta$ (strings of Ts and NTs)

-Permits arbitrary substitutions of strings

-No limitation on production rules: at least one nonterminal on LHS.

-not relevant to specification of PLs.

Example:

Start = <S>

<S> ⇒ <S> <S>

<S> ⇒ <A> <B> <C>

<A> ⇒ a

<B> ⇒ b

<C> ⇒ c

<S> ⇒ ϵ

<A><B> ⇒ <B><A>

<B><A> ⇒ <A><B>

<A><C> ⇒ <C><A>

<C><A> ⇒ <A><C>

<B><C> ⇒ <C><B>

Strings generated:

ϵ, abc, aabbcc, cabcab, acacacacacacbbbbbb, ...

# Type-1 (Context sensitive grammar)

$\alpha A \beta = \alpha \Pi \beta$

-not relevant to specification of PLs.

# Type-2 (Context free grammar)

- A = Π
- Limit production rules to have exactly one nonterminal on LHS, but anything on RHS.

–suited for programming language specification

Example:

<PAL> ⇒ 0 <PAL> 0 Start = <PAL>

⇒ 1 <PAL> 1

⇒ 0

⇒ 1

⇒ ϵ

Strings generated:

ϵ, 1, 0, 101, 001100, 111010010111, …

# Type-3 (regular grammar/ linear grammar)

A= tB|t or Bt|t

<id> = l|<id>l|<id>|d

-Limit production rules to have exactly one nonterminal on LHS and at most one nonterminal and terminal on RHS:

- restricted to the specification of lexical units
- nesting of construct or matching parenthesis can not be specified

Example:

<A>⇒<B> 0 Start = <A>

<B> ⇒<A> 1

<A> ⇒ ϵ

Strings generated:

ϵ, 10, 1010, 101010, 10101010, …

# Operator Grammar (OG)

An Operator grammar is a grammar none of whose productions contain two or more consecutives NTs in any RHS alternatives.

- 

## Ambiguity in Grammatic specification

- For a given string and grammar, two distinct parse tree exists then grammar known as ambiguous grammar.
- For example

$$< exp > \; ::= \; < id > | < exp > + < exp > | < exp > * < exp >$$
$$< id > \; ::= \; \mathtt{a} \,|\, \mathtt{b} \,|\, \mathtt{c}$$

- Two parse tree exist for string a+b*c

< *exp* > ::= < *id* > | < *exp* > + < *exp* > | < *exp* > ∗ < *exp* >

< *id* > ::= a | b | c

<exp> ⇒<exp>+<exp>

⇒ <id>+<exp>

⇒ a+<exp>

⇒ a+<exp>*<exp>

⇒ a+<id>*<exp>

⇒ a+b*<exp>

⇒ a+b*<id>

⇒ a+b*c

<exp> ⇒<exp>*<exp>

⇒<exp>+<exp>*<exp>

<id>+<exp>*<exp>

⇒a+<exp>*<exp>

⇒a+<id>*<exp>

⇒a+b*<exp>

⇒a+b*<id>

⇒a+b*c

## Eliminating ambiguity

- An ambiguous grammar should be rewritten to eliminate ambiguity.
- The grammar must be rewritten such that reduction of '*' precedes the reduction of '+' in string a+b*c
- The normal method of achieving this is to use a hierarchy of NTs in the grammar and to associate the derivation or reduction of an operator with an appropriate NT.

# Programming Language Grammars

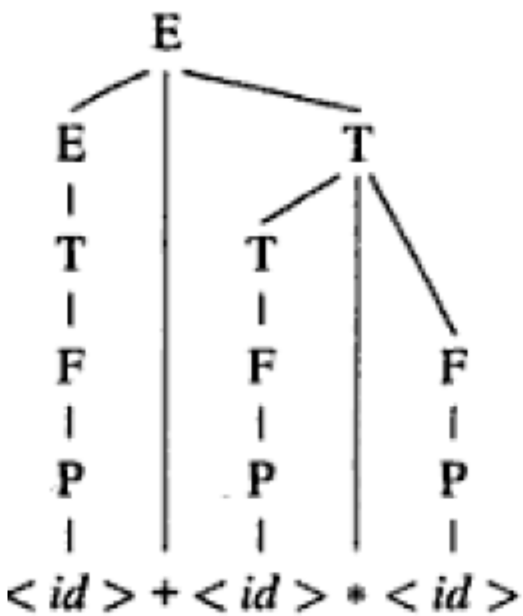
- Unambiguous Grammar

E:=E+T|T

T:=T*F|F

F:=P

P:=a|b|c




$$\langle exp \rangle ::= \langle id \rangle \mid \langle exp \rangle + \langle exp \rangle \mid \langle exp \rangle * \langle exp \rangle$$

$$\langle id \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$$

# Binding

- "A Binding is the association of an attribute of a program entity with a value"
- Binding time is the time at which a binding is performed.
- Different Binding times:
  - Language definition time of L
    - The keywords of the programming language L are bounded to their meanings. Example: main, for, while
  - Language implementation time of L
    - The time when language translator designed example the size of type **int** could be bounded to 2 or 4 bytes, its determined by the architecture of the target machine.

# Cont.

- Compilation time of P
  - The binding of the attributes of variables is performed. Example the **int** is bounded with a variable *var*.
- Execution init time of Proc
  - Memory addresses of local variables of procedure are bound at every execution init time of proc.
- Execution time of Proc
  - Value attribute binding may be done more then one during the execution of the procedure or function.

# Importance of binding time

- The binding time of an attribute of program entity determines the manner in which a language processor can handle the use of the entity.
- This affect execution efficiency of the target program.
- Type of binding
  - Static binding:
    - Static binding is a binding performed before the execution of program begins.
  - Dynamic binding
    - Dynamic binding is a binding performed after the execution of program has begun

# Lex program for symbol table

```
%{
#include<stdio.h>
int flag=0,flag2=0,flag3=0,value;
char *id,*datatype;
%}
datatype int|float|double|char;
%%
{datatype} {flag=1;datatype=yytext;}
([(A-Za-z)]+[(\_*)]*[0-9]*)*
{if(flag==1){flag2=1;id=yytext;}else{return
0;}}
[(\=?)] {if(flag2==1){flag3=1;}}
([0-9])*
{if(flag3==1){createsymboltable(datatype,id,y
ytext);}}
%%
```

- createsymboltable(datatype,id,value)

```
{
    printf("datatype=>%s\nid=>%s\nvalue=>%s",datatype,id,value);
}
main()
{
    yylex();
}
```