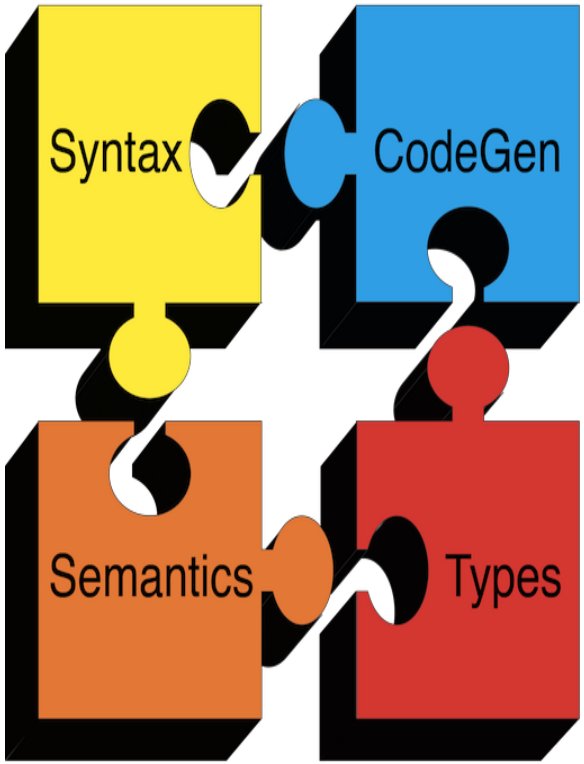
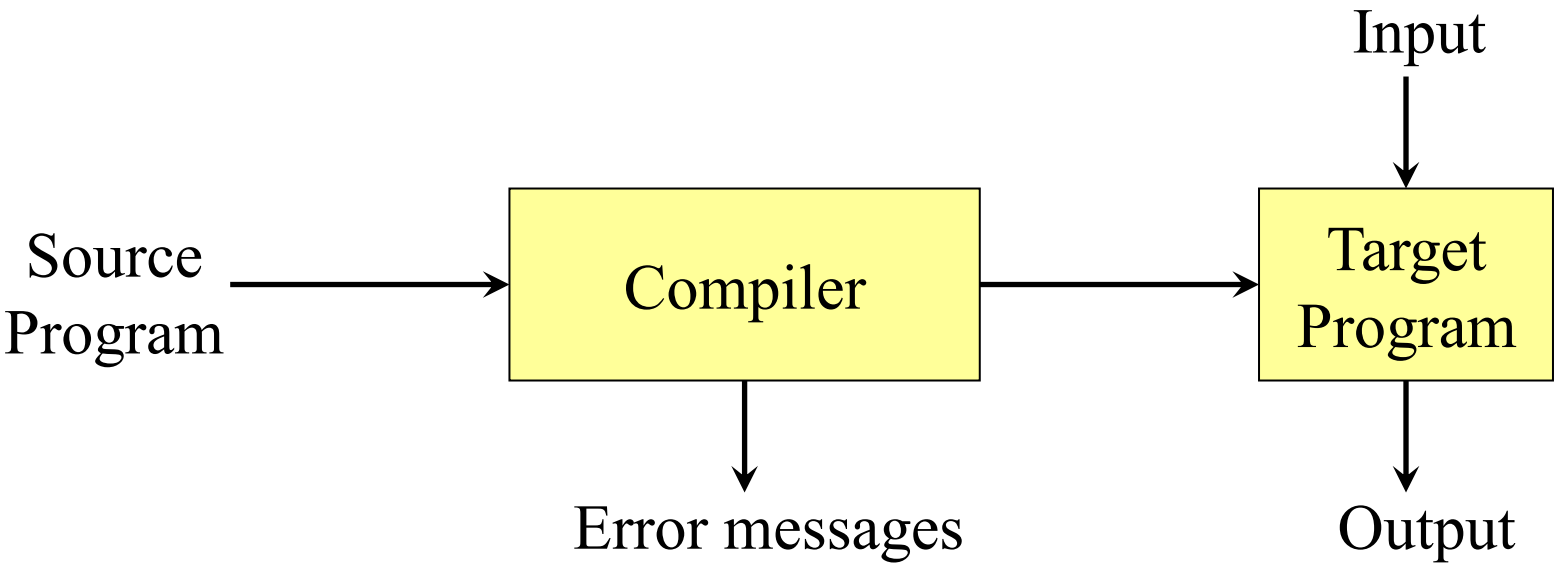


Compiler Theory

Introduction

Compilers

- **Compilation**
  - **Translation of a program written in a source language into a semantically equivalent program written in a target language**



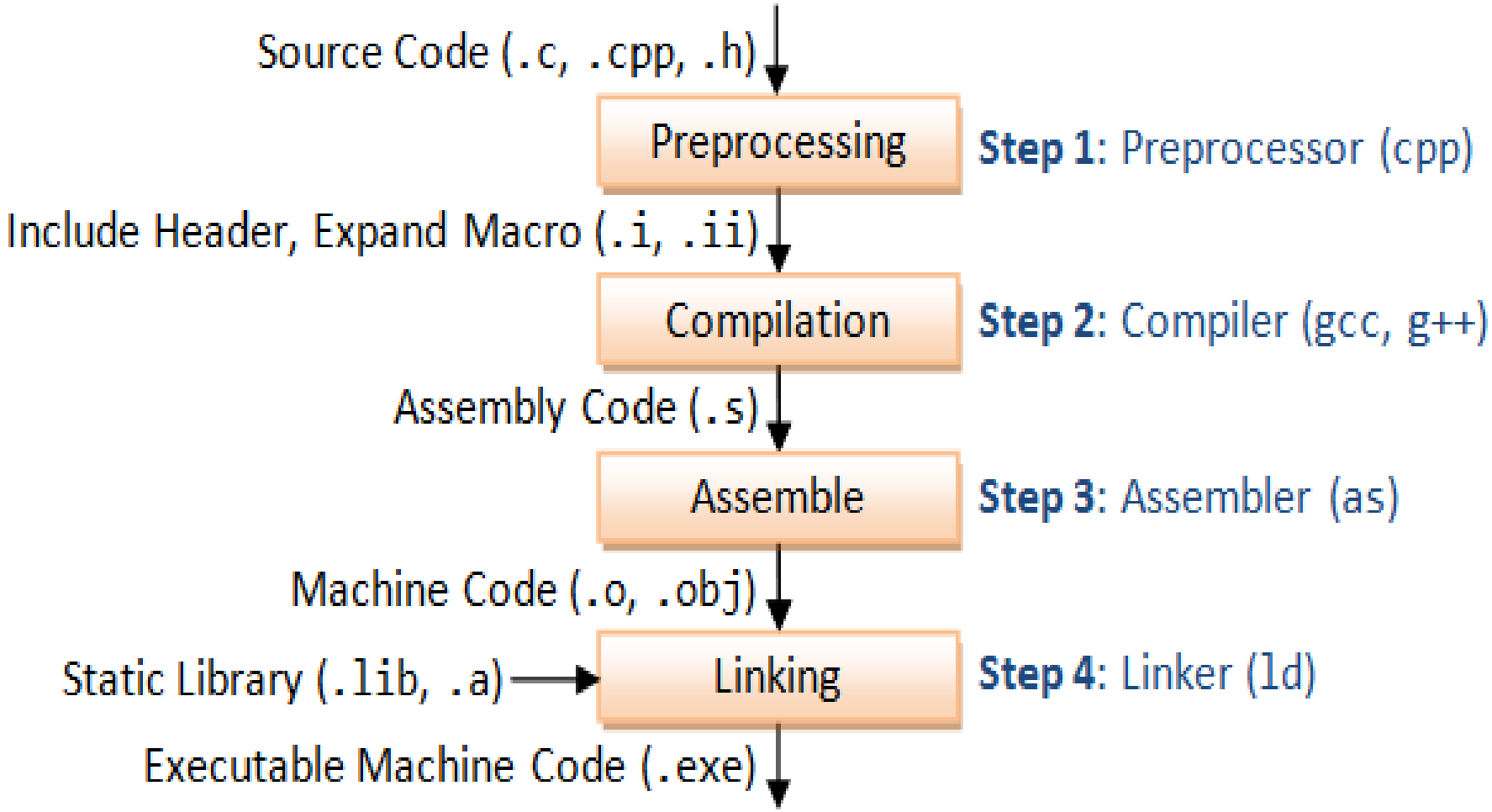
What is a Compiler?

- **Example of tasks of compiler**
  1. Add two numbers
  2. Move numbers from one location to another
  3. Move information between CPU and memory

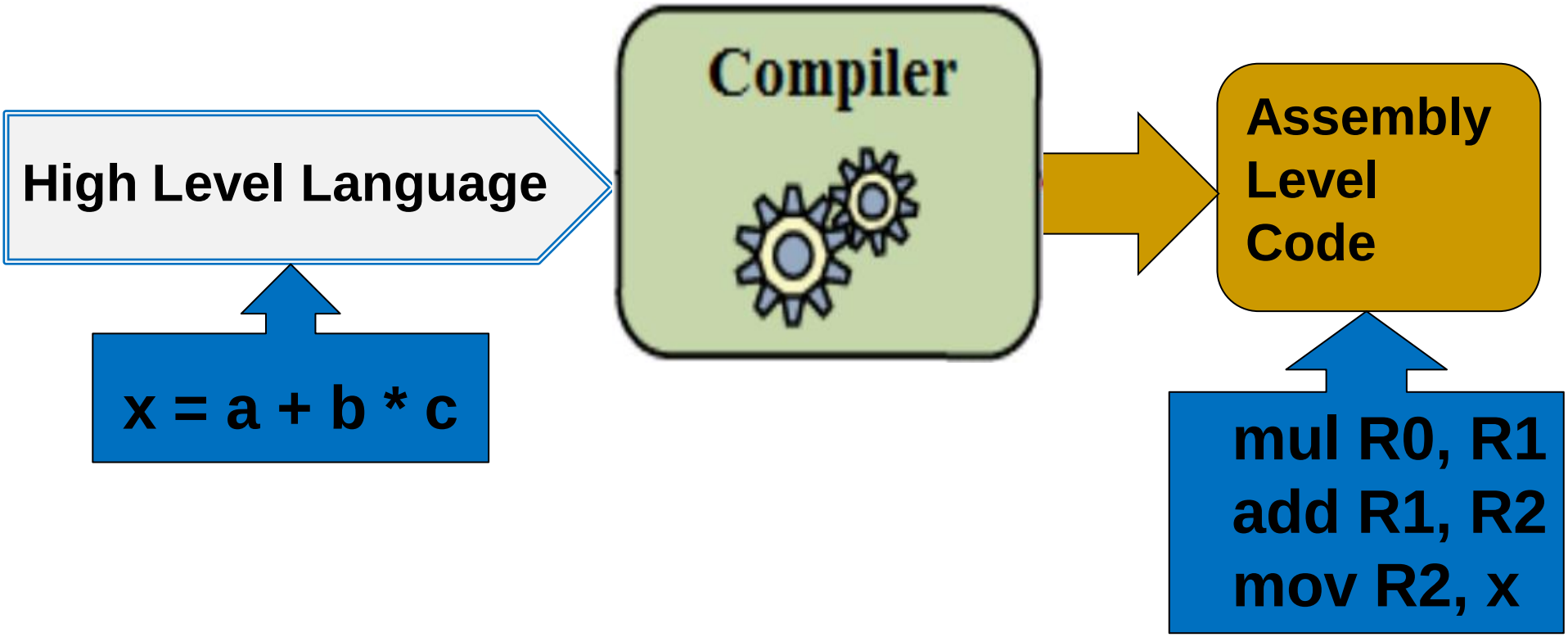
Use the Compiler Techniques

- *Programming Languages (C、C++…)*
- *Scripts (Javascript、bash)*
- *Editors* (syntax highlighting)
- *Pretty printers* (e.g. Doxygen)
- *Static checkers* (e.g. Lint and Splint)
- *Text formatters* (e.g. TeX and LaTeX)
- *Silicon compilers* (e.g. VHDL)
- *Query compilers* (Databases)

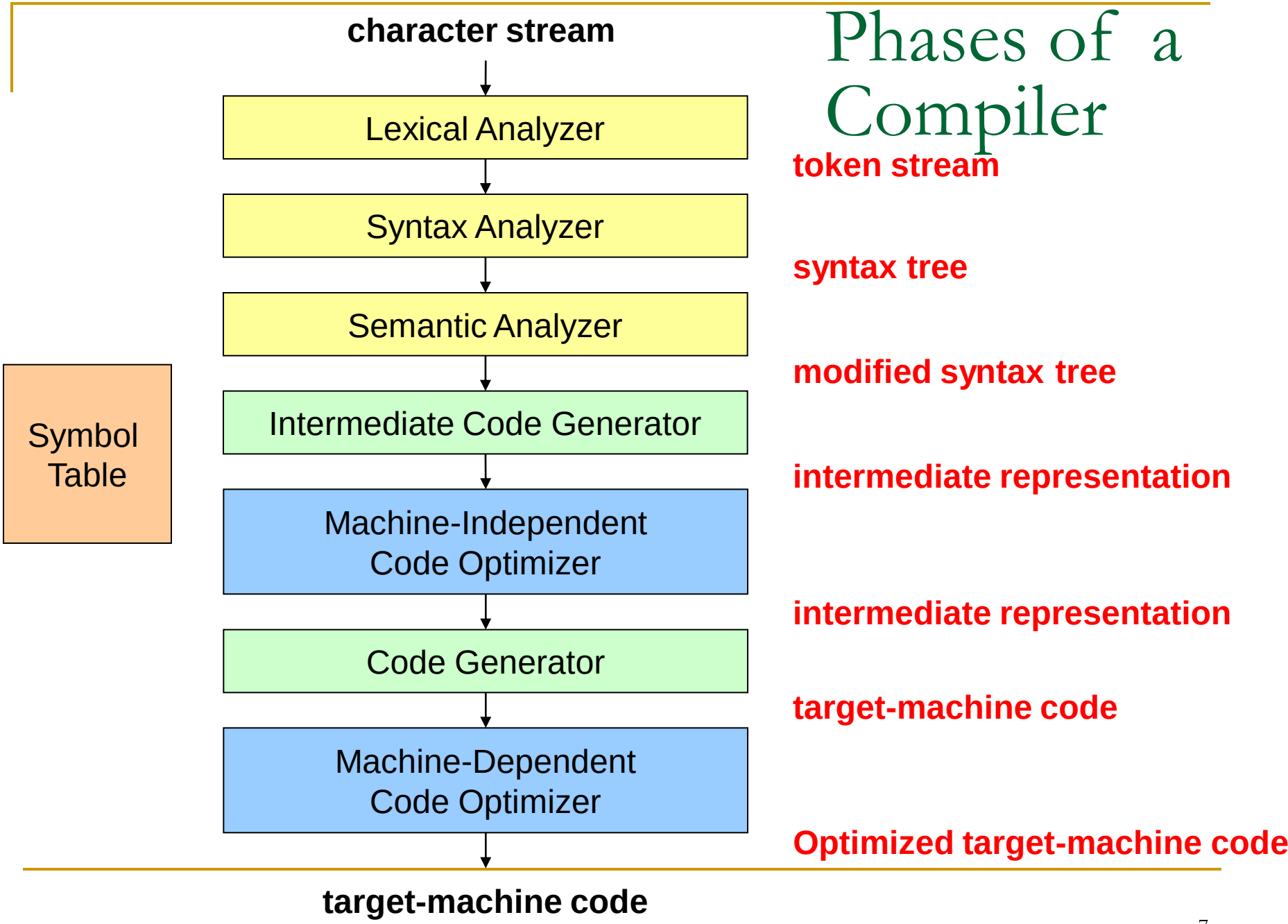
A Language-Processing System



Compiler

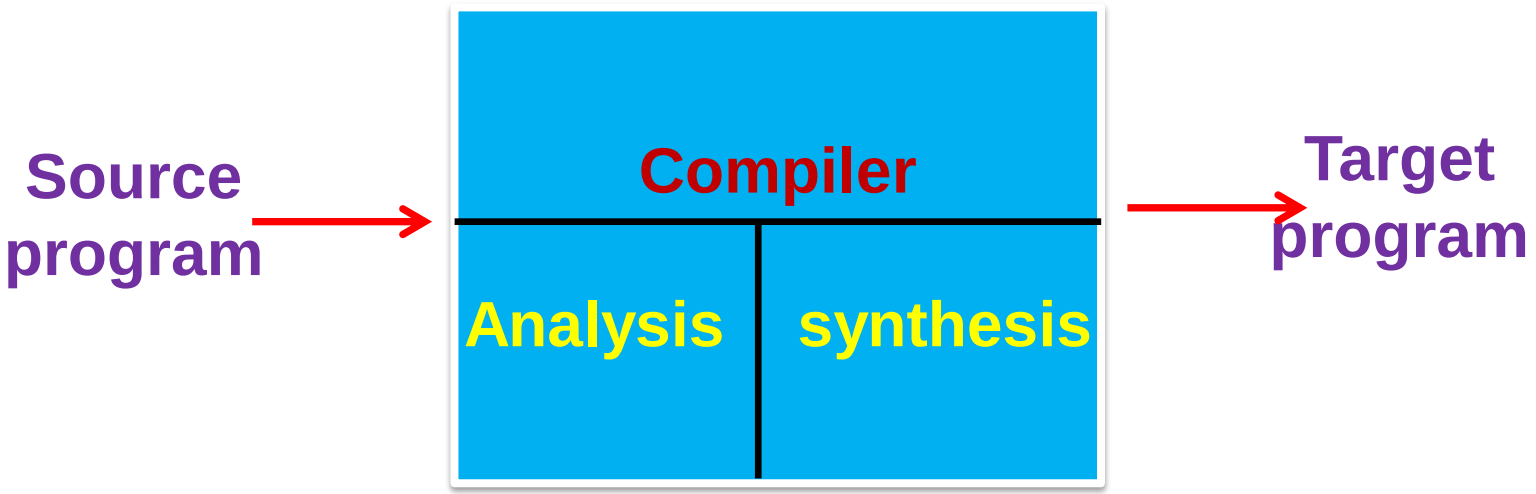

High Level Language
x = a + b * c
Compiler
Assembly Level Code
mul R0, R1
add R1, R2
mov R2, x

Phases of a Compiler



The Structure of a Compiler

- **Compiler maps a source program into a semantically equivalent target program.**
- **There are two parts to this mapping: analysis and synthesis.**



The Compiler Structure

The Analysis-Synthesis Model of Compilation

- There are two parts to compilation:
  - *Analysis* determines the operations implied by the source program which are recorded in a tree structure
  - *Synthesis* takes the tree structure and translates the operations therein into the target program

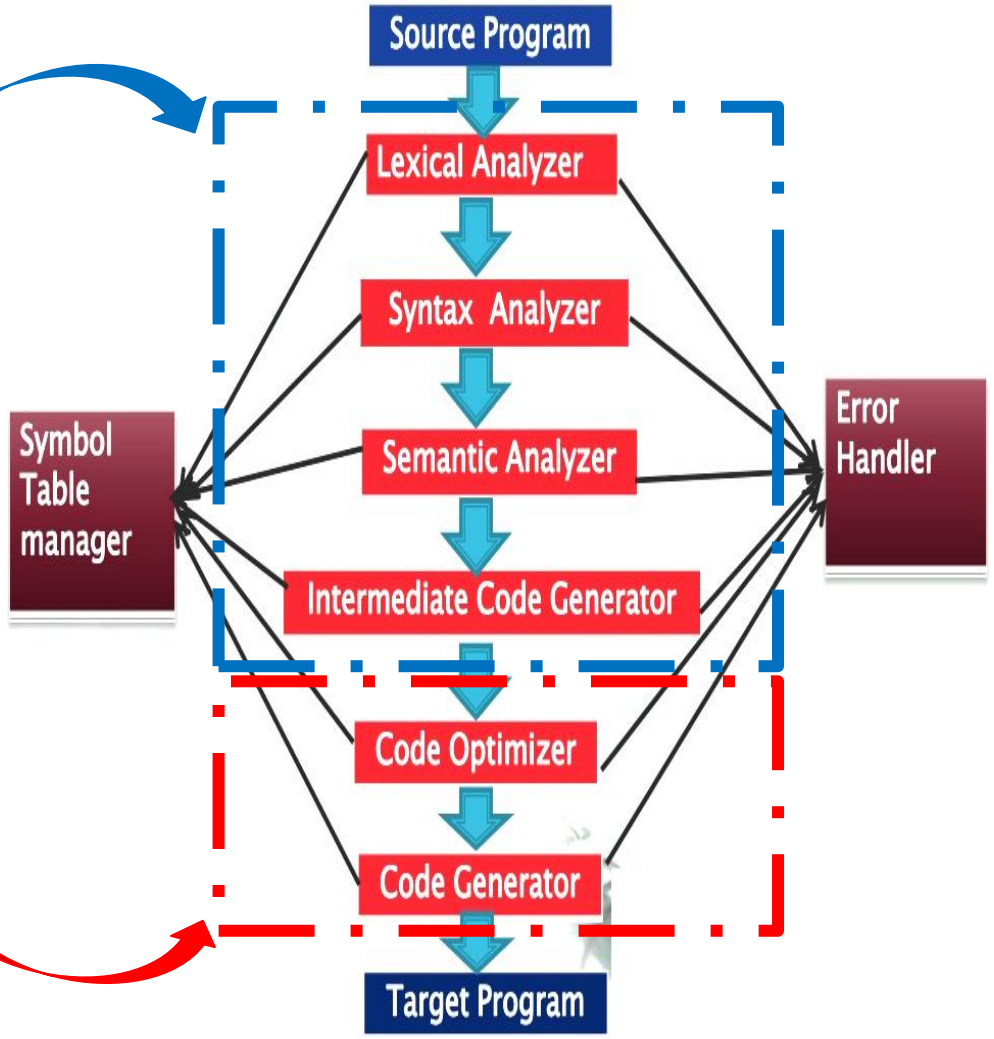
Grouping of Phases into Passes

Front End

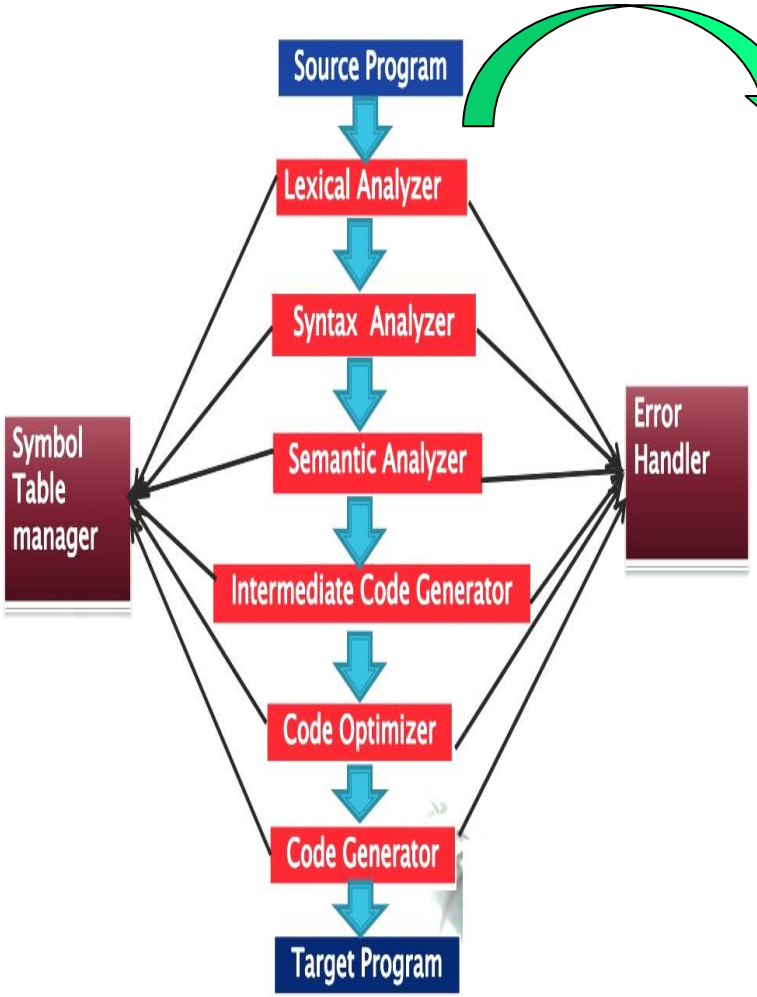
- Machine independent.
- Analyses the source program.
- Produces intermediate code

Back End

- machine dependent
- Synthesis the target program from intermediate code.

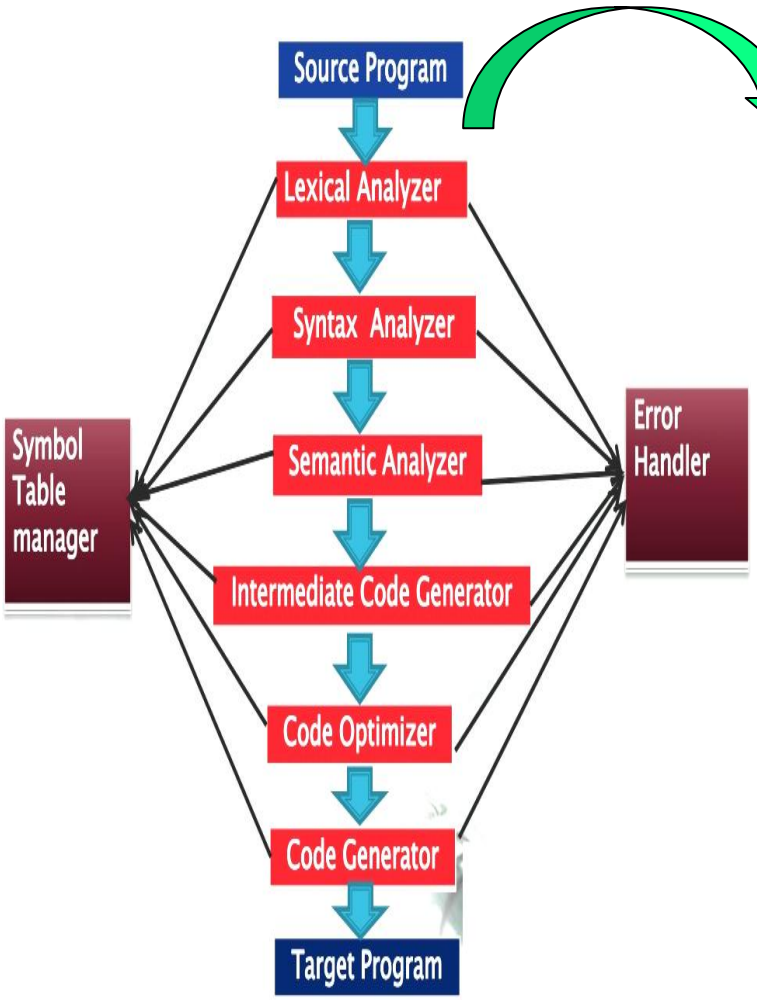


Lexical Analyzer



- Tokenization
- Remove whitespaces
- Remove comment lines
- Uses Regular expression to identify tokens

Lexical Analyzer



Input

source code:

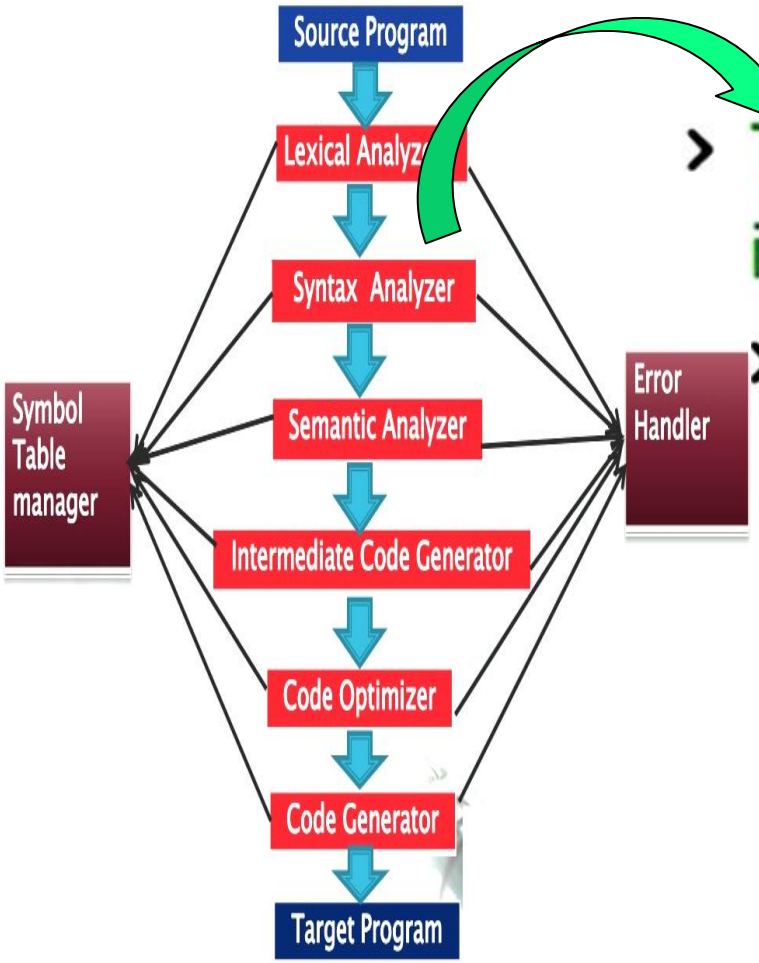
```
x = a + b * c  //this is demo
```

Output

Stream of tokens after removing whitespace and comment:

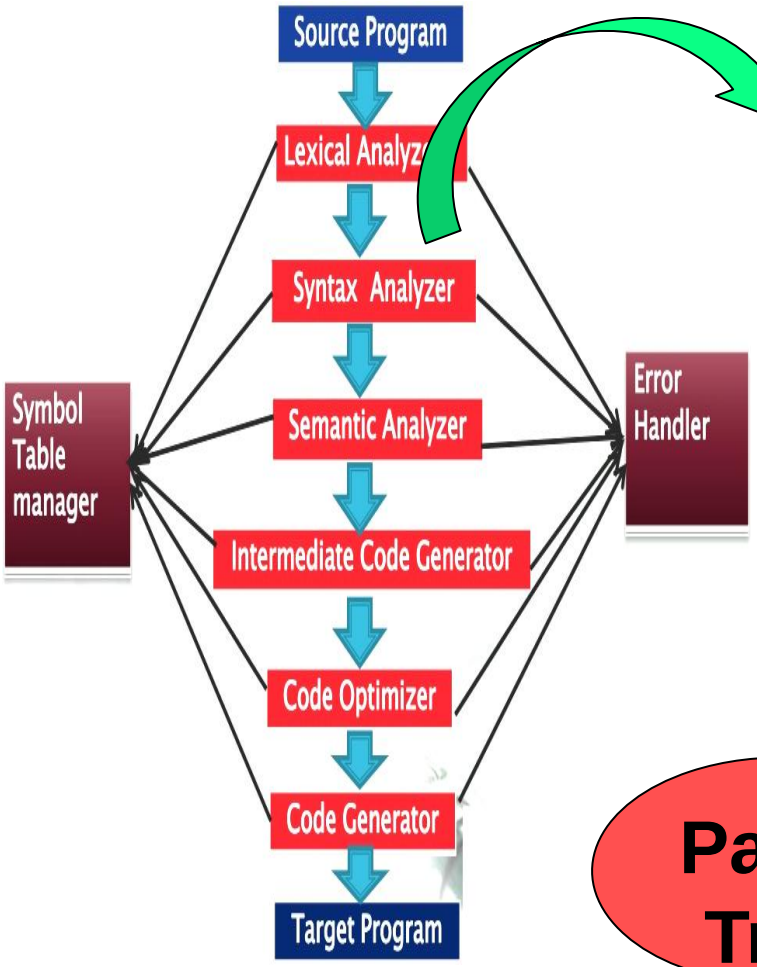
<id,1> <=> <id,2> <+> <id,3> <*> <id,4>

Syntax Analyzer



- Takes tokens and converts into parse tree
- Uses context free grammar to make parse tree
- Grammars are productions or rules which define the programming language

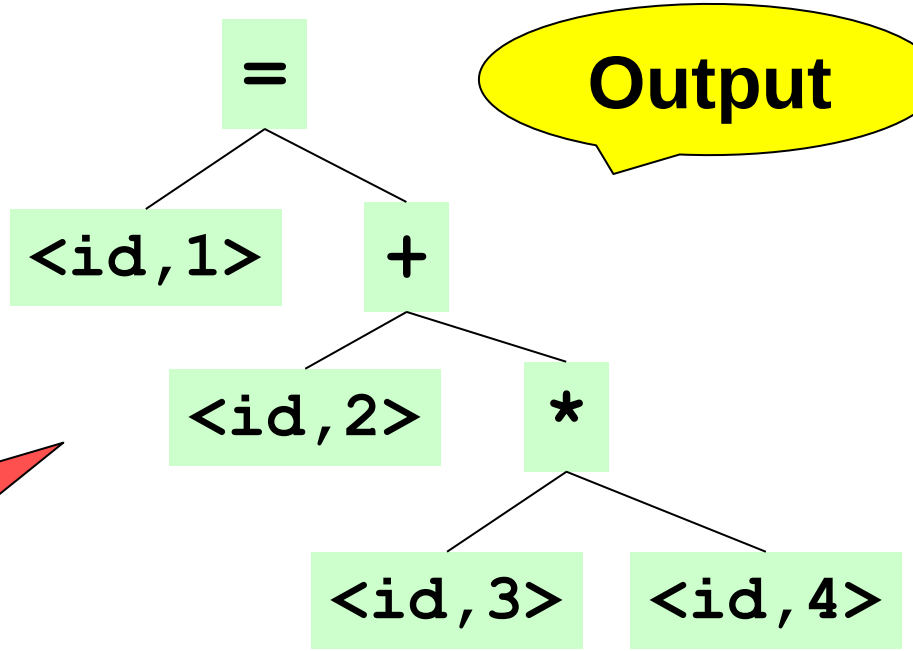
Syntax Analyzer (Parser)



Input

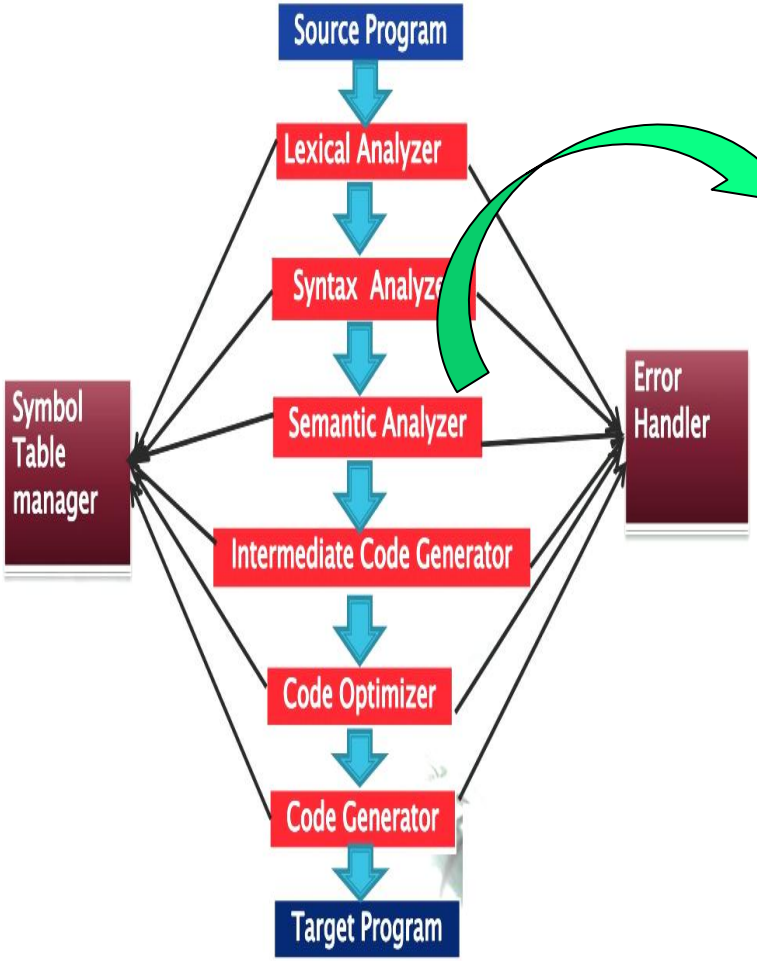
<id,1> <=> <id,2> <+> <id,3> <*> <id,4>

Output



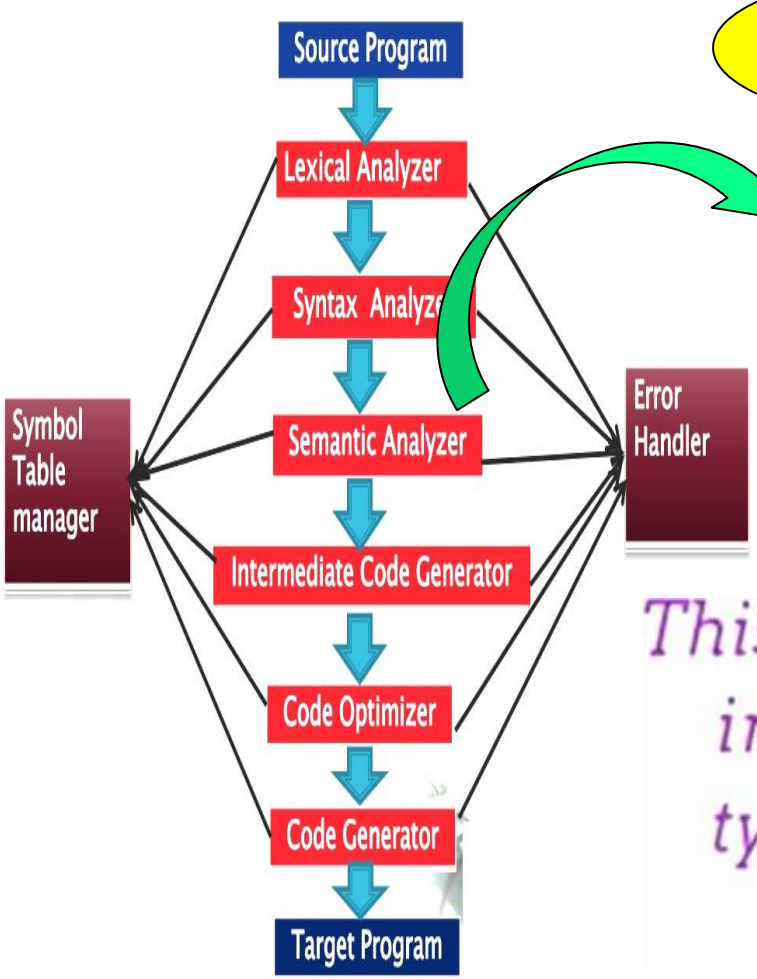
Parse Tree

Semantic Analyzer

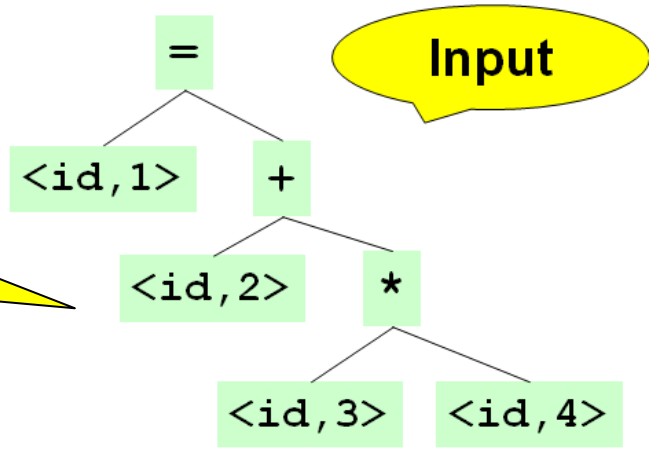


- Takes parse tree and verifies it semantically.
  - Scope resolution
    - Type checking
      - Array bound checking

Semantic Analyzer



Output

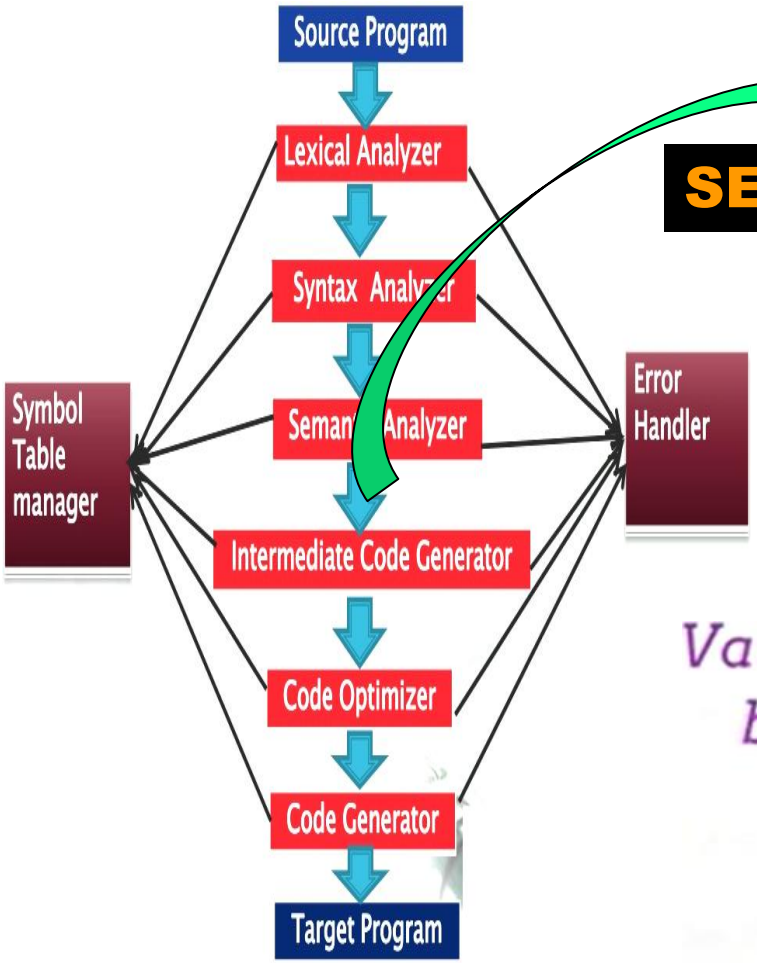


Example:

INT A = "HELLO"

This statement will not issue an error in lexical and syntax phase but a type mismatch error in semantic phase.

Intermediate Code Generator



Input

SEMANTICALLY VERIFIED PARSE TREE

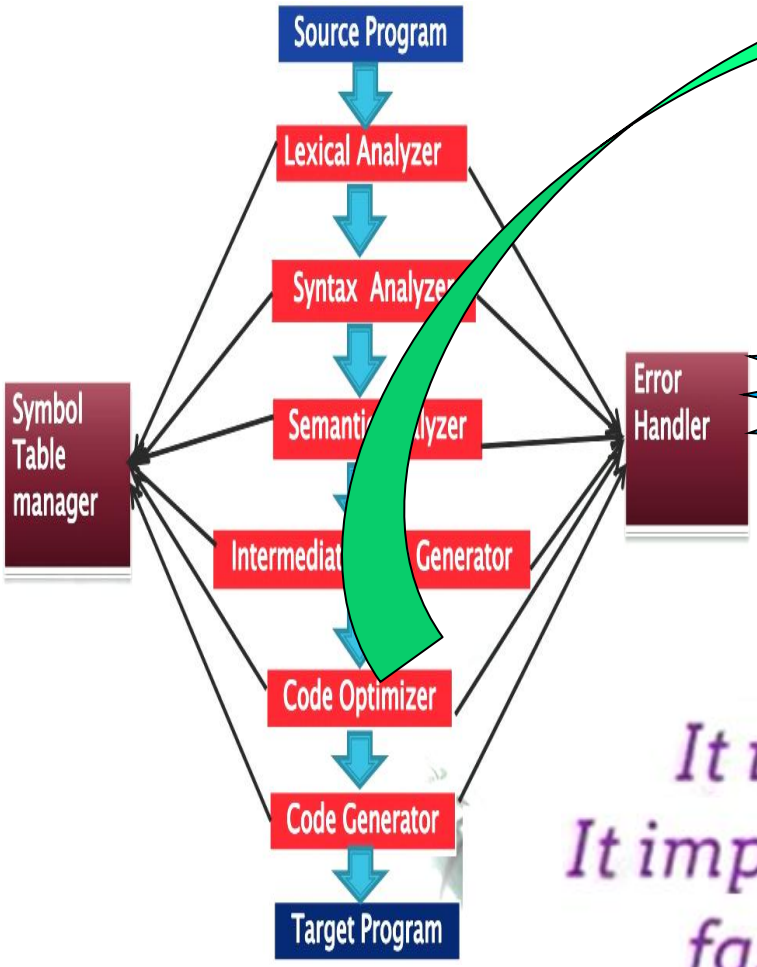
Output

```
T1=B*C
T2=T1+A
X=T2
```

Various types of intermediate code may be generated but most commonly 3 address code is generated The code generated is machin independent.

Two for operands and one for result

Code Optimizer



Intermediate Code

Input

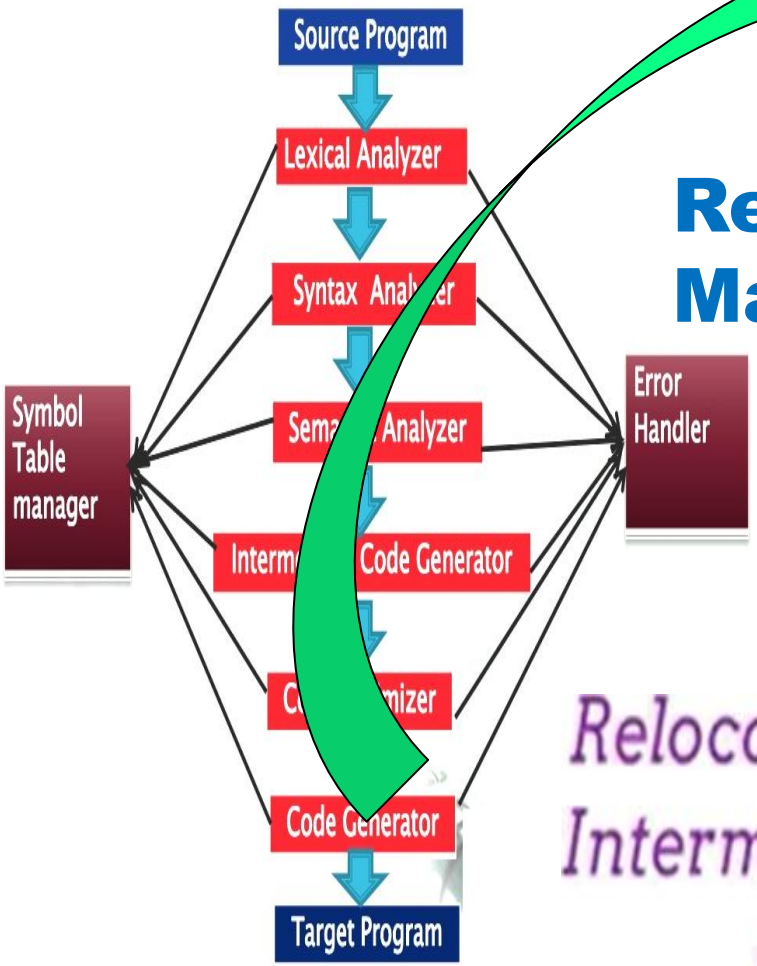
T1=B*C
T2=T1+A
X=T2

T1=B*C
X=T1+A

Output

It is machine dependent.
It reduces the number of lines.
It improves the intermediate code to faster running machine code.

Target Code Generation



Intermediate Code

Input

T1=B*C
T2=T1+A
X=T2

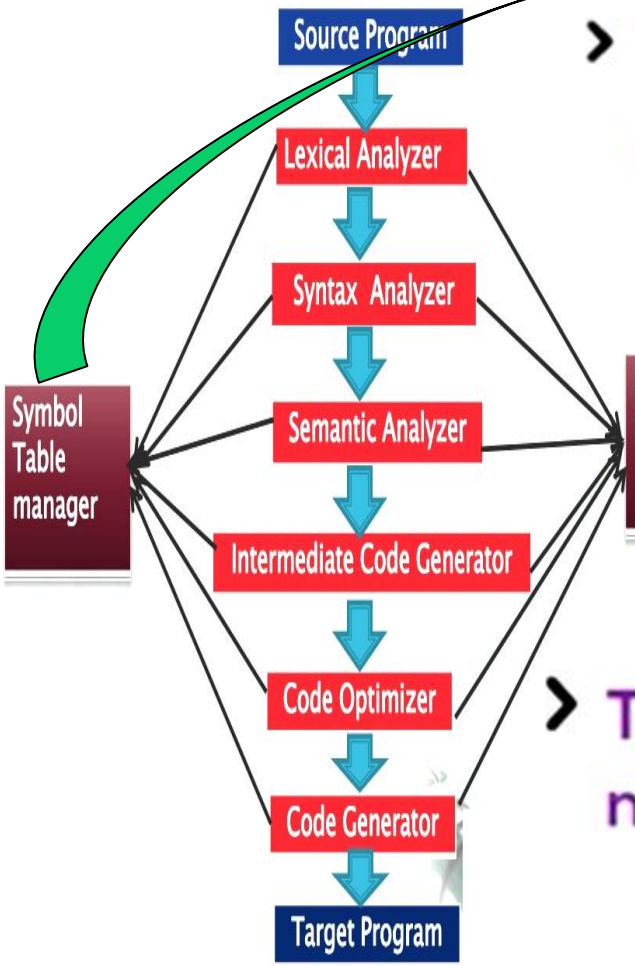
Relocatable Machine Code

mul R0, R1
add R1, R2
mov R2, X

Output

Relocatable machine code is generated. Intermediate instruction is converted to sequence of assembly code.

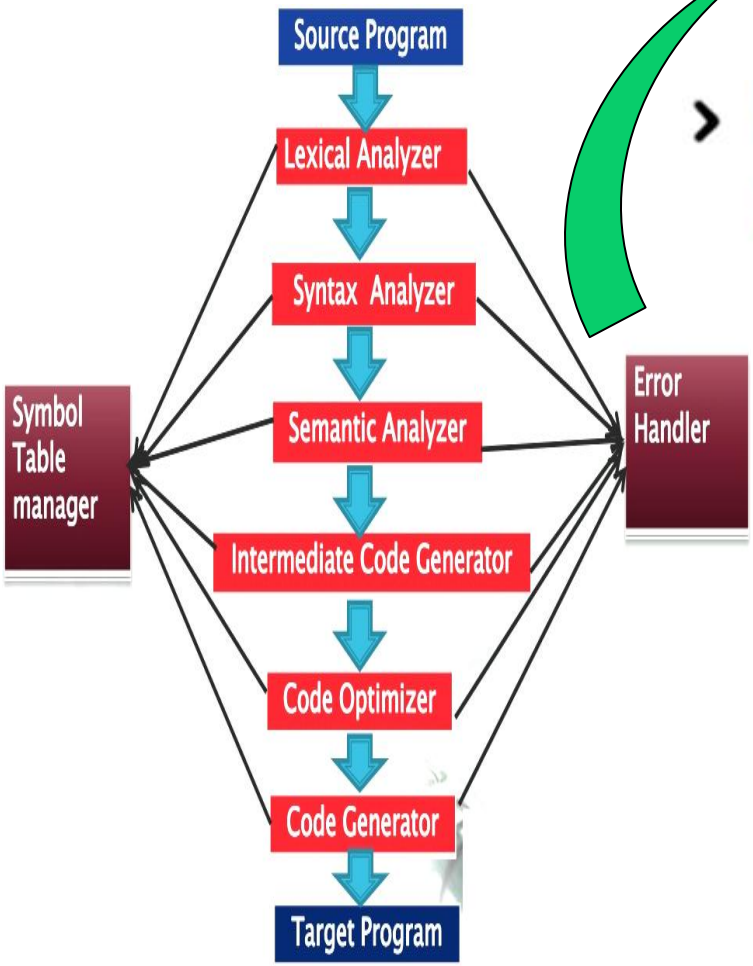
Symbol Table



- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking.
- To determine the scope of a name (scope resolution)

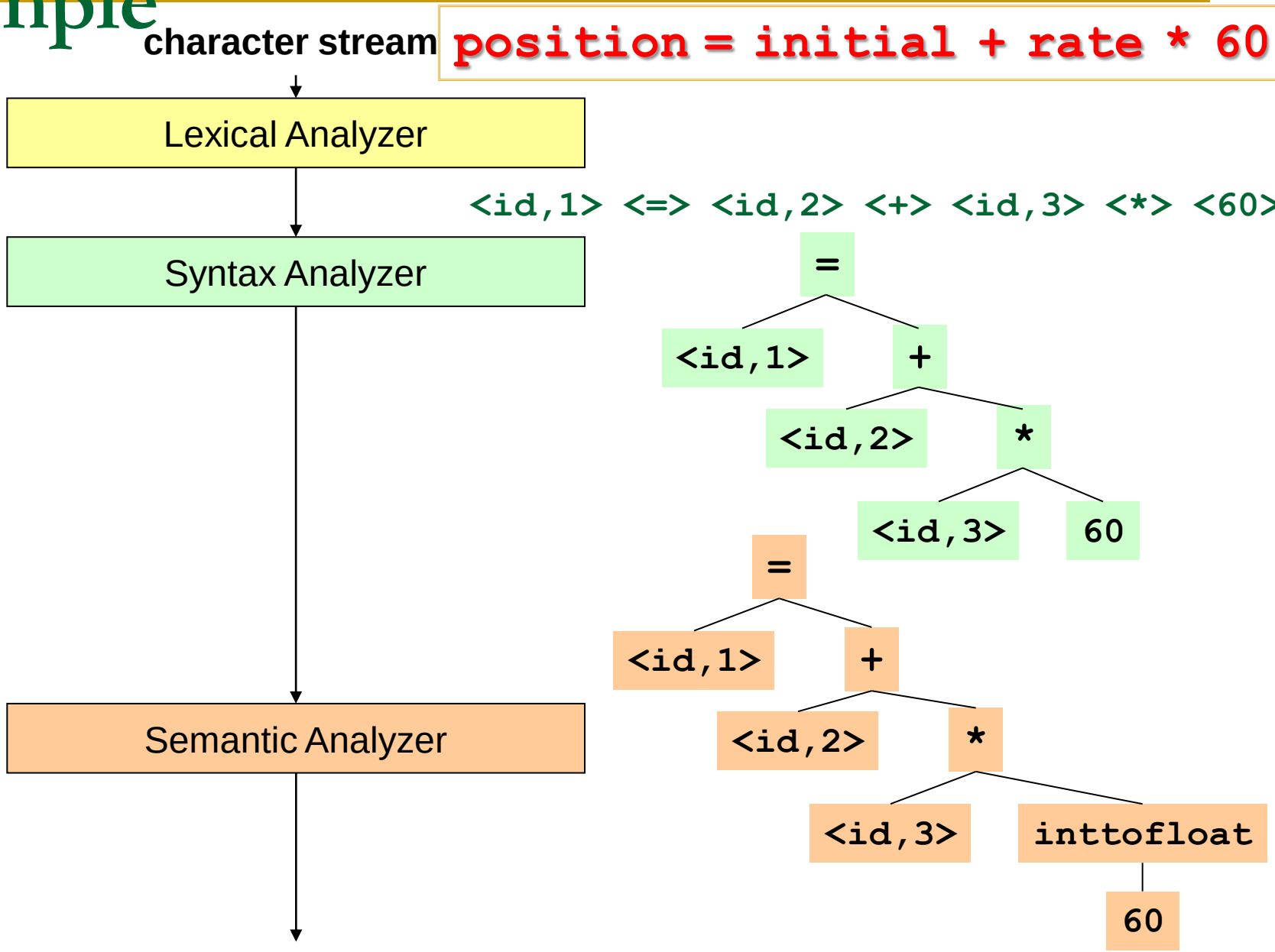
Symbol Table
scopes *symbols* *attributes*
Create AddSymbol GetAttributes OpenScope CloseScope Destroy

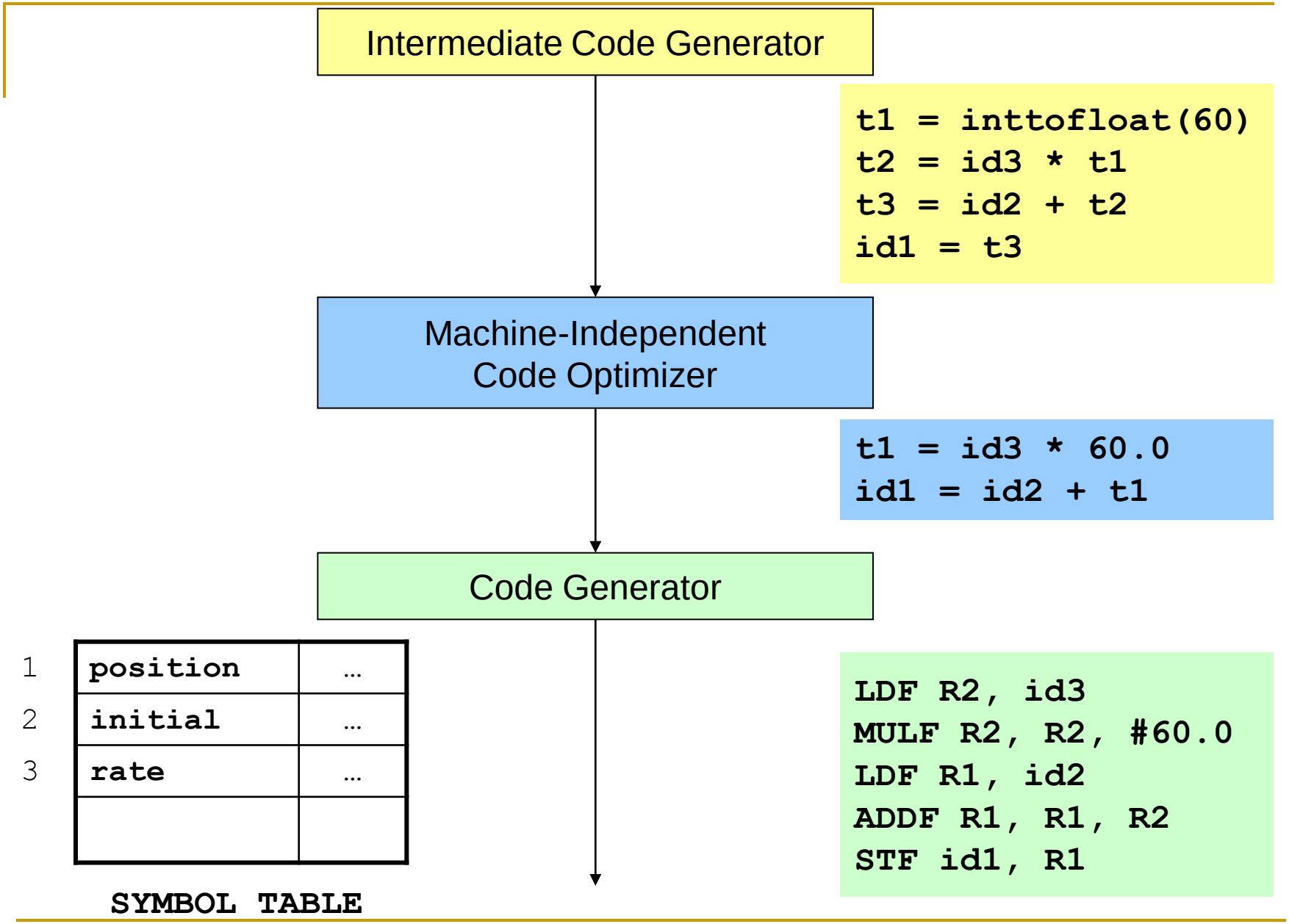
Error Handler



- Lexical: name of some identifier typed incorrectly.
- Syntactical: missing semicolon or unbalanced parenthesis.
- Semantical: incompatible value assignment.
- Logical: code not reachable, infinite loop.

Example


character stream
position = initial + rate * 60
Lexical Analyzer
<id,1> <=> <id,2> <+> <id,3> <*> <60>
Syntax Analyzer
=
<id,1>
+
<id,2>
*
<id,3>
60
=
<id,1>
+
<id,2>
*
<id,3>
inttofloat
60
Semantic Analyzer


Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Machine-Independent
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
1 position …
2 initial …
3 rate …
SYMBOL TABLE

Example

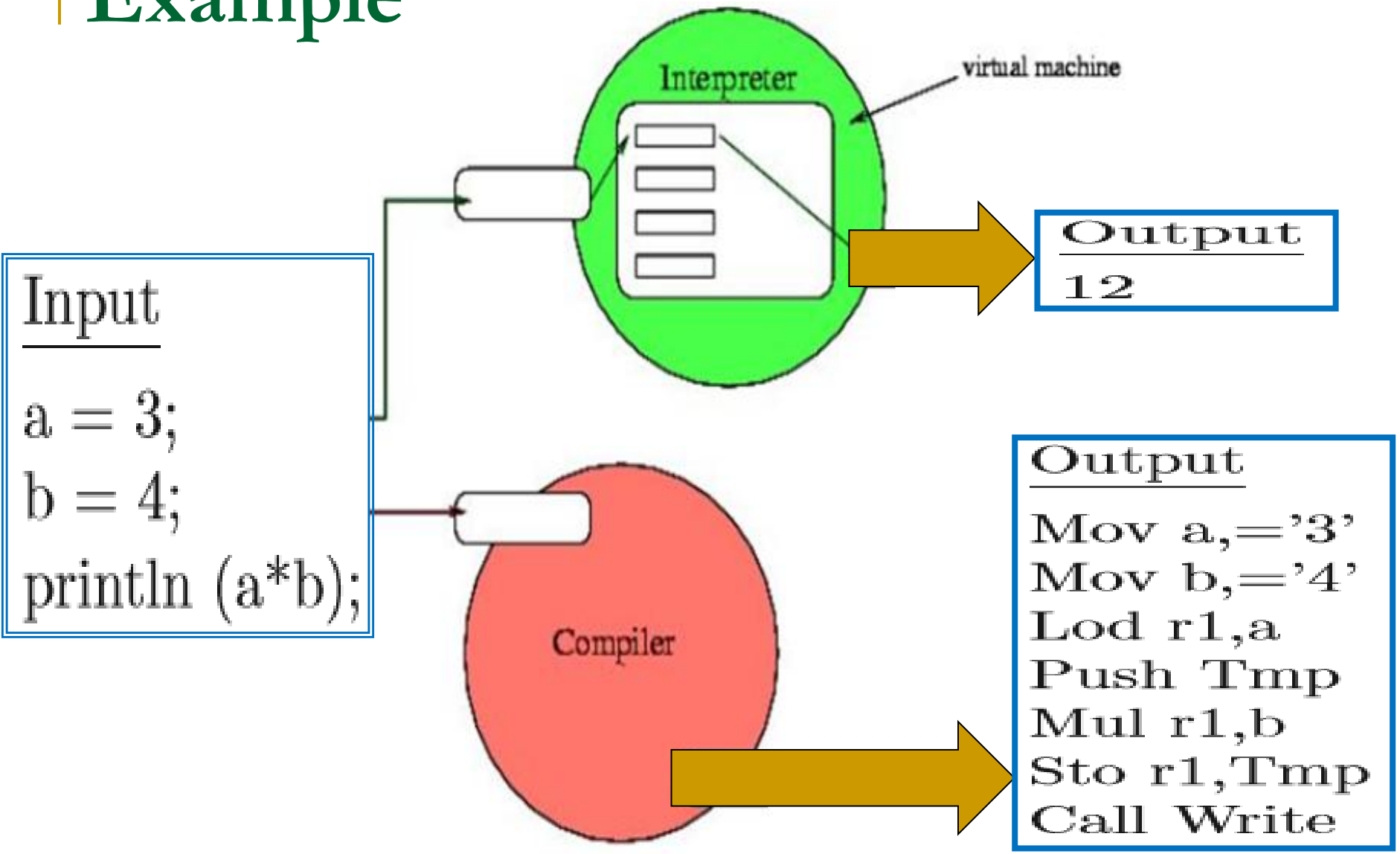
- For the following Java input string:

```
Input

a = 3;
b = 4;
println (a*b);
```

- Show the output of :
  - a Java native code compiler.
  - An Interpreter.

Example


Interpreter
virtual machine
Input
a = 3;
b = 4;
println (a*b);
Output
12
Compiler
Output
Mov a,='3'
Mov b,='4'
Lod r1,a
Push Tmp
Mul r1,b
Sto r1,Tmp
Call Write