SOT
SCHOOL OF TECHNOLOGY

COMPILER DESIGN

Presented by:

Dr. Shivangi K. Surati

Assistant Professor,
Department of Computer Science and Engineering,
School of Technology,
Pandit Deendayal Energy University

Subject Overview

- Introduction to Compiler
- Lexical Analysis
- Parsing Theory
- Type Checking
- Run Time Environments
- Intermediate Code Generation
- Code Optimization
- Code Generation

- Reference book:
  - A. V. Aho, R. Sethi and J. D. Ullman, “Compilers, Principles, Techniques and Tools”, Pearson

Practical

- In C, C++, Java programming
- Using LEX tool
- Using YACC tool

Introduction of Compiler

- Overview of the Translation Process- A Simple Compiler, Difference between interpreter, assembler and compiler
- Overview and use of linker and loader
- Types of Compiler
- Analysis of the Source Program
- The Phases of a Compiler
- Cousins of the Compiler, The Grouping of Phases
- Front-end and Back-end of compiler
- Pass structure
- A simple one-pass compiler: overview

Programming language

- Programming language and its formation
- Input methodology
  - Input domain
  - From where to take input
  - How to take input
  - Any specific format?
- Rules followed by the programming language

  For example, I (I | d)*

  where I : A-Z | a-z | _

  d: 0-9
- Output methodology
  - Where to display output
  - Format of output

Single underscore (_)?

- In C language, only single _ (underscore) or only multiple _ (underscores) are allowed as an identifier.
- EX: int _; //allowed

  int ____; //Multiple underscores allowed
- In Java, only single _ (underscore) is considered as keyword since version 1.9. Multiple underscores are allowed as an identifier.
- EX: int _; //Not allowed, error

  int __ ; // Allowed

  __ = 5;

  System.out.println(__); // generates output 5

Language Processors

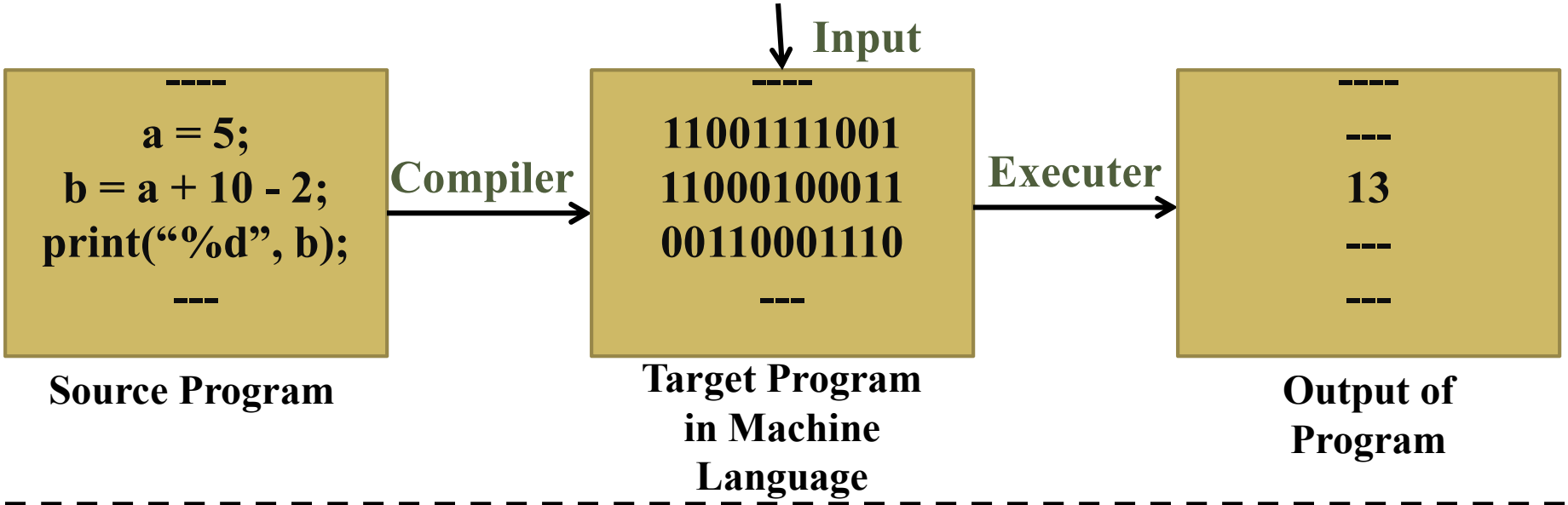
- Preprocessors
- Compilers
- Assemblers
- Interpreters
- Migrators
- Linker

Language Translators

Compilers

- Source program – In natural language like English
- Machine – can understand machine language (Binary)
- High level Source program -----→ Machine language
- Compiler
  - converts the source code into lower level programming language (assembly language) or into Machine language
  - Generates errors and warning messages in the source code
  - Creates symbol tables that stores variable names and function names along with different attributes

Compilers…


Input
---
a = 5;
b = a + 10 - 2;
print("%d", b);

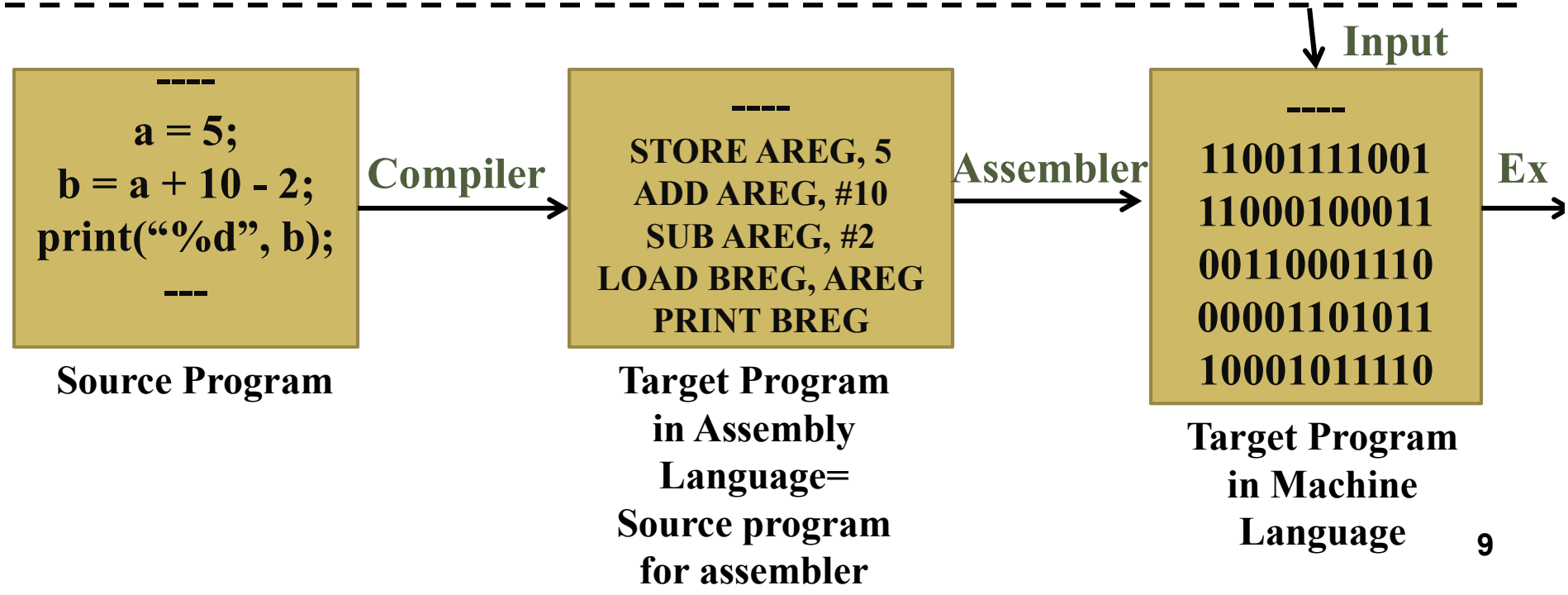
Compiler

11001111001
11000100011
00110001110

Executer

13

Source Program
Target Program in Machine Language
Output of Program

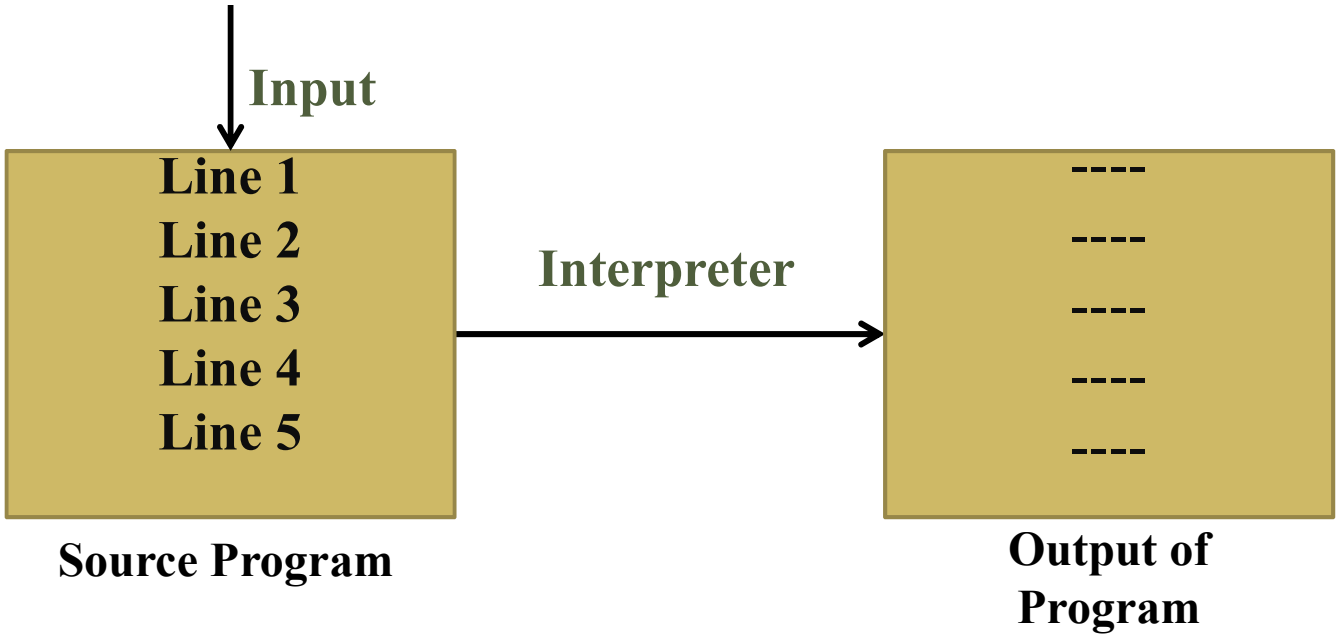

Input
---
a = 5;
b = a + 10 - 2;
print("%d", b);

Compiler

STORE AREG, 5
ADD AREG, #10
SUB AREG, #2
LOAD BREG, AREG
PRINT BREG
Assembler

11001111001
11000100011
00110001110
00001101011
10001011110
Ex
Source Program
Target Program in Assembly Language= Source program for assembler
Target Program in Machine Language

Interpreters



- Interprets the code line by line
  - Fetch first line, interpret and checks for error, execute and generate output, then fetch second line
- No intermediate form of machine code

Compilers and Interpreters

Compilers	Interpreters
Takes entire program- compile all lines and check errors- generate target code-execute	Takes 1st line-interpret-error-execute- take next line
Translates high level language to a lower level language (assembly or m/c)- creates object code or machine code and stores in memory	Translates an instruction immediately into m/c language and executes it before next instruction (line by line)- no saving of m/c code
Compiler design is split into a number of relatively independent phases & passes	Design is not divided in to phases as entire code is not processed before execution

Compilers and Interpreters…

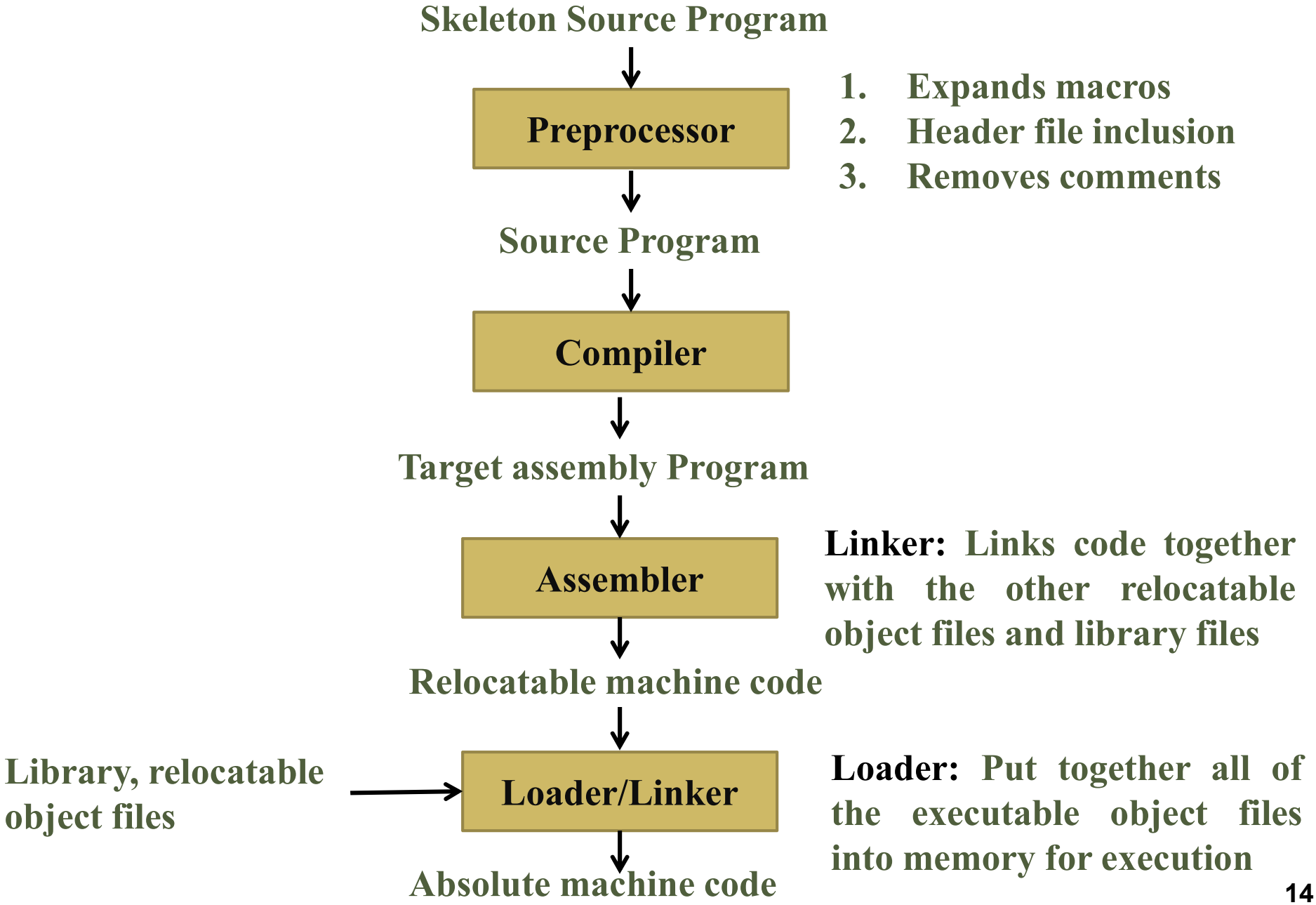
Compilers	Interpreters
After compilation, target code is stored somewhere for execution and it is relocatable	Code is not relocatable, used for programs having commands
Compile time and runtime memory can be different (more memory requirement)	Compile time and run time memory is same (less memory requirement)
Faster execution, target program executes independently w/o compiler	Slower execution as translation and execution at the same time, each time code is interpreted before execution

Compilers and Interpreters…

Compilers	Interpreters
Comments are removed and code is optimized	No optimization of code, translates and executes everything from i/p program
Links different files and generates an executable program	No linking of different files
C, C++	PHP, Perl

Some programming languages use both compiler and interpreter. Ex, Java, Python

A Language Processing System



Preprocessor

test.c file

```
#define M 5
#include<stdio.h>
void main()
{
        # code starts
        int a;
        a=M+7;
        printf("M=%d",M);
}
```

Output?

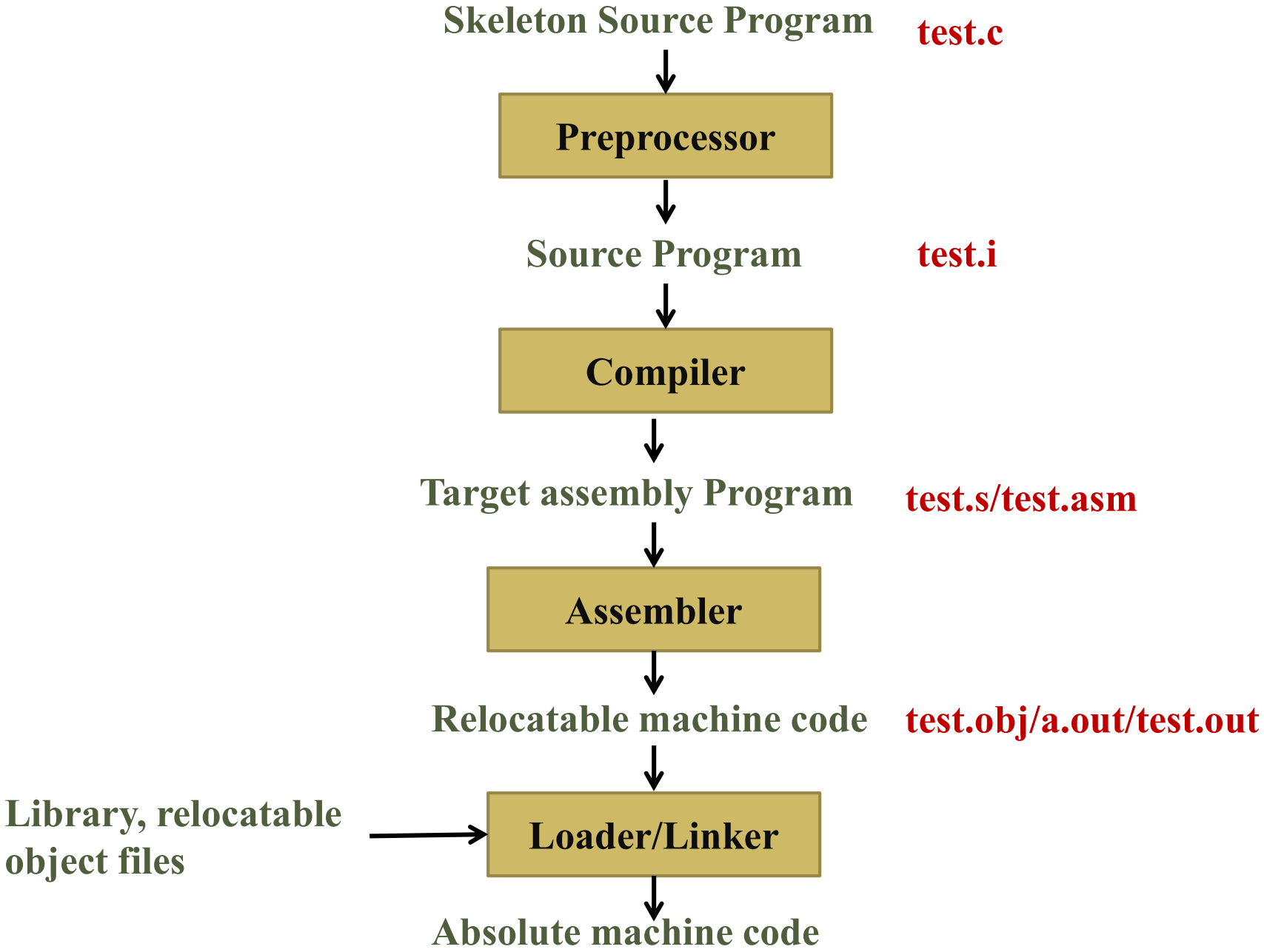
M=5

test.i file

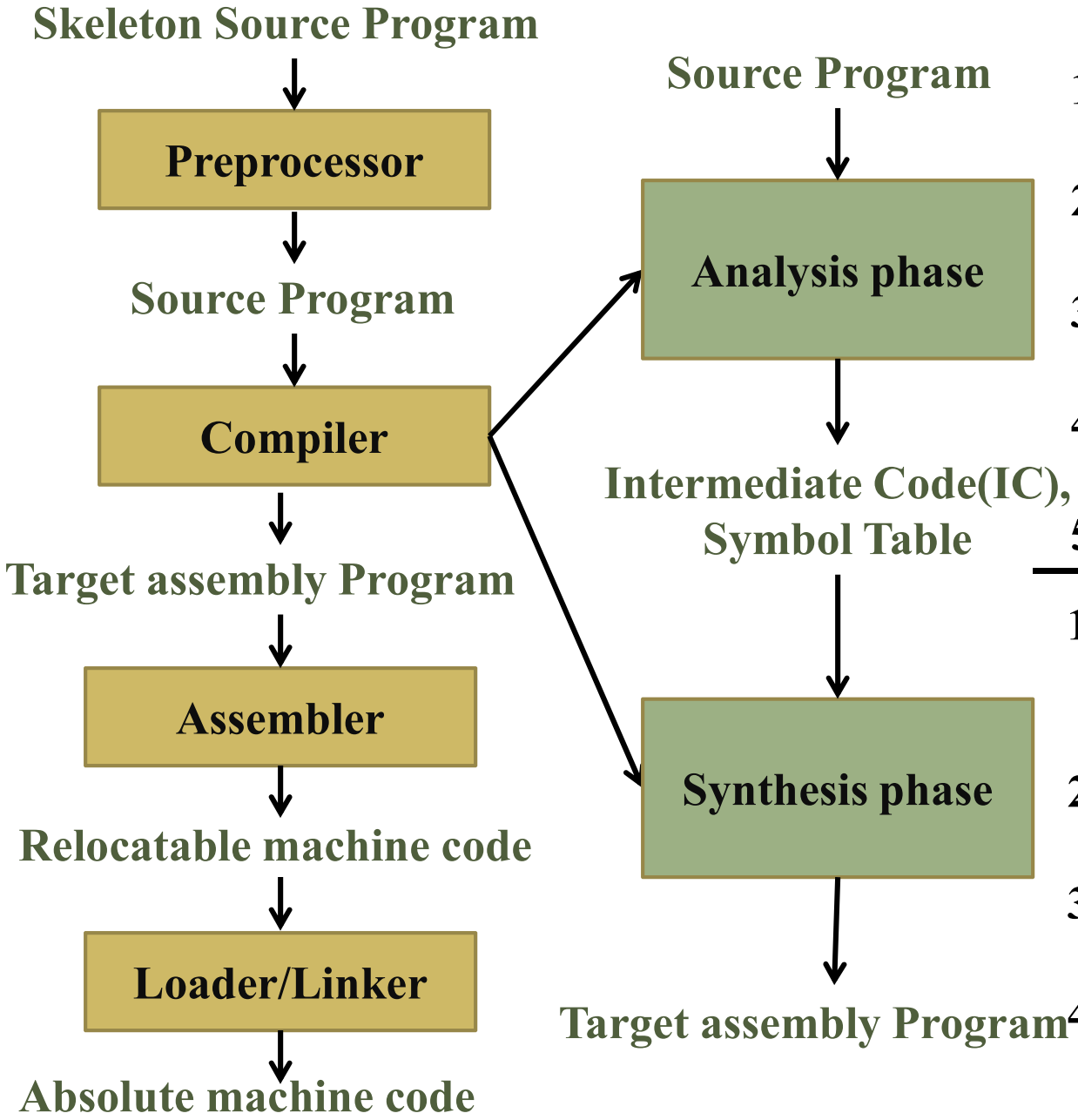
code written in stdio.h file

```
int _EXFUN(printf, (const char *, ...));
int  _EXFUN(scanf,  (const  char *, ...));
void main()
{
        int a;
        a=5+7;
        printf("M=%d",5);
}
```

A Language Processing System



Parts (Steps) of Compiler

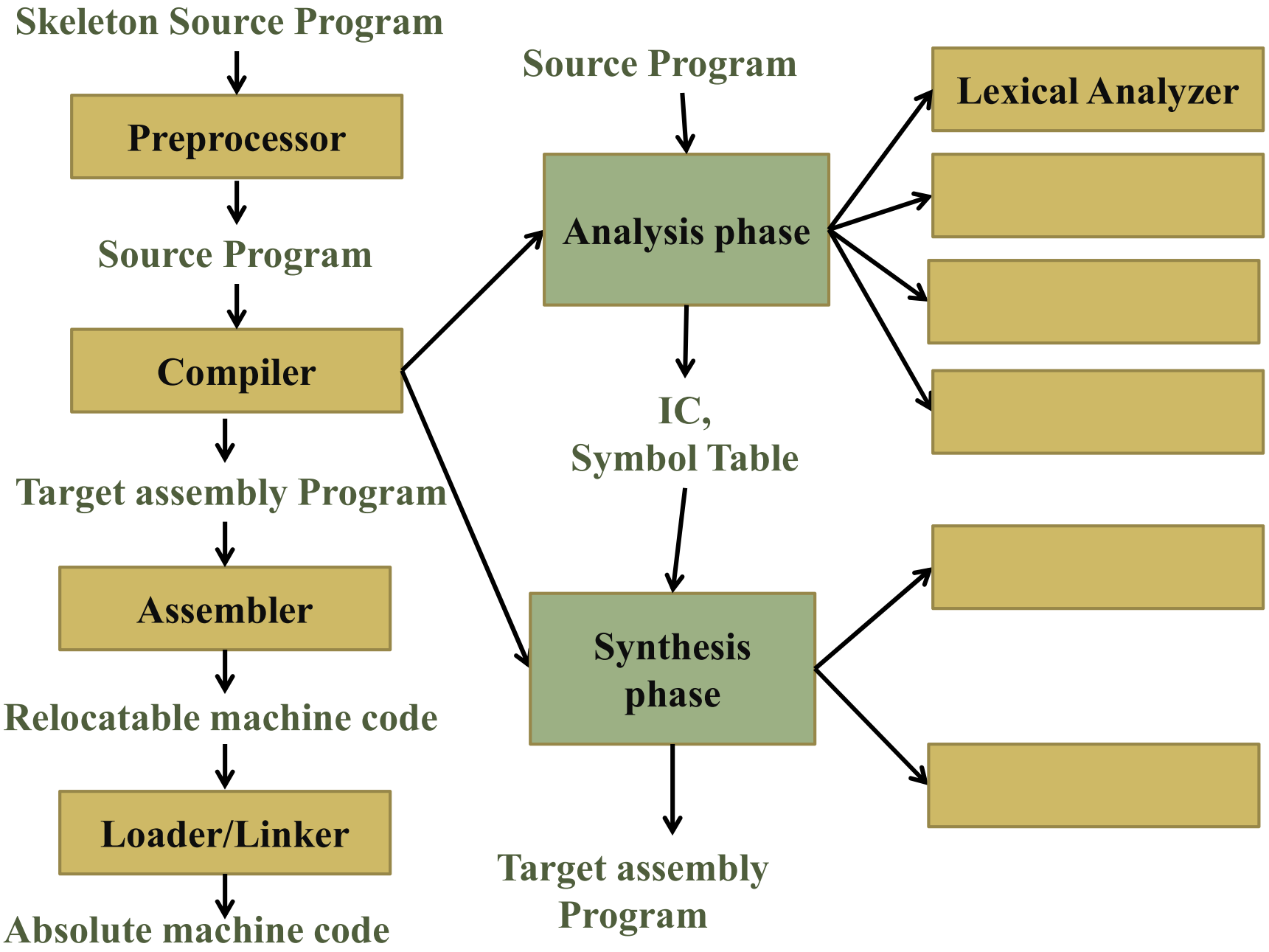


1. Performs analysis of input code
2. Generates errors and warnings
3. Largely dependent on the source language
4. Largely independent of the target machine
5. Creates FRONT END

---

1. Takes IC, uses symbol table and generates target program
2. Largely independent of the source language
3. Largely dependent on the target machine
4. Creates BACK END

Phases of Compiler



Lexical Analysis

- Grouping into tokens (sequence of characters with meaning)
- Identifies valid words or lexemes in source program
- Removes white spaces, tabs, newlines
- Passes sequence of tokens to syntax analysis phase
- Implemented as a finite automata
- Error generated: invalid identifier

Lexical Analysis

- For example:

  1) position = initial + rate * 60

  Tokens: position (id1), '=', initial (id2), '+', rate (id3), '*', 60

  Output of the lexical phase:

  <id1, 1> <=> <id2, 2> <+> <id3, 3> <*> <60>

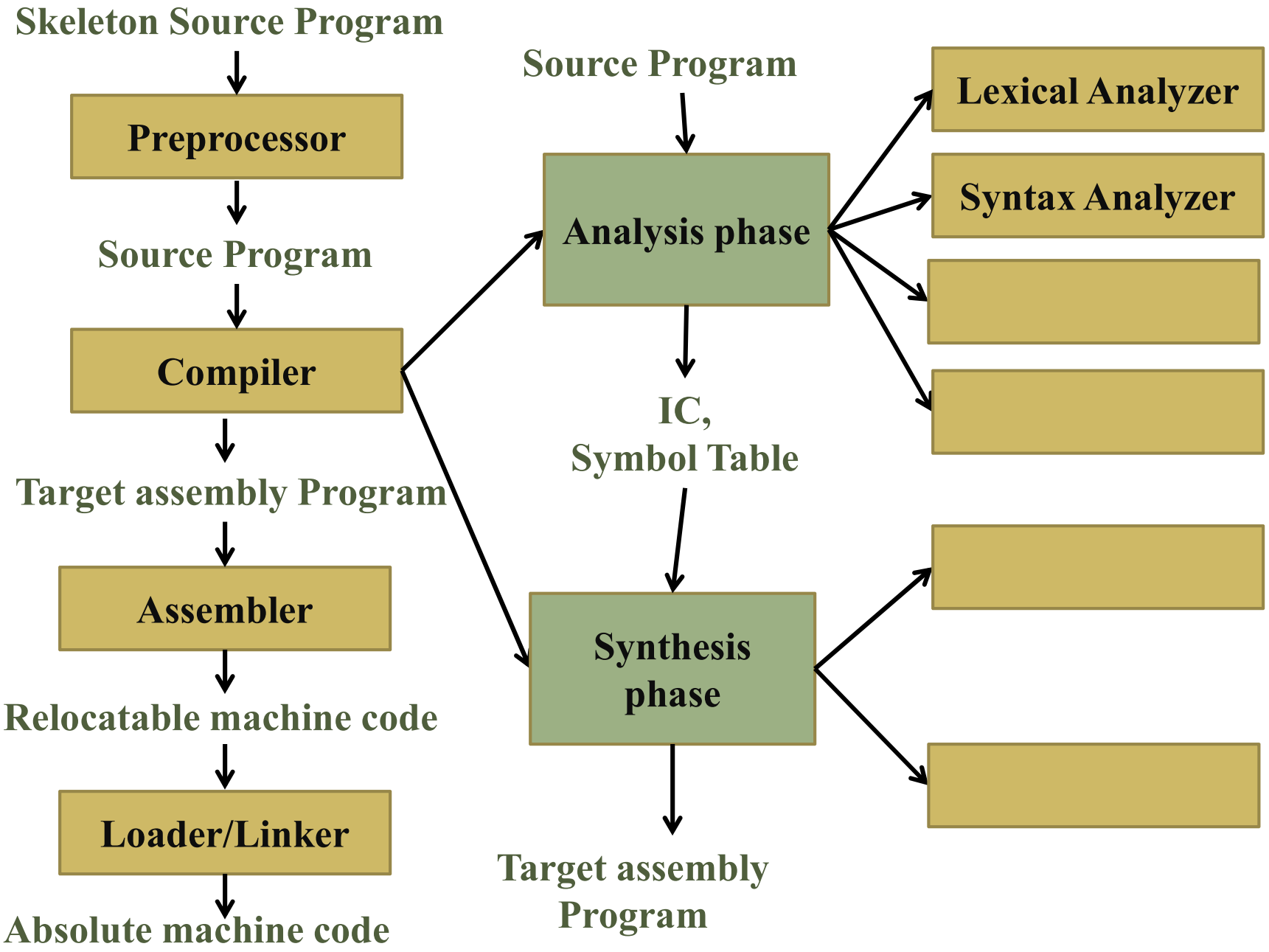
  2) int no1, no2;
     float f;
     no1 = no2 + f * 5 – no2;

  Tokens: int, no1 (id1), ',', no2(id2), ';', float, f (id3), ';', no1(id1), '=', no2(id2), '+', f(id3), '*' , 5, '-' , no2(id2), ';'

Lexical Analysis

- Limitations:
  - It can identify validity of lexemes (individual words) only
  - Not powerful to analyze expression or statement (Ex, it can't match a pair of parenthesis)

Phases of Compiler



Syntax Analysis (Hierarchical Analysis, Parsing)

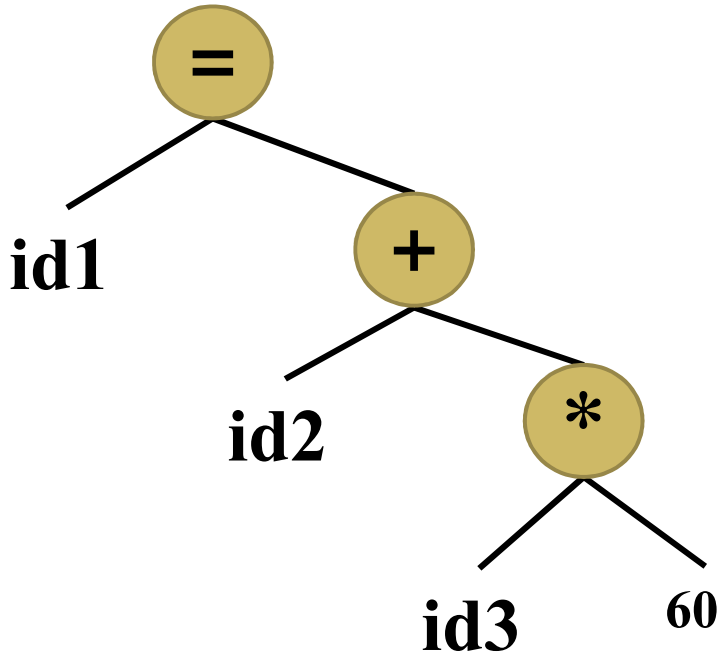
- Takes sequence of tokens from lexical analyzer
- Checks syntactic correctness of expressions/ statements/ blocks/ functions/ program
- Successful if it can identify grammar rules (CFG, CSG) for a sequence of tokens
- Generates a parse tree from a sequence of tokens
- If parse tree is complete -> construct is correct
- If parse tree is incomplete -> syntax error in the construct
- Uses recursive grammar rules
- Errors generated: missing parenthesis, missing semicolon, missing operand

Syntax Analysis…

- For example:

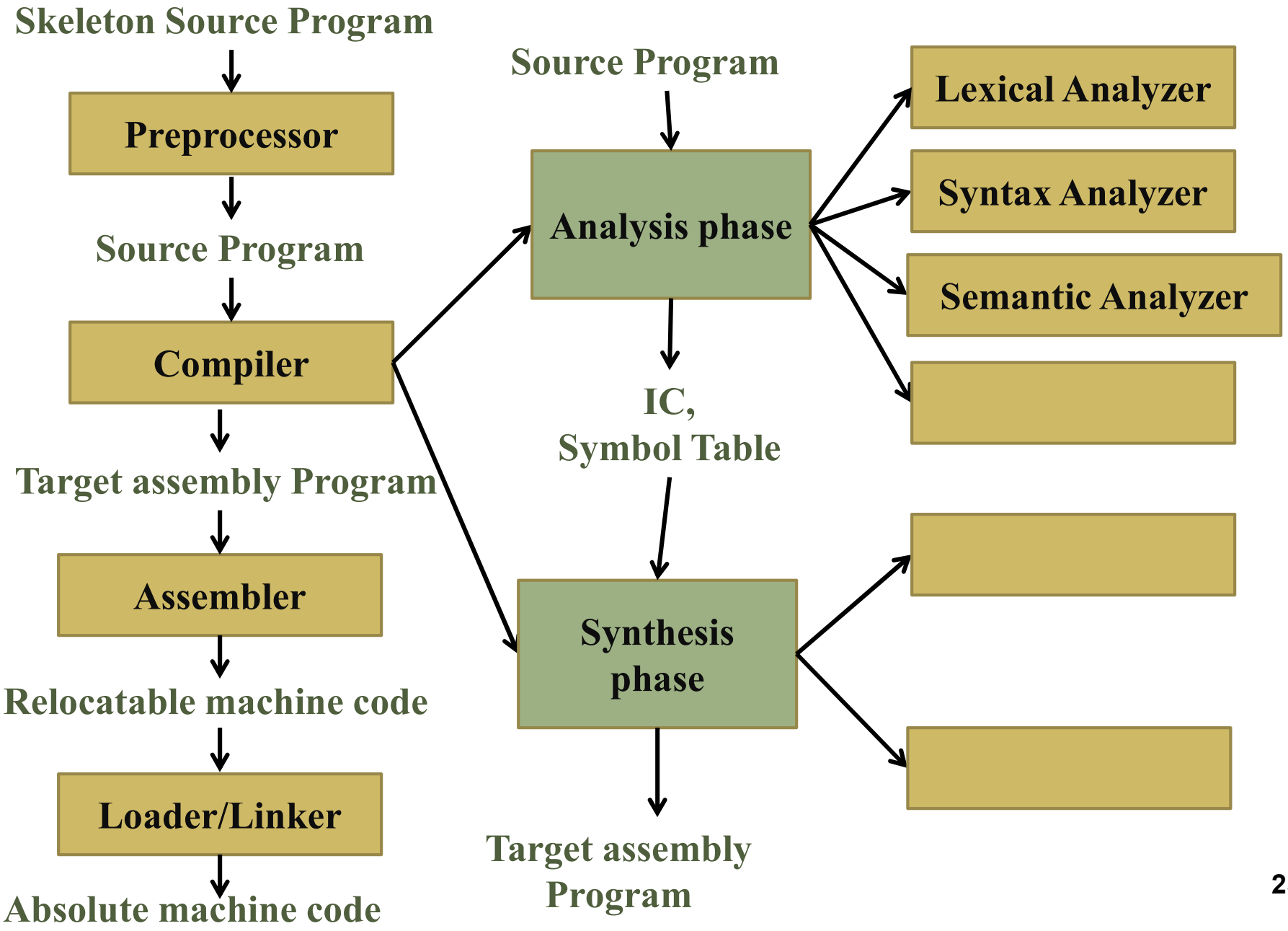
  1) position = initial + rate * 60

  Tokens: position (id1), '=', initial (id2), '+', rate (id3), '*', 60



? Input: a = b + ;

Phases of Compiler



Semantic Analysis

- If program is syntactically correct, then check for semantic (meaning) correctness of the program
- Semantics- depends on the programming language
- Check for semantic errors
- Gathers type information
- Uses parse tree generated by the syntax phase
- Common checks are:
  - Type of variables and type-casting
  - Applicability of operators on operands – each operator has operands permitted by the source language (string operators, only int index in array)
  - Scope of variables and functions
  - Determine definition of variables and functions (function overloading)

Semantic Analysis…

- For example,

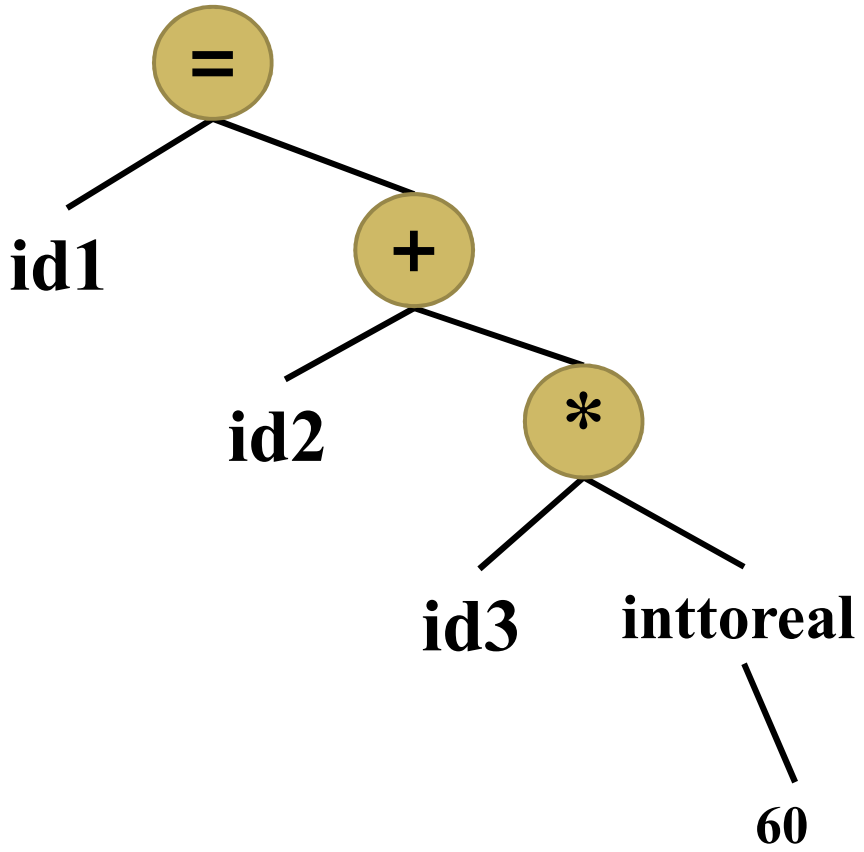
  int a=5, b=2;

  float f;

  f = a / b;

  f = a / 2.0; //inbuilt type-casting

- Errors:
  - float array index
  - Cannot convert ‘int’ to ‘int*’
  - Undeclared variable

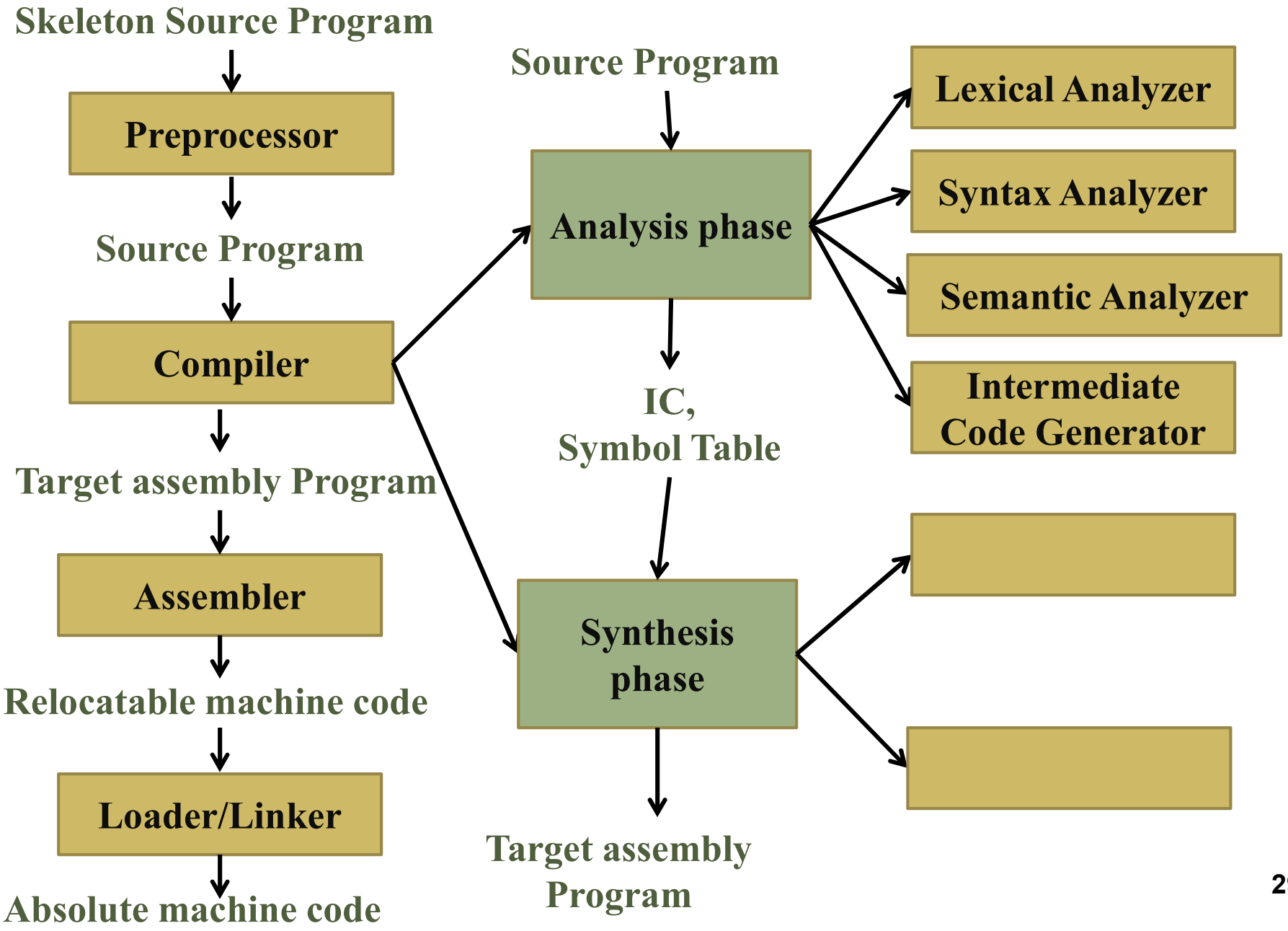


Errors

1. fi (a==b); // misspelled if or function call?
2. int a, a;
3. int a[10], b;

   c = a + b;

Phases of Compiler

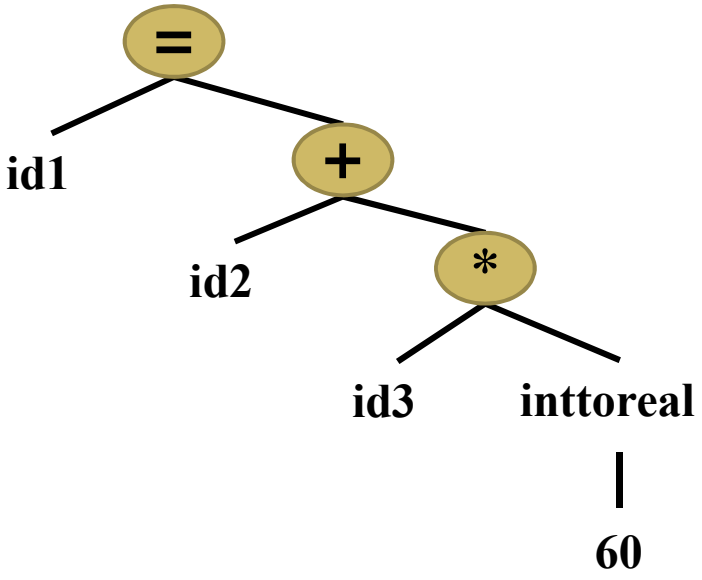


Intermediate Code Generator

- After semantic analysis, Intermediate Code (IC) of source program will be created (low-level or machine-like IR)
- Properties
  - Easy to produce
  - Easy to translate into target code
- Variety of forms
  - Three Address code (3A code)
  - Postfix representation

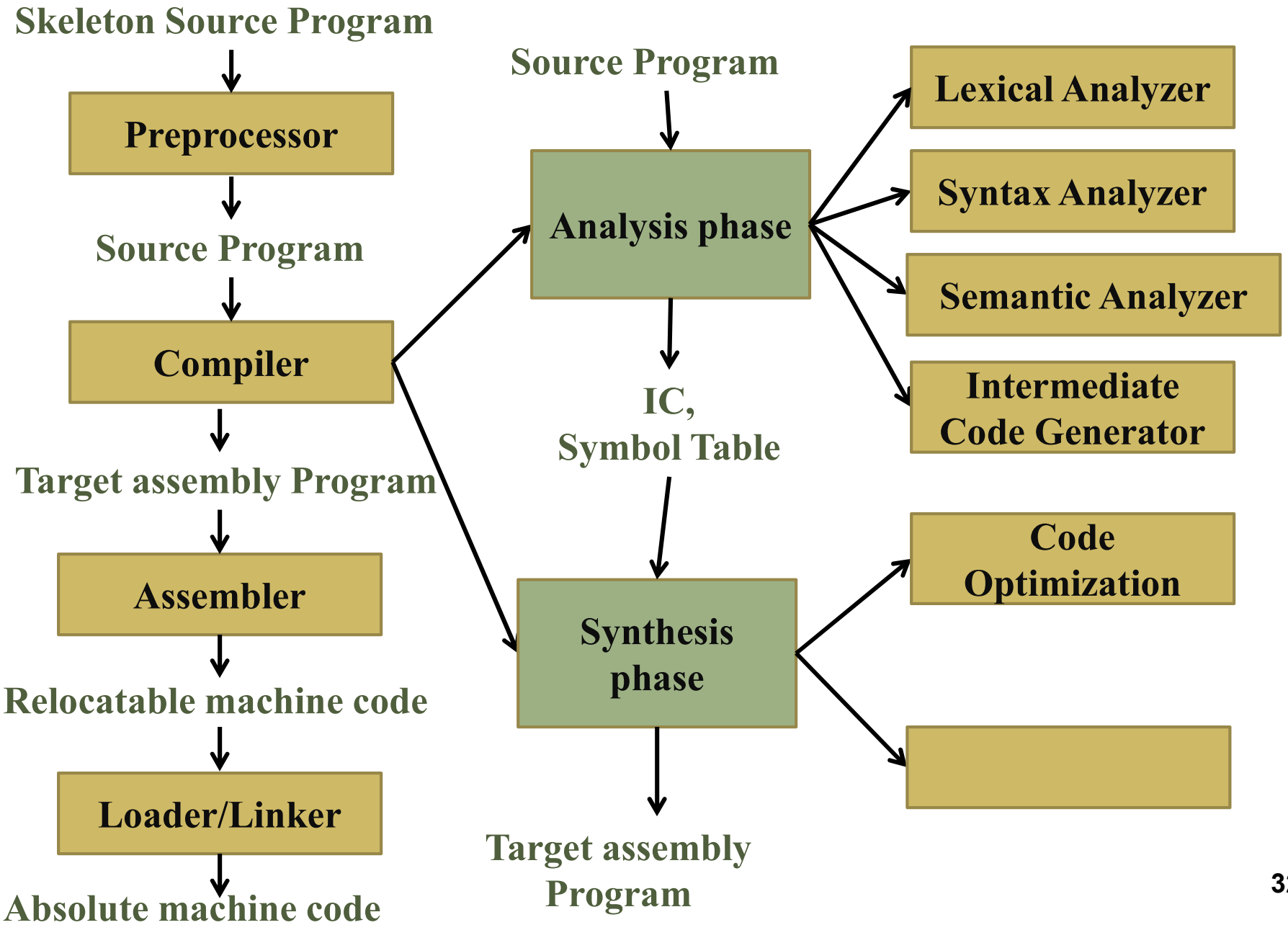
Three Address code (3A code)

- Each instruction has
  - Maximum three operands
  - Maximum one operator in addition to assignment
- Compiler will decide order of evaluation
- Compiler will generate a temp name to hold computed results
- For example, id1 = id2 + id3 * 60



t1 = inttoreal (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Phases of Compiler



Code Optimization

- Improves the intermediate code for
  - Faster execution of m/c code
  - Space optimization
- For example,
  - 60.0 instead of inttoreal(60)
  - No need of t3

```
t1 = inttoreal (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

- Removes dead code like unreachable code

Code Optimization

- Loop optimizations – major source of optimization
  - Save even a single line in loop – improves execution time significantly
  - Ex, b=5;

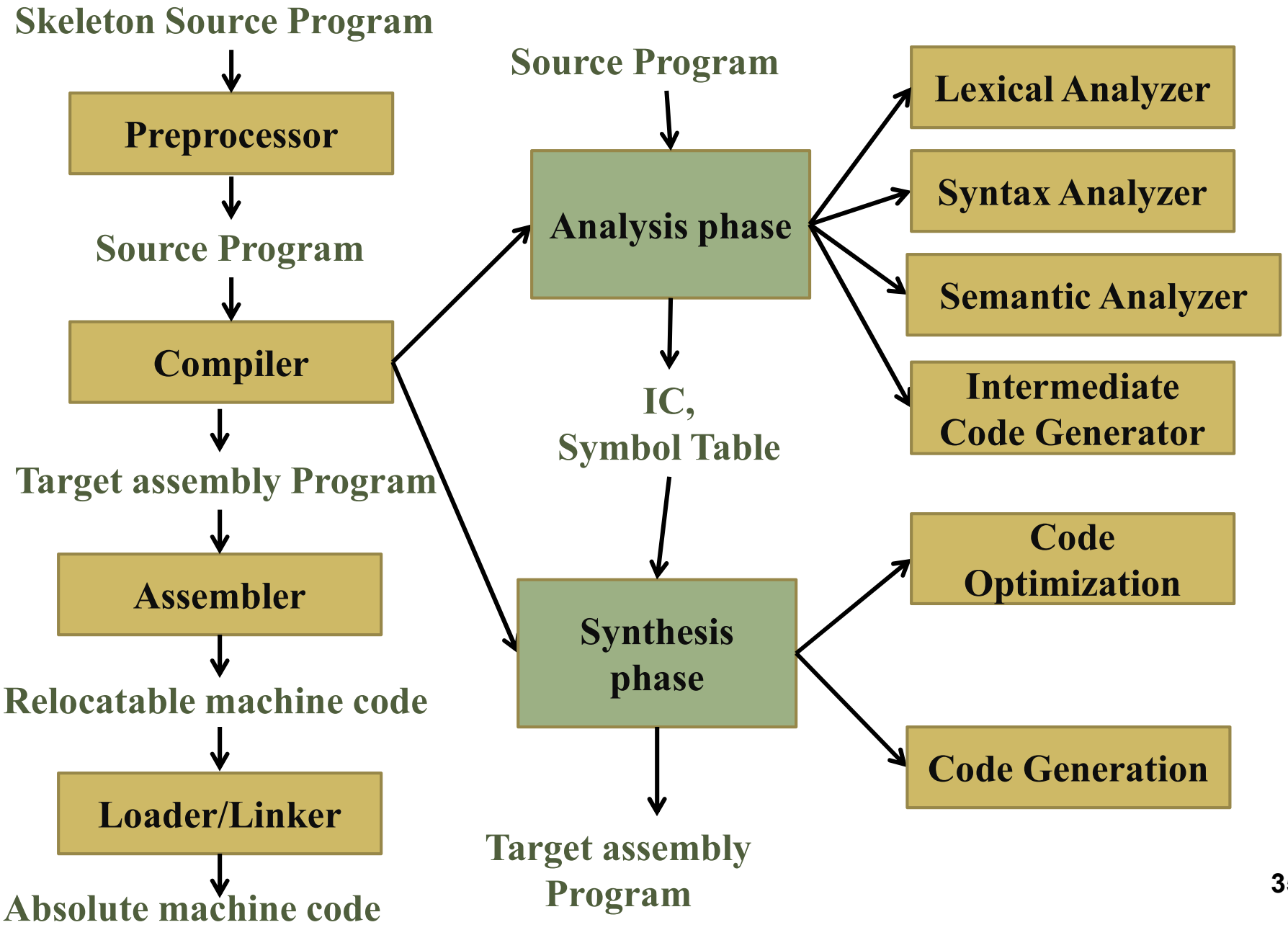
```
for (i=0; i< 10000; i++) {
        a = b + 100 – i * 2;
}
```

Optimized code:

```
b=5;
c = b + 100;
for (i=0; i< 10000; i++) {
        a = c – i * 2;
}
```

Algebraic simplification

Phases of Compiler



Code Generation

- Takes intermediate code as an input and maps to target code
- The target code can be
  - Relocatable m/c code
  - Assembly code
- Memory locations are selected for each variables of a program
- IC is translated to a sequence of m/c instructions

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Code Generation

- Part of back end of the compiler design
- Target code generation depends on
  - availability of m/c instructions
  - addressing modes
  - number of registers (general and special purpose)

Symbol Table Management

- Interacts with all phases- used as a reference table by all the phases
- Compiler needs
  - To record the identifiers (variables) used in source program
  - To collect information about various attributes of each identifier
  - Attributes for variable names
    - Name, type, class, size, relative offset within program, scope, value
  - Attributes for function names
    - Name, no of arguments, types of arguments, return type, method of passing each parameter

Symbol Table Management…

- It is a data structure containing a record for each identifier with fields for the attributes of the identifier.
- Lexical analyzer: detects identifier and enters in the symbol table, but can't determine all its attributes
- The remaining phases enter information about identifiers into symbol table and uses the entered information in different ways
- Semantic analyzer and IC generation needs to know type of variable
- Code generator: enters and uses details about the storage assigned to identifier

Symbol Table Management…

- During compilation, all phases often look up this table for definition of variables, example
  - Variable used is defined before that or not?
- Hence, operations in the symbol table are
  - Allocate, free, search (look up), insert, set_attribute, get_attribute in the symbol table
- Use the data structure to implement symbol table such that
  - It minimizes search time
  - Use hierarchical table instead of single flat one
  - Linear list, search trees, hash tables

Symbol Table Example

```
int g; //global
variable
void main() {
    int a, b;
    { float f; }
    { int i , j;}
}
int add(int a, int b){
     char c = a+b;
     {  int k; }
}
```

Global Symbol Table

Symbol	Type	Scope	Data type/ Return type
g	var	global	int
main	proc	global	void
add	proc	global	int

Symbol Table of main

Symbol	Type	Scope	Data type/ Return type
a	var	proc main	int
b	var	proc main	int

Symbol Table of add

Symbol	Type	Scope	Data type/ Return type
f	var	inner	float

Symbol	Type	Scope	Data type/ Return type
i	var	inner	int
j	var	inner	int

```
int g; //global
variable
void main() {
    int a, b;
    { float f; }
    { int i , j;}
}
int add(int a, int b){
     char c = a+b;
     {  int k; }
}
```

Symbol Table Example …

Global Symbol Table

Symbol	Type	Scope	Data type/ Return type
g	var	global	int
main	func	global	void
add	func	global	int

Symbol Table of main

Symbol Table of add

Symbol	Type	Scope	Data type/ Return type
a	arg	proc add	int
b	arg	proc add	int
c	var	proc add	char

Symbol	Type	Scope	Data type/ Return type
k	var	inner	int

Error Handler

- Error detection and reporting
- Each phase can identify errors
- After detection of errors, a phase should deal with that error
  - So compiler can proceed
  - Allows further error detection in the program
- Semantic error- parser can progress
- Syntax error- parser reaches in error state
  - Indicate regarding type of error
  - Undo some processing already carried out by parser- error recovery
- Error detection and recovery- vital role in overall operations of compiler

Practical-1

- Identify keywords and identifiers from given ‘C’ program.