COMPILER DESIGN

UNIT 1

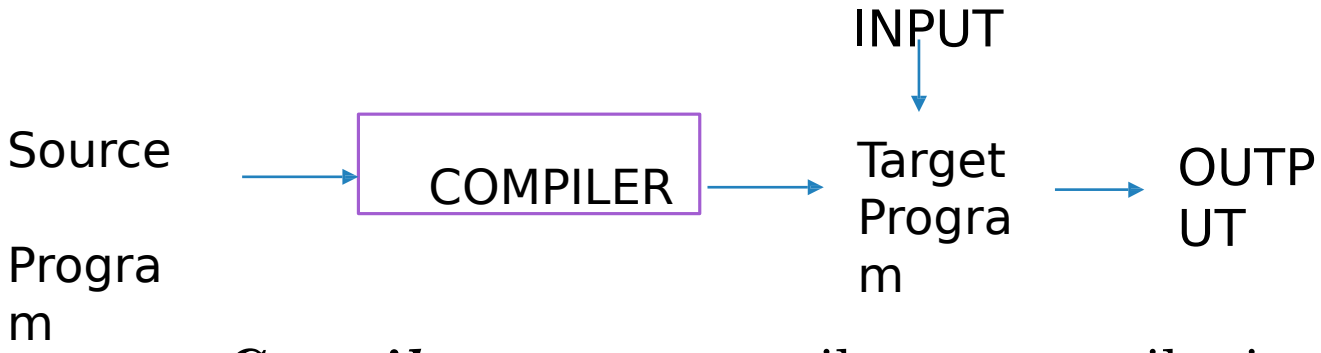
Pragya Gaur

TRANSLATOR

- A translator is a programming language processor that converts a computer program from one language to another. It takes a program written in source code and converts it into machine code. It discovers and identifies the error during translation.

COMPILER

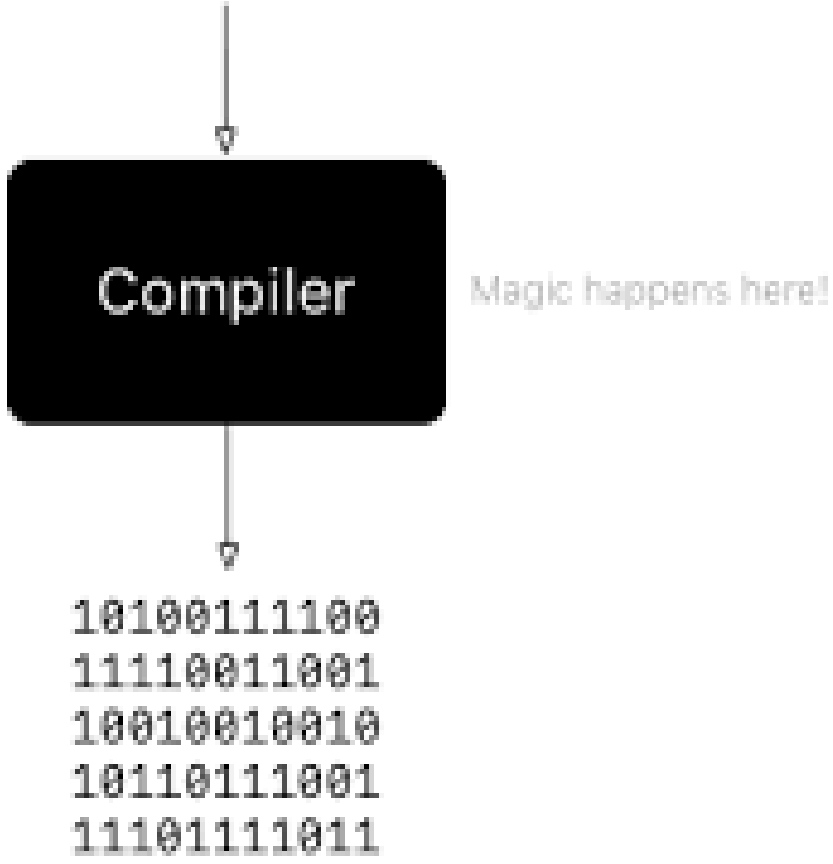
- IT'S A SOFTWARE UTILITY THAT TRANSLATED HIGH LANGUAGE CODE INTO TARGET LANGUAGE
  CODE ,AS COMPUTER DOESN'T UNDERSTAND HIGH LANGUAGE.



- ***Source-to-source Compiler*** or transcompiler or transpiler is a compiler that translates source code written in one programming language into source code of another programming language..

WHAT COMPILER DOES...

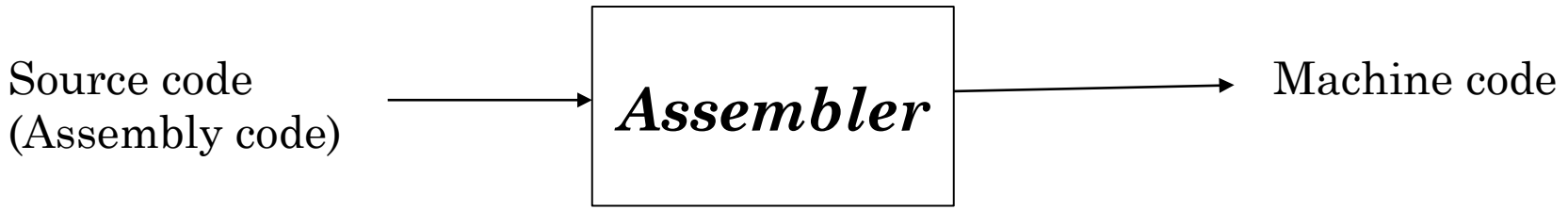
```
func greet() = {
    Console.println("Hello, World!")
}
```



Pragya Gaur

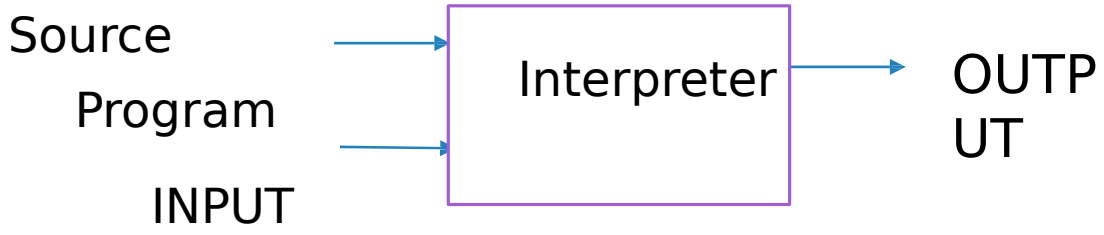
ASSEMBLER

- Assembler is a program that translate a source code written in assembly language into target code/ machine code.



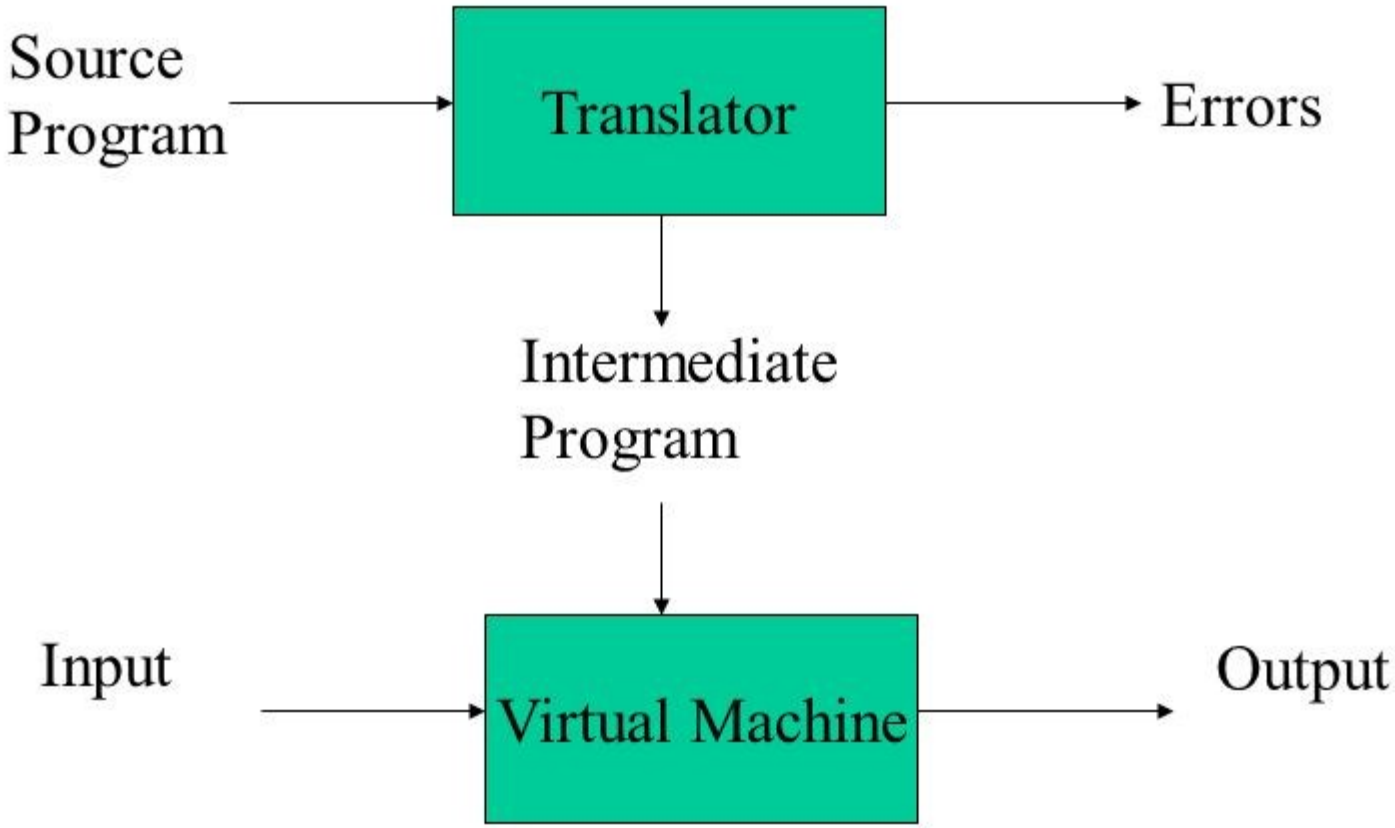
INTERPRETER

- IT’S ANOTHER SOFTWARE UTILITY THAT TRANSLATES HIGH LANGUAGE CODE INTO TARGET
  LANGUAGE CODE LINE BY LINE UNLIKE COMPILER .



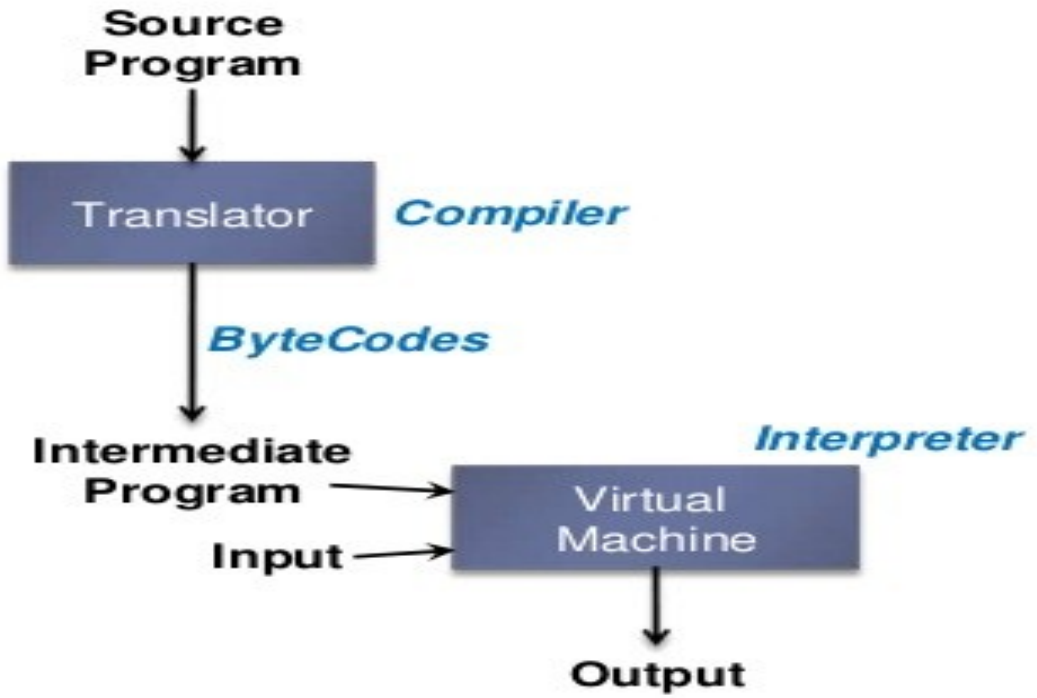
- INTERPRETER IS ONLINE MODE, IN WHICH DATA AND SOURCE PROGRAM ARE EXECUTED SIMULTANEOUSLY GIVING THE OUTPUT .NO PRE-PROCESSING IS DONE EARLIER.

HYBRID COMPILER

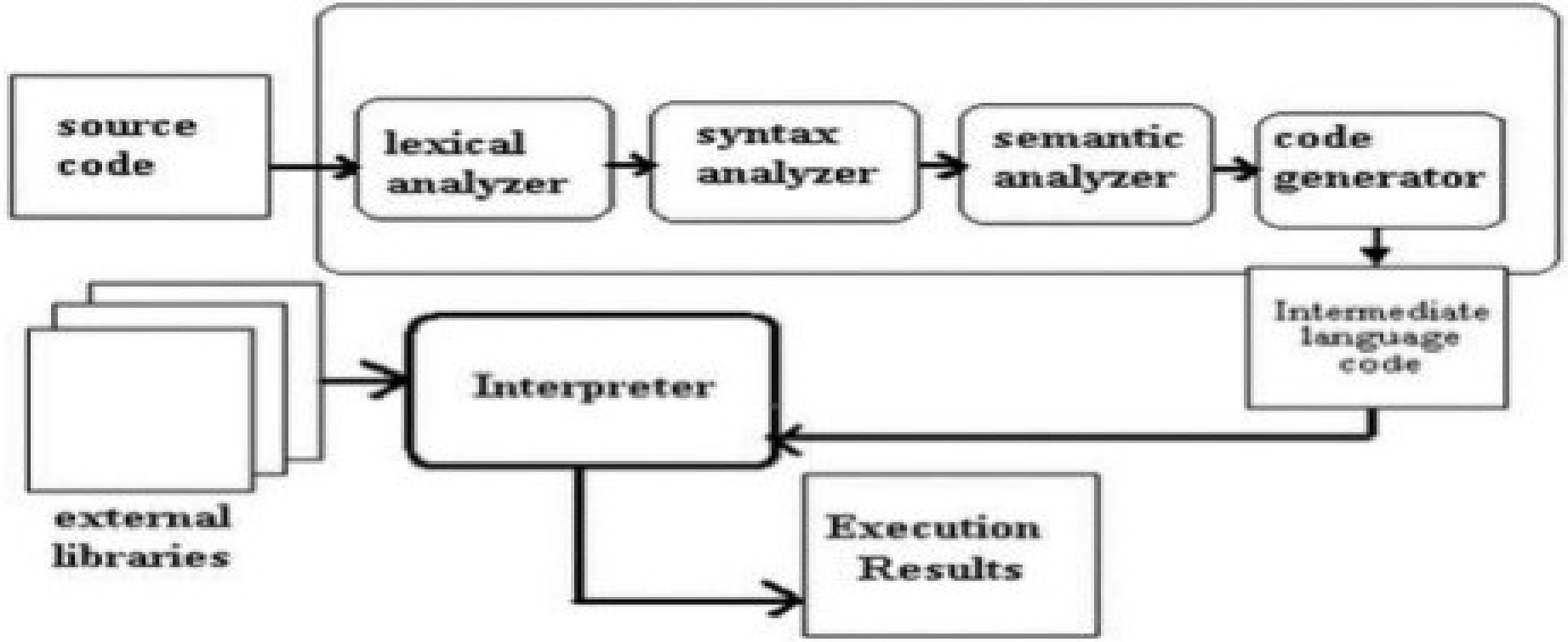


HYBRID COMPILER

- Java language processors combine compilation and interpretation
- To achieve faster processing of inputs to outputs, some java compilers, called *just-in-time* compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.



INTERNAL STRUCTURE OF HYBRID COMPILER



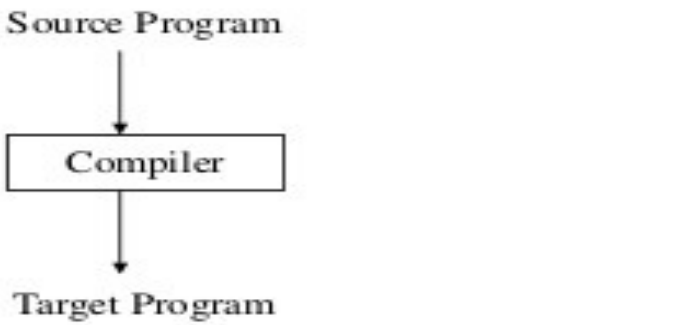
Pragya Gaur

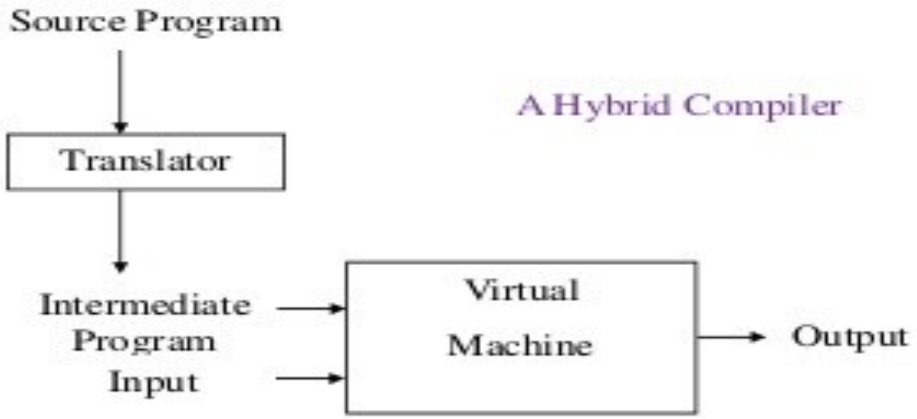
EXAMPLES

MOST LANGUAGES ARE USUALLY THOUGHT OF AS USING EITHER ONE OR THE OTHER:

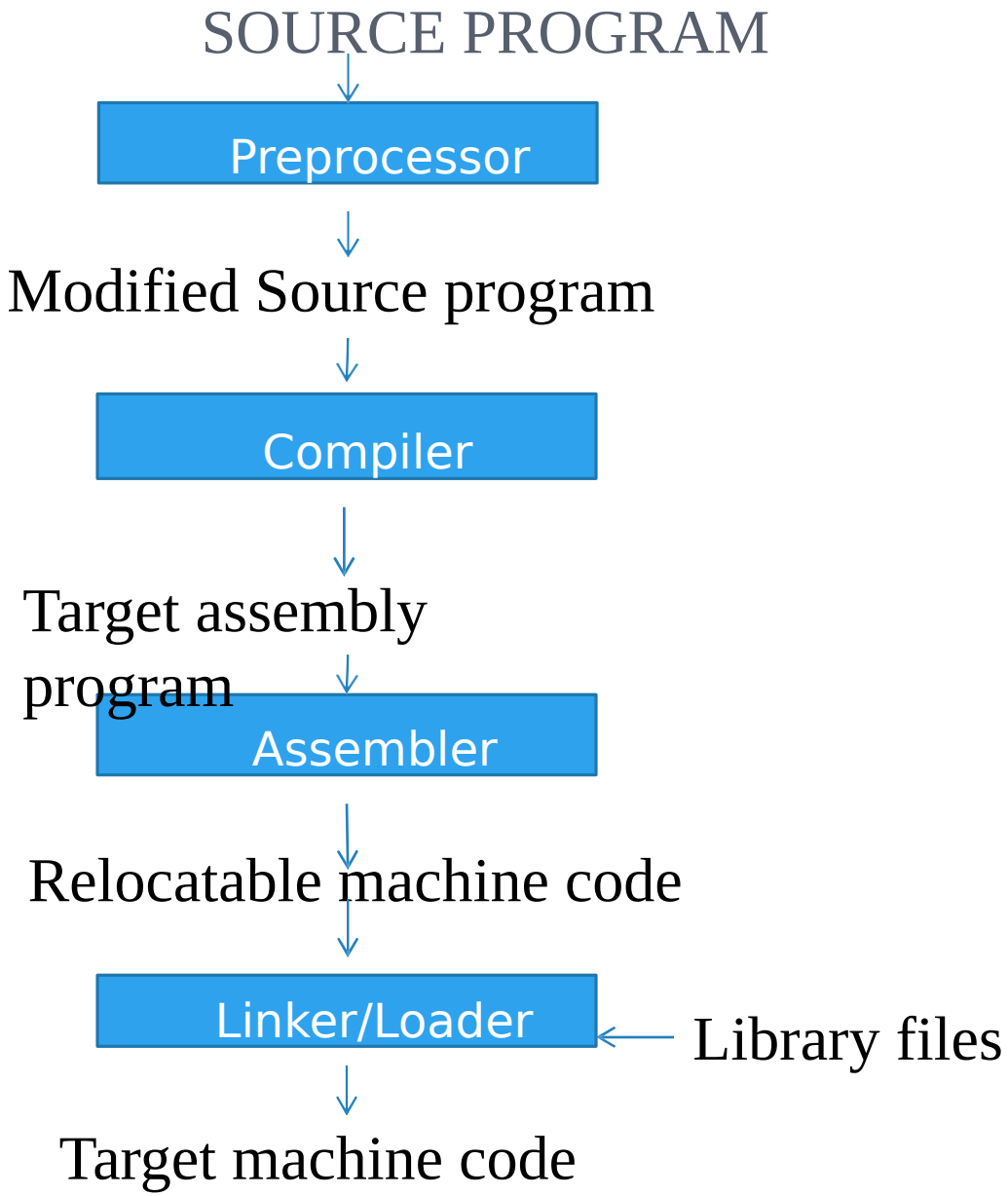
- COMPILERS: FORTRAN, COBOL, C, C++, PASCAL, PL/1
- INTERPRETERS: LISP, SCHEME, BASIC, APL, PERL, PYTHON, SMALLTALK

Compiler Vs. Interpreter

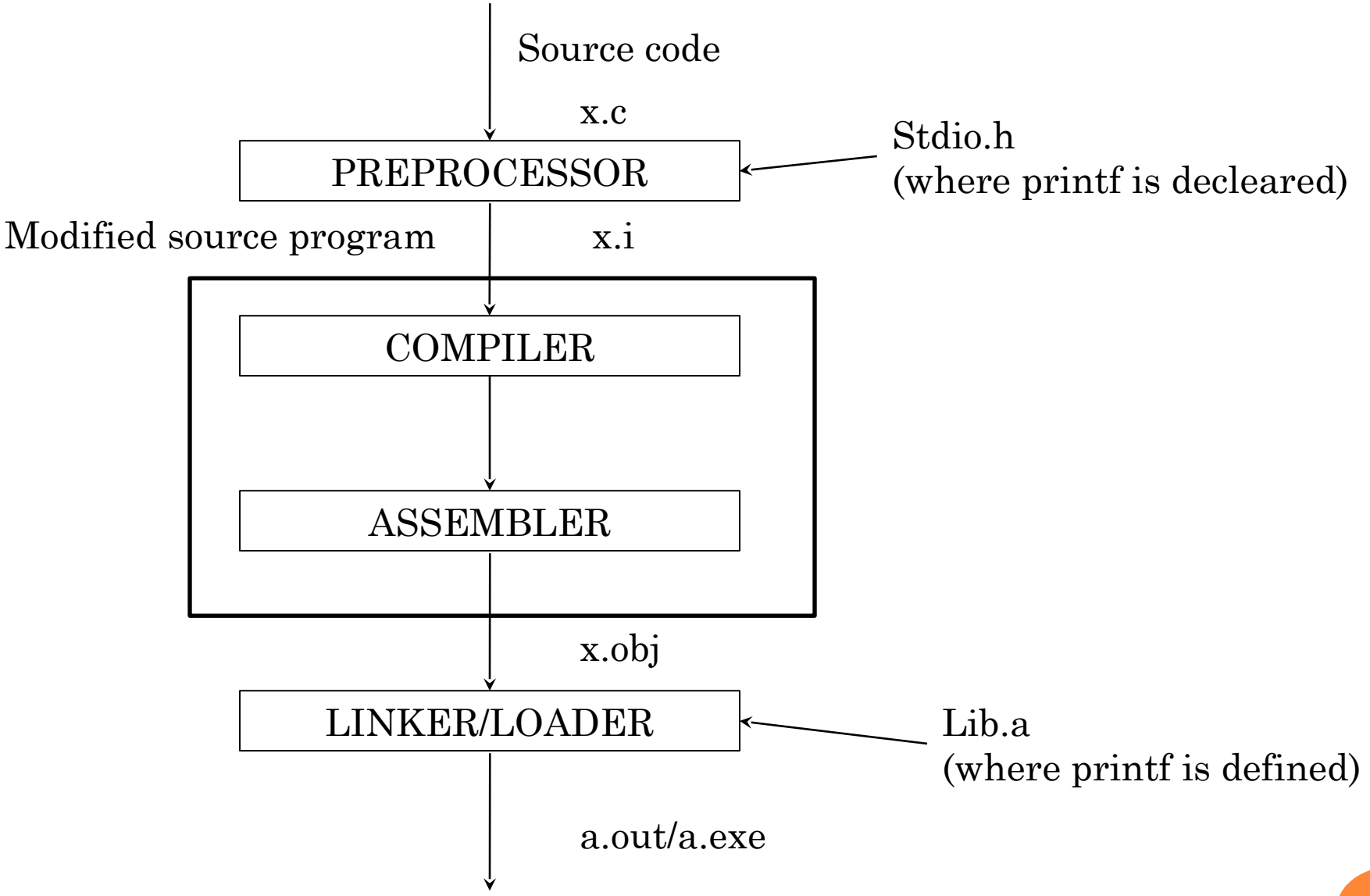




STEPS FOR LANGUAGE PROCESSING SYSTEM.



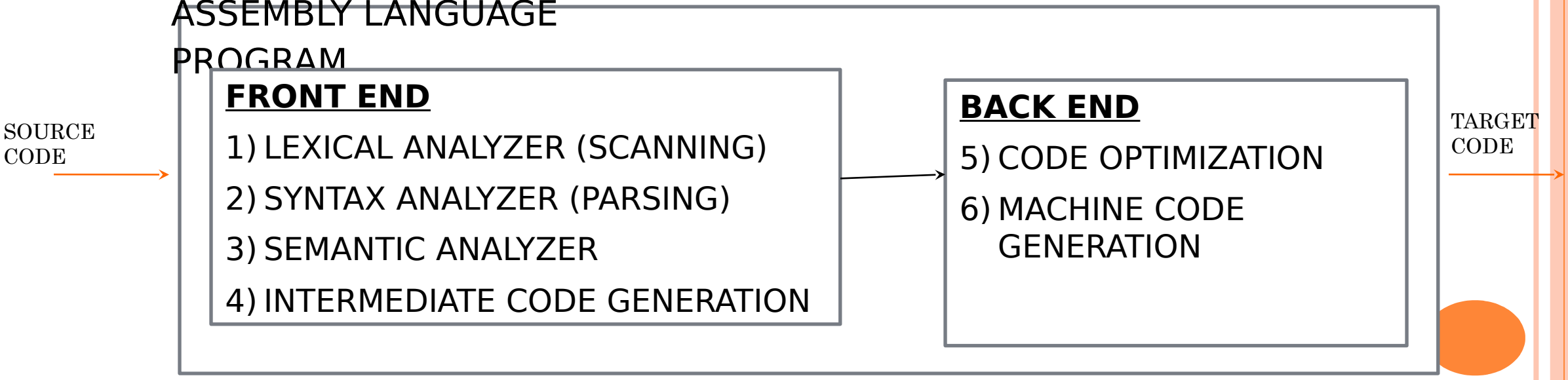
Pragya Gaur


Source code
x.c
PREPROCESSOR
Stdio.h
(where printf is decleared)
Modified source program
x.i
COMPILER
ASSEMBLER
x.obj
LINKER/LOADER
Lib.a
(where printf is defined)
a.out/a.exe

STRUCTURE OF COMPILER

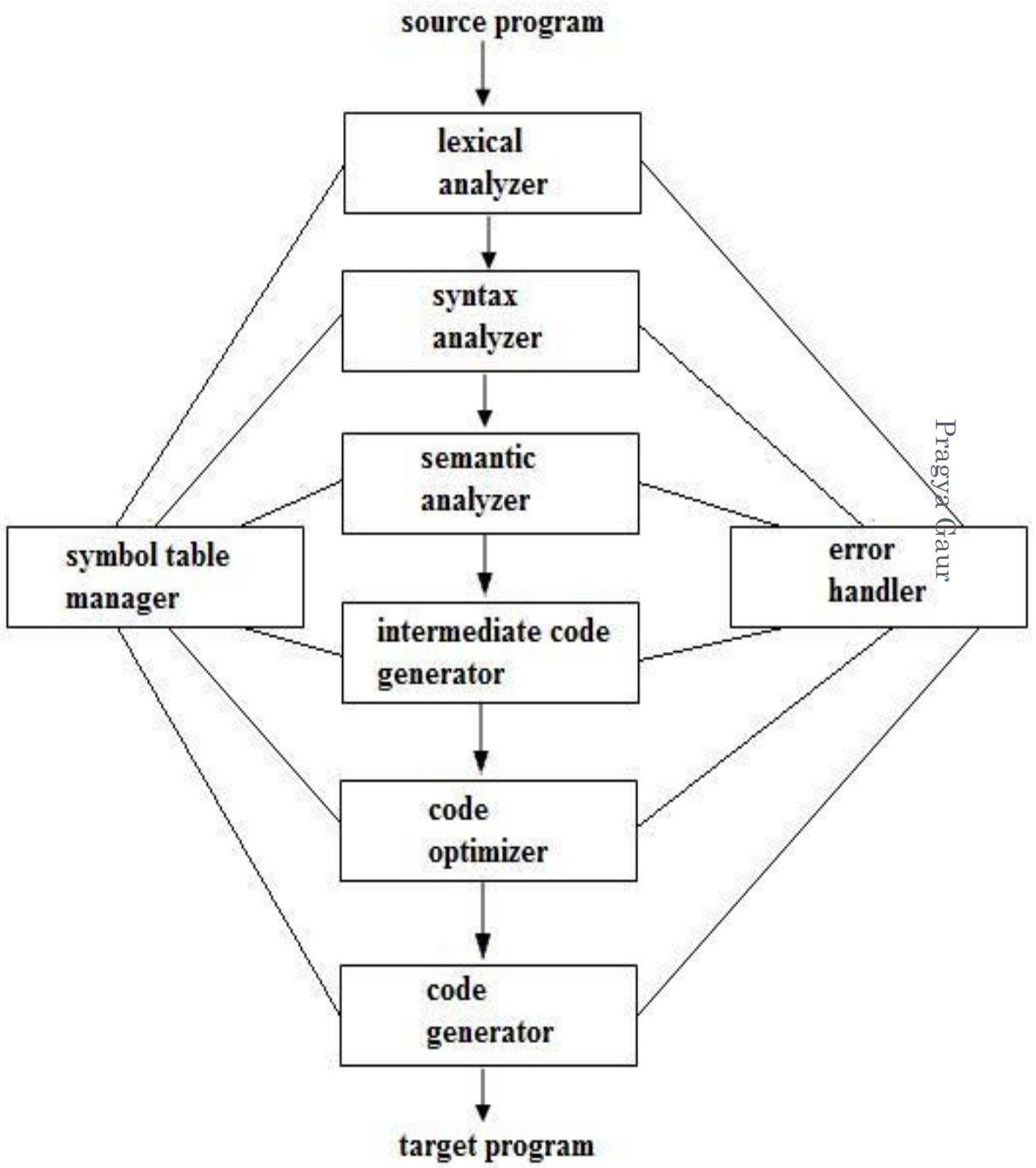
THE TRANSLATION OF INPUT FILE INTO TARGET CODE IS DIVIDED INTO 2 STAGES :

1. **FRONT END (ANALYSIS):** TRANSFORM SOURCE CODE INTO INTERMEDIATE CODE ALSO CALLED INTERMEDIATE REPRESENTATION (IR) . IT'S A MACHINE-INDEPENDENT REPRESENTATION.
2. **BACK END (SYNTHESIS):** IT TAKES IR AND GENERATES THE TARGET ASSEMBLY LANGUAGE PROGRAM



PHASES OF COMPILER

1) LEXICAL ANALYZER (SCANNING)
2) SYNTAX ANALYZER (PARSING)
3) SEMANTIC ANALYZER
4) INTERMEDIATE CODE GENERATION
5) CODE OPTIMIZATION
6) MACHINE CODE GENERATION



LEXICAL ANALYSIS/SCANNING

- READS THE STREAM OF CHARACTERS MAKING UP THE SOURCE PROGRAM AND GROUP THE
  CHARACTERS INTO MEANINGFUL SEQUENCES CALLED LEXEMES.
- LEXEME ----□ TOKEN

  < TOKEN-NAME, ATTRIBUTE-VALUE>
- TOKEN NAME – IS AN ABSTRACT SYMBOL USED DURING SYNTAX ANALYSIS.
- ATTRIBUTE VALUE – POINTS TO AN ENTRY IN THE SYMBOL TABLE FOR THIS TOKEN.

LEXICAL ANALYSIS

- POSITION = INITIAL + RATE * 60

  <ID,1> < = > <ID,2> < +> <ID,3> < *><60>

- a=b+c*10

  <id,1> <=> <id,2> <+> <id,3> <*> <10>

  1 2 3 4 5 6 7

- printf(“compiler Design”)

  <printf> <(> < “compiler Design”> <)>

  1 2 3 4

Pragya Gaur

SYNTAX ANALYSIS/PARSING

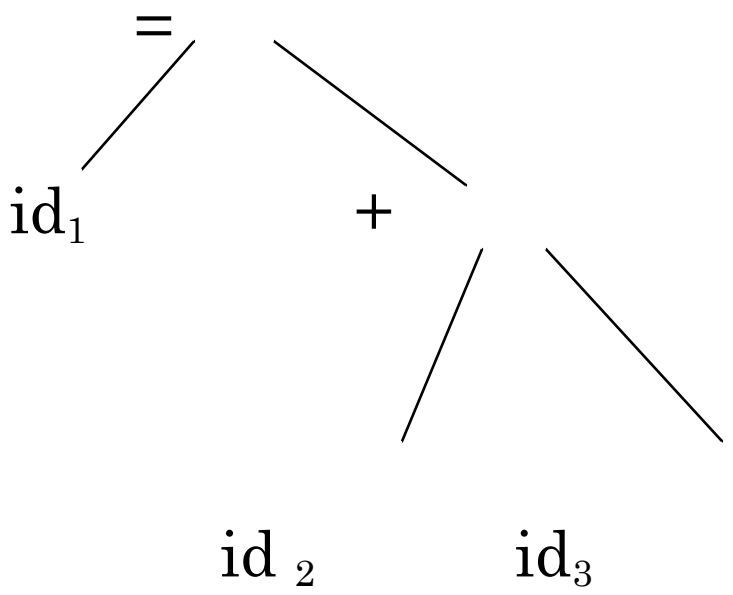
- RECOGNIZES “SENTENCES” IN THE PROGRAM USING THE SYNTAX OF THE LANGUAGE
- CREATES TREE LIKE STRUCTURE FROM TOKENS (SYNTAX TREE)
- NODE REPRESENTS **OPERATION**
- CHILDREN REPRESENTS **ARGUMENTS**
- REPRESENTS THE SYNTACTIC STRUCTURE OF THE PROGRAM, HIDING A FEW DETAILS THAT ARE IRRELEVENT TO LATER PHASES OF COMPILATION.

Pragya Gaur

SYNTAX ANALYSIS/PARSING

a = b + c

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

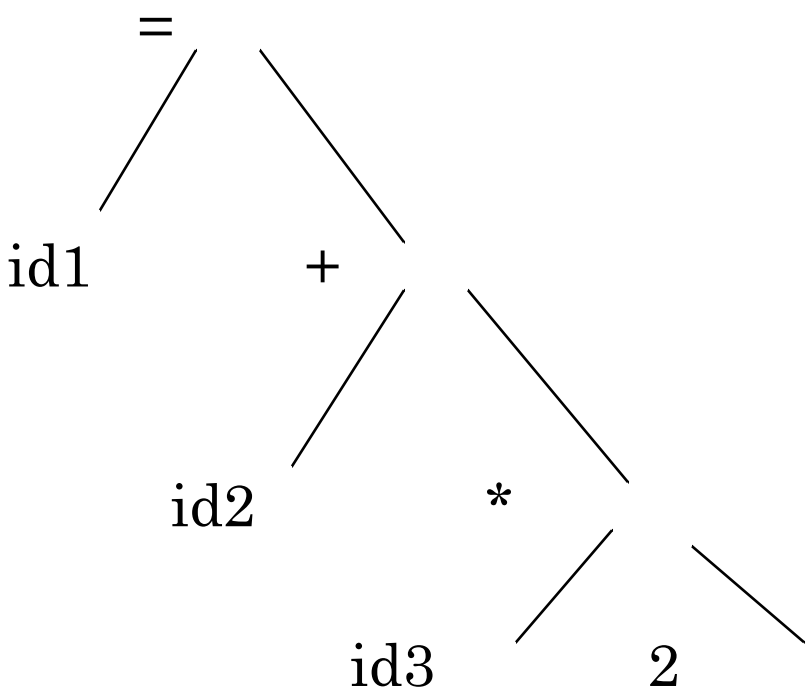


SEMANTIC ANALYSIS

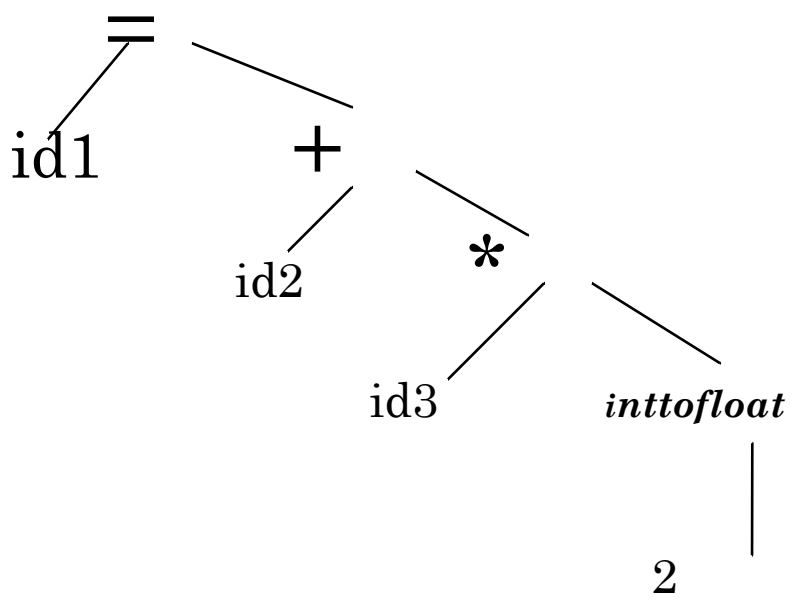
- INFERS INFORMATION ABOUT THE PROGRAM USING THE SEMANTICS OF THE LANGUAGE
- USES SYNTAX TREE AND INFO. IN SYMBOL TABLE TO CHECK FOR SEMANTIC CONSISTENCY.
- GATHERS TYPE INFO. AND SAVES IT IN EITHER THE SYNTAX TREE OR SYMBOL TABLE FOR USE IN ICG
- TYPE CHECKING – CHECKS THAT EACH OPERATOR HAS MATCHING OPERANDS. E.G ARRAY INDEX SHOULD BE INTEGER.
- TYPE CONVERSIONS CALLED COERCIONS
  - BINARY ARITHMETIC OPERATOR (INT OR FLOAT)
  - IF 6+7.5, THEN CONVERT 6 TO 6.0

SEMANTIC ANALYSIS

- a=b+c*2



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



INTERMEDIATE CODE GENERATION

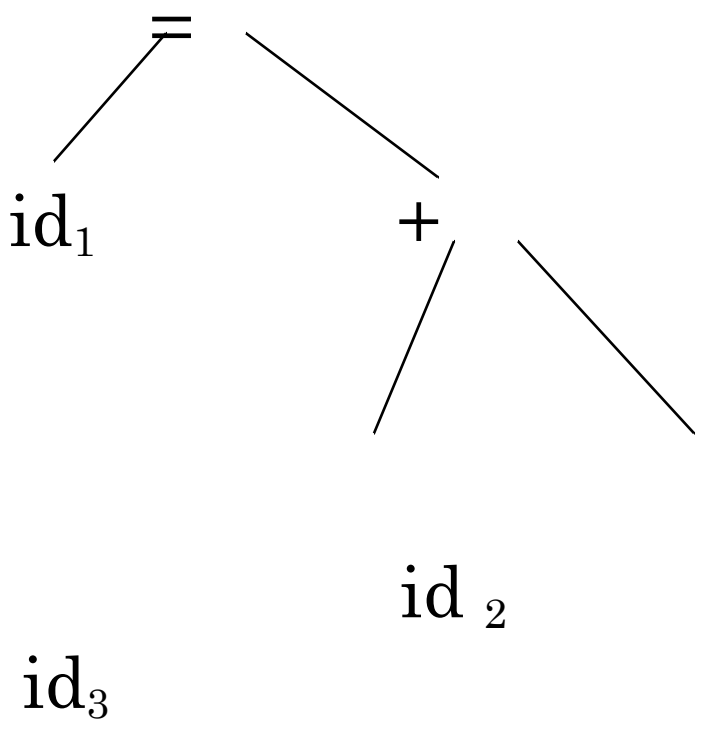
- GENERATES “ABSTRACT” CODE BASED ON THE SYNTACTIC STRUCTURE OF THE PROGRAM AND THE SEMANTIC INFORMATION FROM PREVIOUS PHASE .
- VARIOUSMETHODS AREUSED TO GENERATE INTERMEDIATE CODE
  - SYNTAX TREE
  - POSTFIX NOTATION
  - THREE ADDRESS CODE

Pragya Gaur

INTERMEDIATE CODE GENERATION

- SYNTAX TREE

  a=b+c



- POSTFIX NOTATION

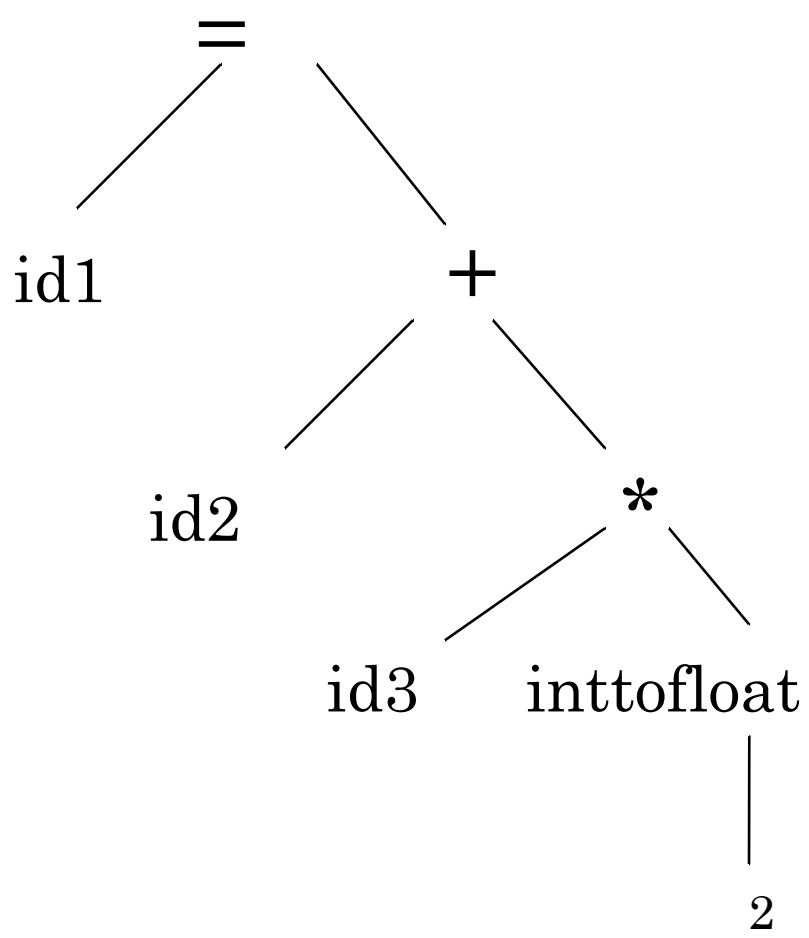
  a=b+c

  ⇩

  abc+=

Pragya Gaur

INTERMEDIATE CODE GENERATION



- T1 = INTTOFLOAT(2)
- T2 = ID3 * T1
- T3 = ID2 + T2
- ID1 = T3

Pragya Gaur

CODE OPTIMIZATION

- REFINES THE GENERATED CODE USING A SERIES OF OPTIMIZING TRANSFORMATIONS.
- Eg: REMOVING DEAD CODE .

  REDUCING ITERATIONS AND LOOPS ETC..
- APPLY A SERIES OF TRANSFORMATIONS TO IMPROVE THE TIME AND SPACE EFFICIENCY OF THE GENERATED CODE.
- PEEPHOLE OPTIMIZATIONS: GENERATE NEW INSTRUCTIONS BY COMBINING/EXPANDING ON A SMALL NUMBER OF CONSECUTIVE INSTRUCTIONS.
- GLOBAL OPTIMIZATIONS: REORDER, REMOVE OR ADD INSTRUCTIONS TO CHANGE THE STRUCTURE OF GENERATED CODE.

Pragya Gaur

INTERMEDIATE CODE GENERATION

- T1 = INTTOFLOAT(2)
- T2 = ID3 * T1
- T3 = ID2 + T3
- ID1 = T3

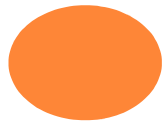
- T1 = id3 * 2.0
- id1 = id2 + T1

Pragya Gaur

CODE GENEARTION

- MAP INSTRUCTIONS IN THE INTERMEDIATE CODE TO SPECIFIC MACHINE INSTRUCTIONS.
- SUPPORTS STANDARD OBJECT FILE FORMATS.
- GENERATES SUFFICIENT INFORMATION TO ENABLE SYMBOLIC DEBUGGING.

- IR -> CG -> TARGET LANGUAGE (E.G MACHINE CODE)
- REGISTERS AND MEMORY LOCATIONS ARE SELECTED FOR EACH VARIABLE USED BY THE PROGRAM.
- LDF R2, ID 3 #60.0
- MULF R2, R2, R1, ID 2 R R 2
- LDF R1, ID 2 R R 2
- ADDF R1, R1, R2 – F – FLOATING POINT NUMBERS # - IMMEDIATE CONST.
- STF ID1, R1



CODE GENEARTION

- T1 = id3 * 2.0
- id1 = id2 + T1

Machine Code:

- LDF R2, ID3, #60.0
- MULF R2, R2, #2.0
- LDF R1, ID2, R1
- ADDF R1, R1, R2
- STF ID1, R1

R1,R2 – REGISTERS

F – FLOATING POINT NUMBERS

- IMMEDIATE CONST.

Pragya Gaur

SYMBOL TABLE

- **SYMBOL TABLE** – DATA STRUCTURE WITH A RECORD FOR EACH IDENTIFIER AND ITS ATTRIBUTES
- ALL THE PHASES ARE CONNECTED TO THE SYMBOL TABLE.
- ATTRIBUTES INCLUDE STORAGE ALLOCATION, TYPE, SCOPE, ETC
- ALL THE COMPILER PHASES INSERT AND MODIFY THE SYMBOL TABLE

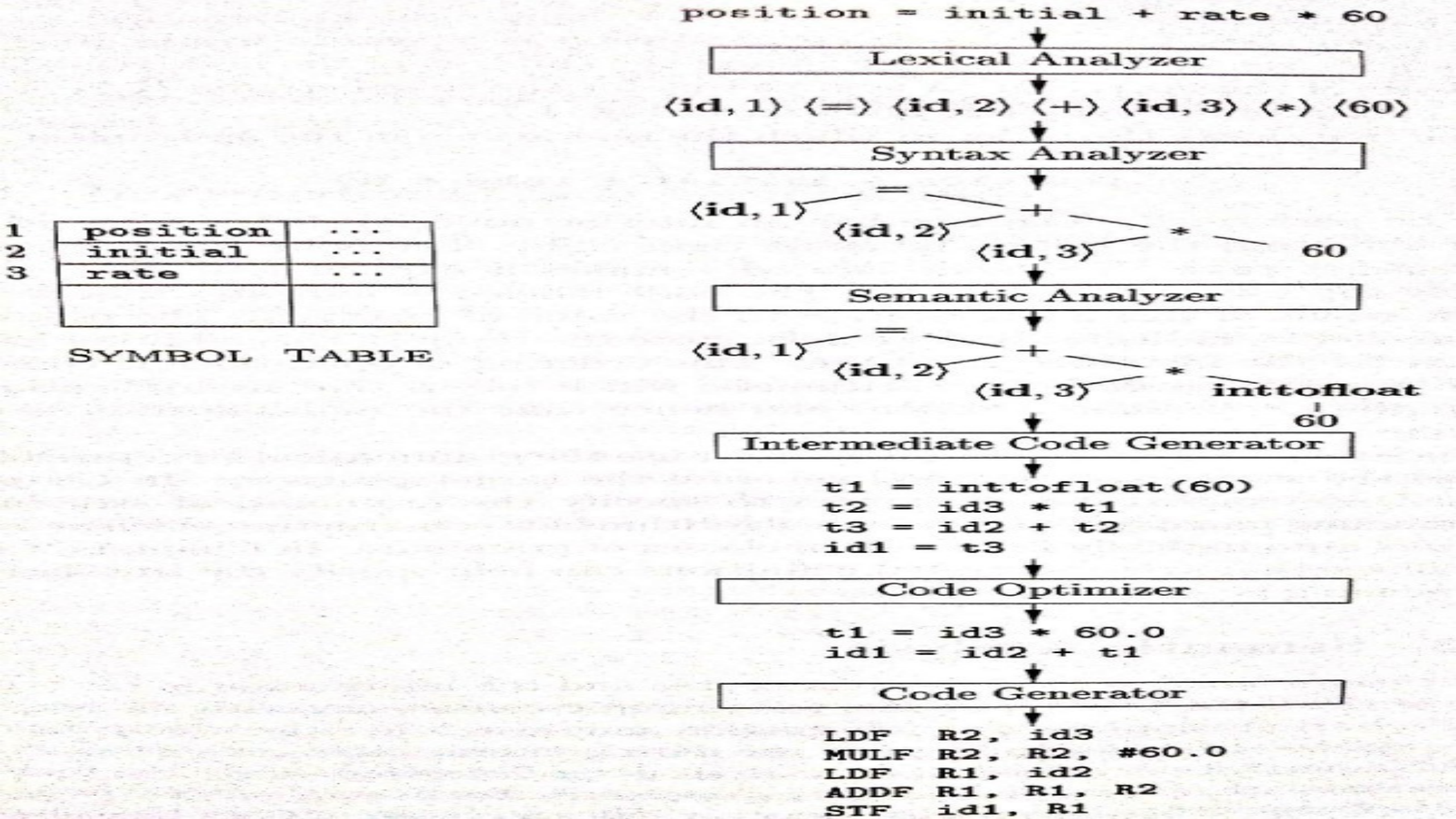
a = b + c

a

b

c

1	a
2	b
3	c


position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
Semantic Analyzer
inttofloat
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
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