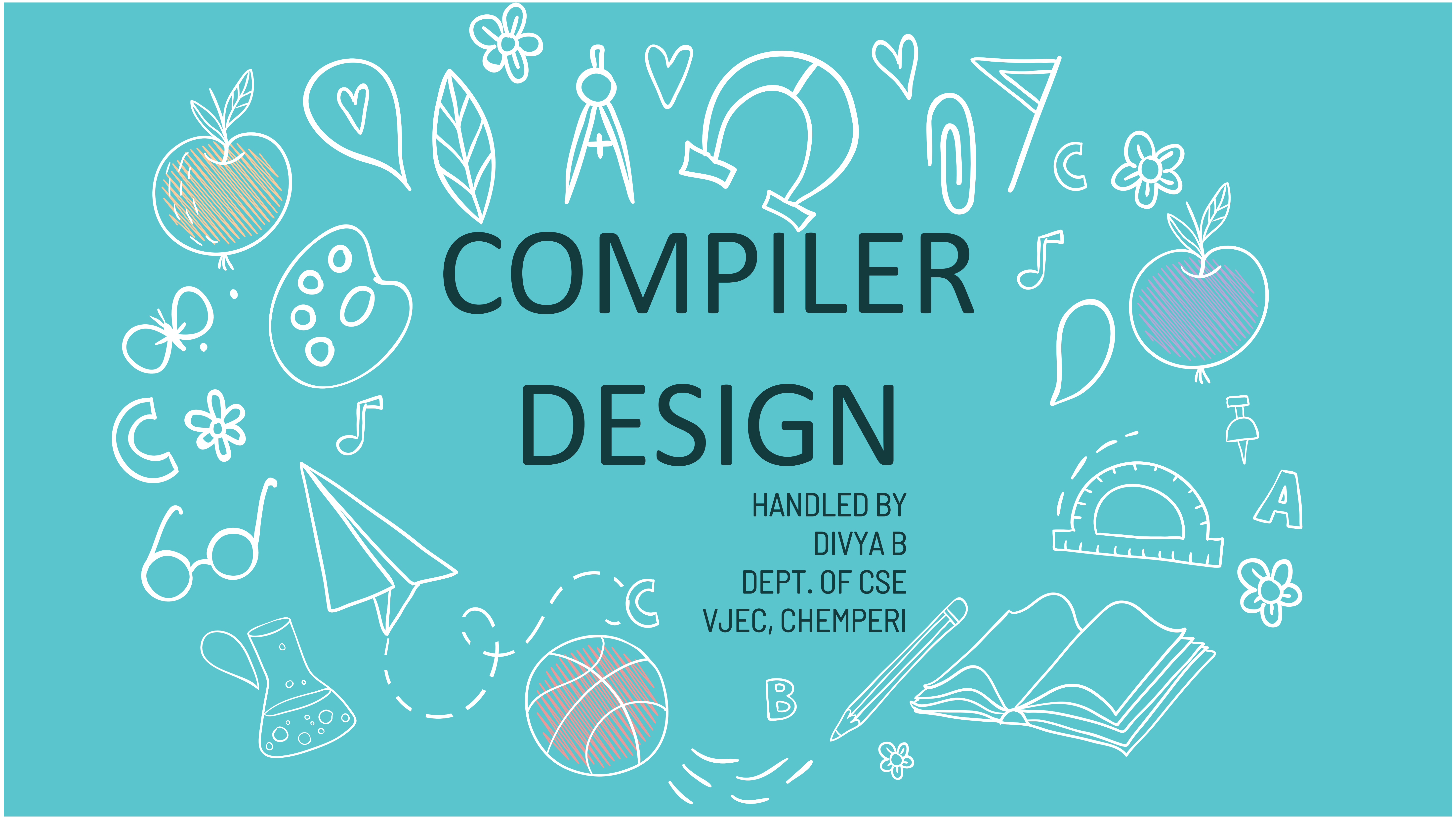


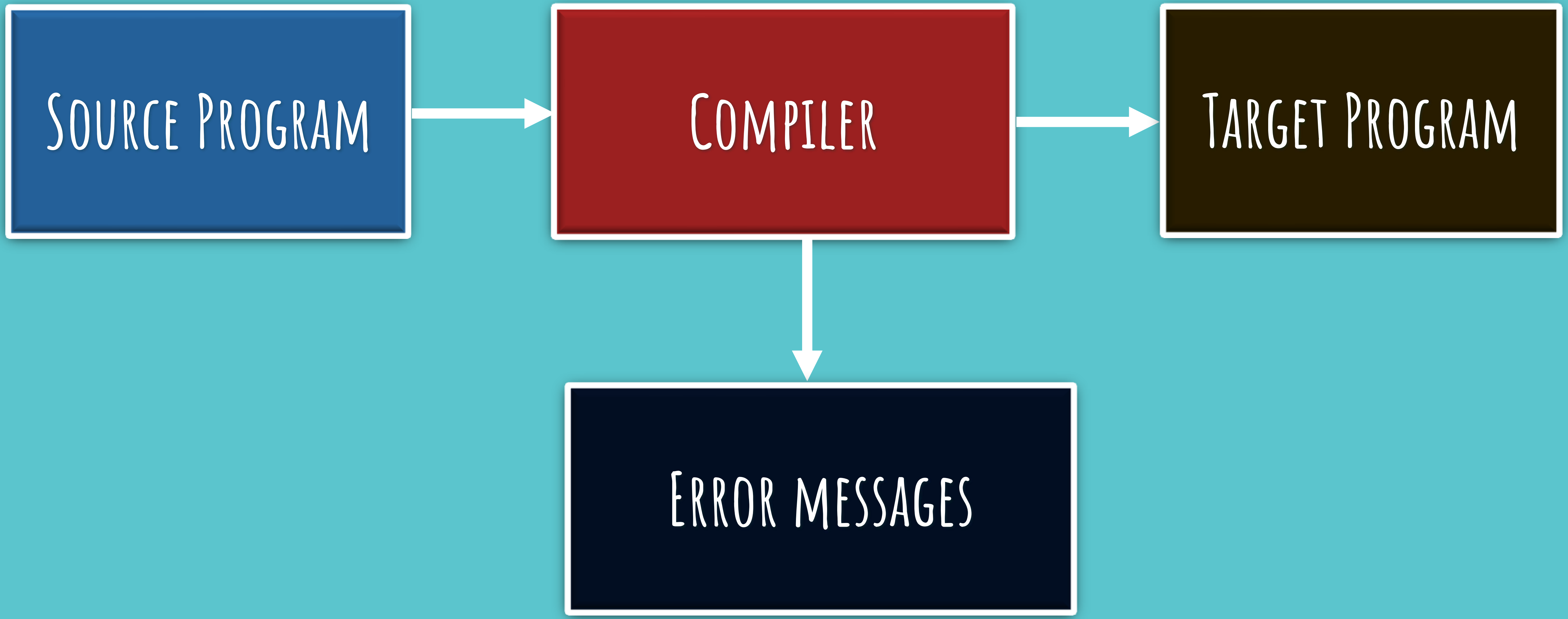
COMPILER DESIGN

HANDLED BY
DIVYA B
DEPT. OF CSE
VJEC, CHEMPERI

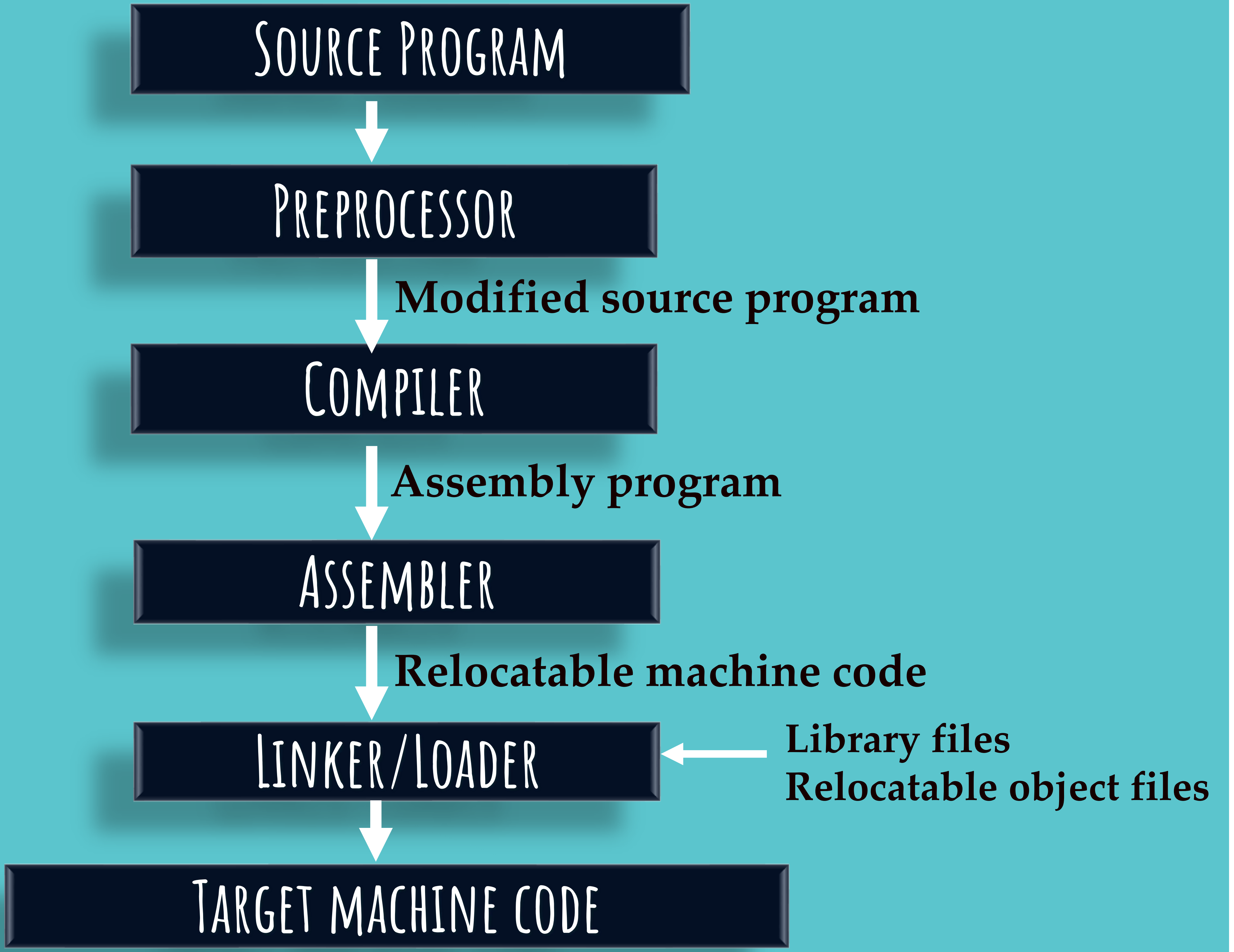
INTRODUCTION

- A ***compiler*** is a program that can read a program in one language (the source language) and translate it into an equivalent program in another language (the target language).

- An important role of the compiler is to report any errors in the source program that it detects during the translation process.


Source Program
Compiler
Target Program
Error messages

LANGUAGE PROCESSING SYSTEM



Divys-Compiler Design PPT

STRUCTURE OF THE COMPILER

- Compilation process has two parts
  - **Analysis part**
    - Called as the front end of the compiler.
  - **Synthesis part**
    - Called as the back end of the compiler.

STRUCTURE OF THE COMPILER

- **Analysis**
  - This breaks the source program into constituent pieces and imposes a grammatical structure on them.
  - This structure is used to create an intermediate structure of the source program.

STRUCTURE OF THE COMPILER

- **Analysis**
  - If the analysis part detects that the source program is syntactically or semantically incorrect, then the compiler should give informative messages to the user.

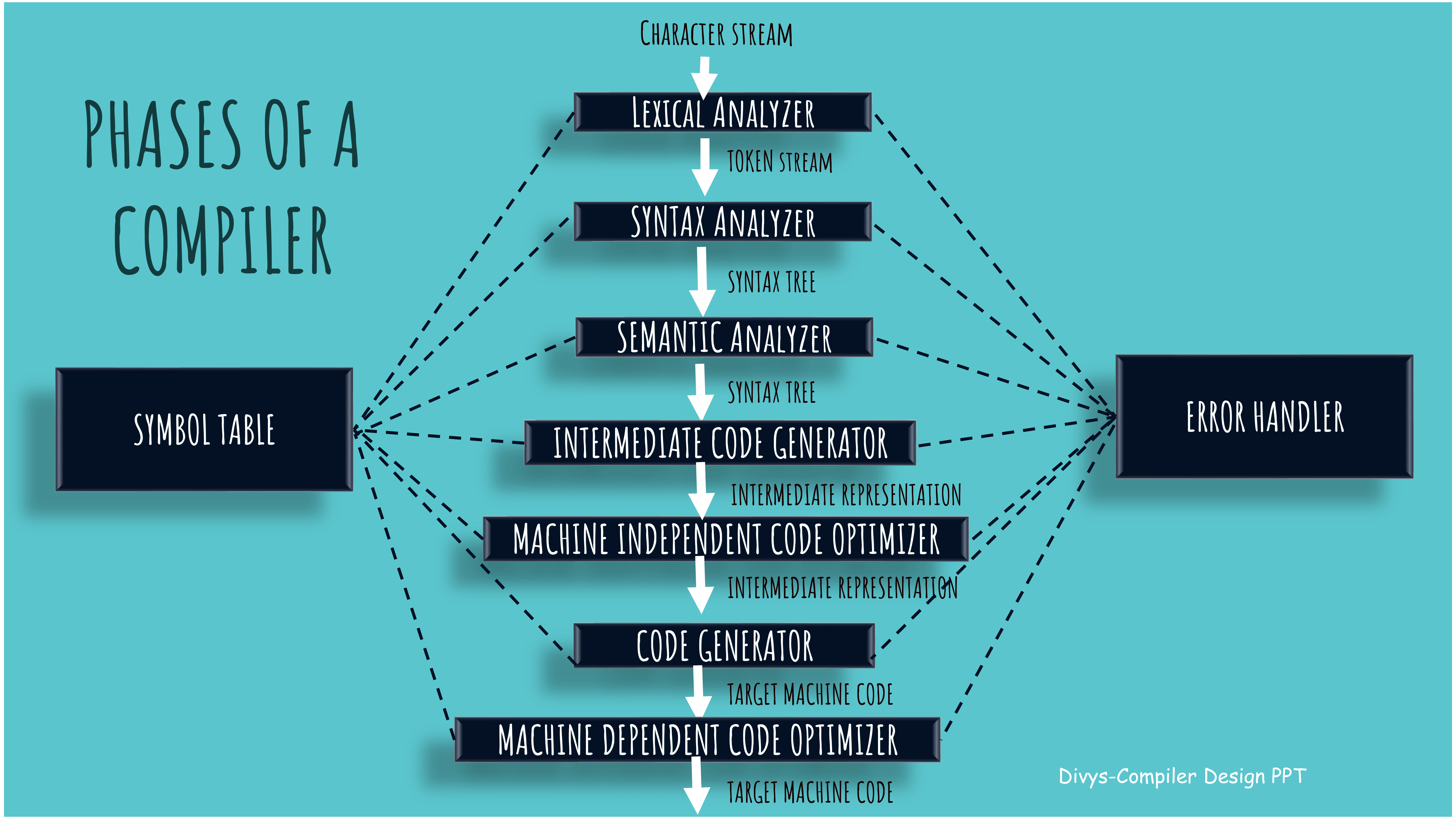
STRUCTURE OF THE COMPILER

- **Analysis**
  - Analysis part also collects information about the source program and stores it in a data structure called a ***symbol table***.
  - Symbol table is passed along with the intermediate representation to the synthesis part.

STRUCTURE OF THE COMPILER

- **Synthesis**
  - Synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.

PHASES OF A COMPILER



Divys-Compiler Design PPT

PHASES OF A COMPILER

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generation

LEXICAL ANALYSIS

- It is also called as ***scanning***.
- The lexical analyzer reads a stream of characters making up the source program and groups the characters into meaningful sequences called ***lexemes***.

LEXICAL ANALYSIS

- For each lexeme, the lexical analyzer produces as output a token of the form

  <token-name, attribute-value>

- This is passed to the syntax analysis phase.

LEXICAL ANALYSIS

- ***token-name*** is an abstract symbol that is used during syntax analysis.
- ***attribute-value*** points to an entry in the symbol table for this token.
- Information from the symbol table entry is needed for semantic analysis and code generation.

LEXICAL ANALYSIS

- Suppose a source program contains the assignment statement

```
position = initial + rate * 60
```

LEXICAL ANALYSIS

- The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer
  1. ***position*** is a lexeme that would be mapped into a token <id, 1>, where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol table entry for an identifier holds information about the identifier, such as its name and type.

LEXICAL ANALYSIS

2. The assignment symbol **=** is a lexeme that is mapped into the token < = >. This token needs no attribute-value, so the second component is omitted.
3. ***initial*** is a lexeme that is mapped into the token < id, 2> , where 2 points to the symbol-table entry for initial .
4. **+** is a lexeme that is mapped into the token <+>.

LEXICAL ANALYSIS

5. ***rate*** is a lexeme that is mapped into the token < id, 3 >, where 3 points to the symbol-table entry for rate.
6. * is a lexeme that is mapped into the token <* > .
7. ***60*** is a lexeme that is mapped into the token <60> .

LEXICAL ANALYSIS

- Blanks separating the lexemes would be discarded by the lexical analyzer.
- The representation of the assignment statement ***position = initial + rate * 60*** after lexical analysis as the sequence of tokens is

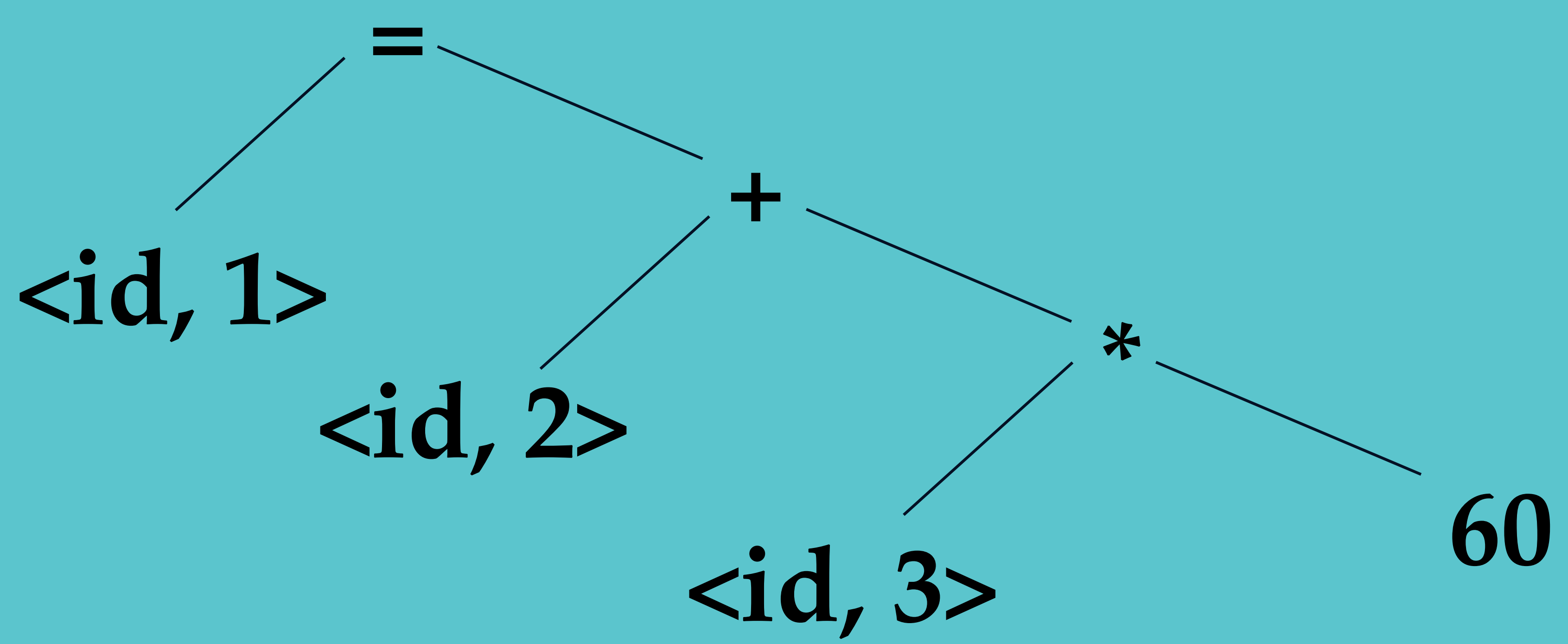
$$< id, 1 > < = > < id, 2 > <+> < id, 3 > <*> < 60 >$$

SYNTAX ANALYSIS

- This phase is also called as ***parsing***.
- The parser uses the components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- A typical representation is a ***syntax tree*** in which each interior node represents an operation and the children of the node represent the arguments of the operation.

SYNTAX ANALYSIS

- Syntax tree for the previous token stream is



SYNTAX ANALYSIS

- The tree has an interior node labeled with <id, 3> as its left child and the integer 60 as its right child.
- The node <id, 3> represents the identifier rate.
- The node labeled * makes it explicit that we must first multiply the value of rate by 60.
- The node labeled + indicates that we must add the result of this multiplication to the value of initial.
- The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier position.

SEMANTIC ANALYSIS

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- An important part of semantic analysis is ***type checking,*** where the compiler checks that each operator has matching operands.

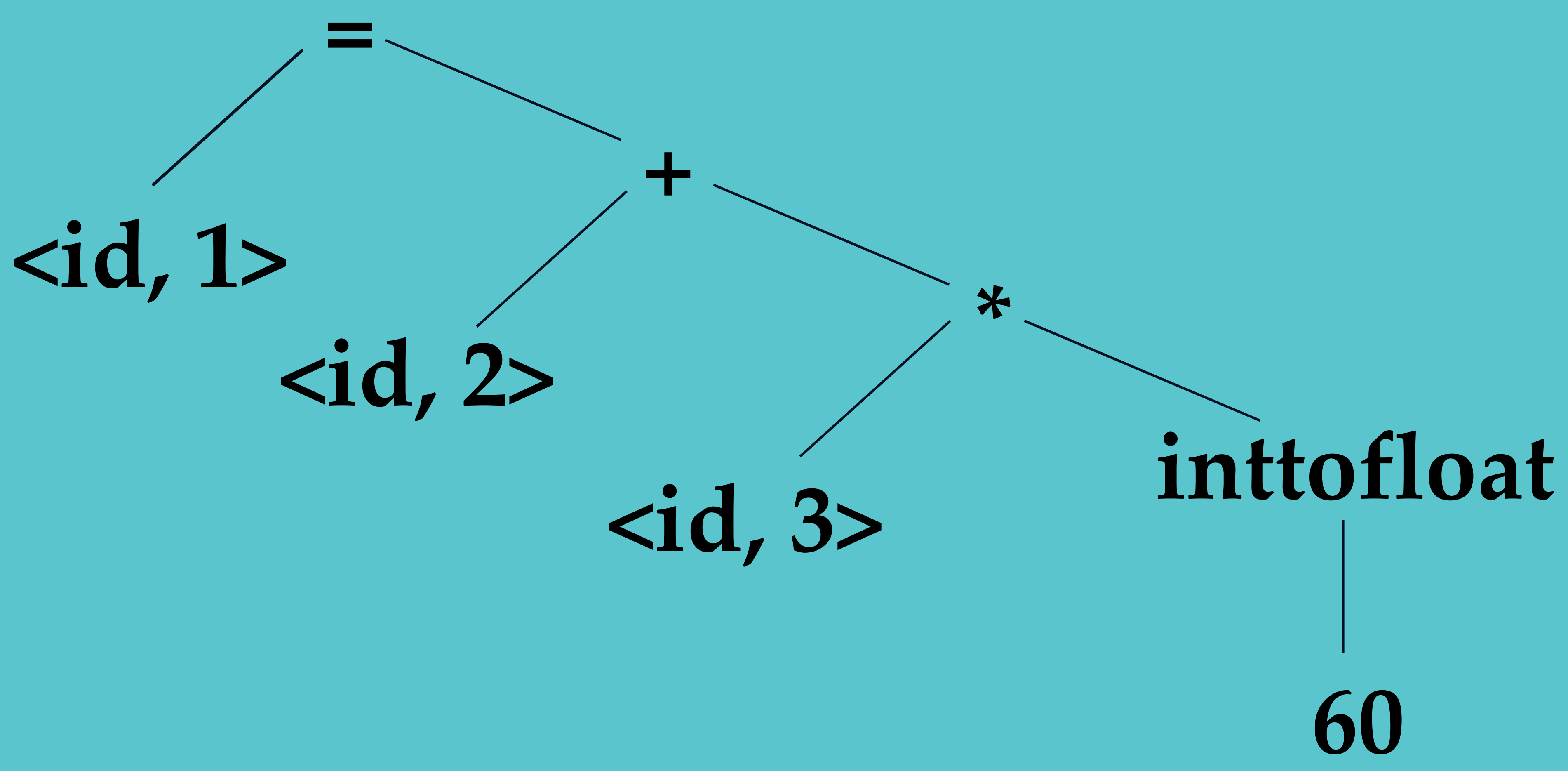
SEMANTIC ANALYSIS

- For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.
- Some sort of type conversion is also done by the semantic analyzer.
    - For example, if the operator is applied to a floating point number and an integer, the compiler may convert the integer into a floating point number.

SEMANTIC ANALYSIS

- In the example, suppose that position, initial, and rate have been declared to be floating- point numbers, and that the lexeme 60 by itself forms an integer.
- The semantic analyzer discovers that the operator * is applied to a floating-point number rate and an integer 60.
- In this case, the integer may be converted into a floating-point number.

SEMANTIC ANALYSIS



INTERMEDIATE CODE GENERATION

- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.
- Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

INTERMEDIATE CODE GENERATION

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.
- This intermediate representation should have two important properties
  - It should be simple and easy to produce.
  - It should be easy to translate into target machine code.

INTERMEDIATE CODE GENERATION

- The intermediate representation used is called ***three-address code,*** which consists of a sequence of assembly-like instructions with three operands per instruction.
- Each operand can act like a register.

INTERMEDIATE CODE GENERATION

- The output of the intermediate code generator in the previous example would be,

$$\boldsymbol{t1 = inttofloat(60)}$$

$$\boldsymbol{t2 = id3 \ * t1}$$

$$\boldsymbol{t3 = id2 + t2}$$

$$\boldsymbol{id1 = t3}$$

CODE OPTIMIZATION

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- The objectives for performing optimization are ***faster execution, shorter code, or target code that consumes less power***.

CODE OPTIMIZATION

- The previous three-address code can be optimized as,

$$\boldsymbol{t1 = id3 \ * 60.0}$$

$$\boldsymbol{id1 = id2 + t1}$$

CODE GENERATION

- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

CODE GENERATION

- A crucial aspect of code generation is the judicious assignment of registers to hold variables.
- If the target language is assembly language, this phase generates the assembly code as its output.

CODE GENERATION

- The code generated for the previous example is,

$$\boldsymbol{LDF\ R2, id3}$$

$$\boldsymbol{MULF\ R2, \#60.0}$$

$$\boldsymbol{LDF\ R1, id2}$$

$$\boldsymbol{ADDF\ R1, R2}$$

$$\boldsymbol{STF\ id1, R1}$$

CODE GENERATION

- The first operand of each instruction specifies a destination.
- The F in each instruction tells us that it deals with floating-point numbers.
- The above code loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0.
- The # signifies that 60.0 is to be treated as an immediate constant.
- The third instruction moves id2 into register Rl and the fourth adds to it the value previously computed in register R2.
- Finally, the value in register Rl is stored into the address of idl , so the code correctly implements the assignment statement

position = initial + rate * 60

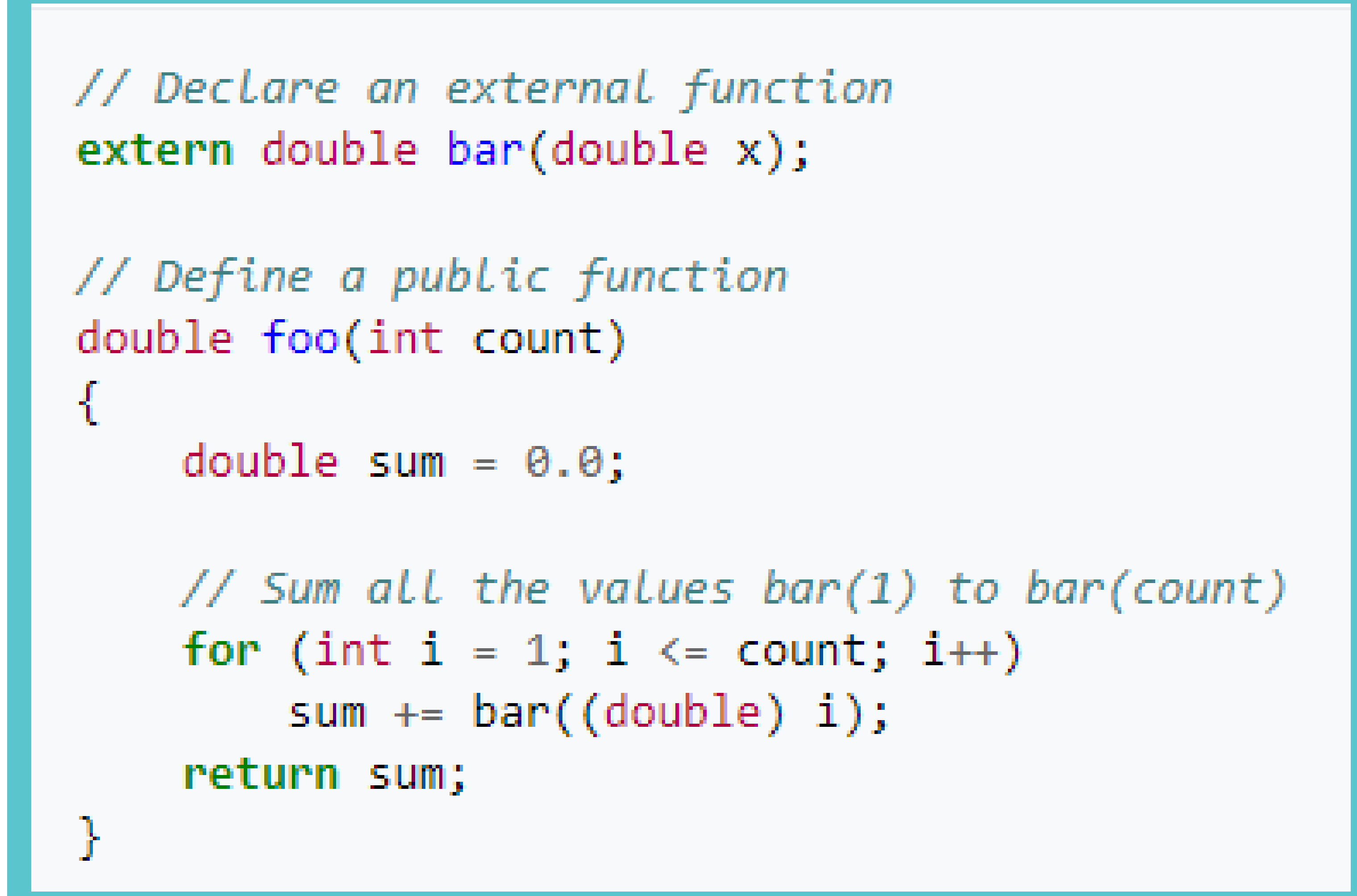
SYMBOL TABLE

- An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.
- These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

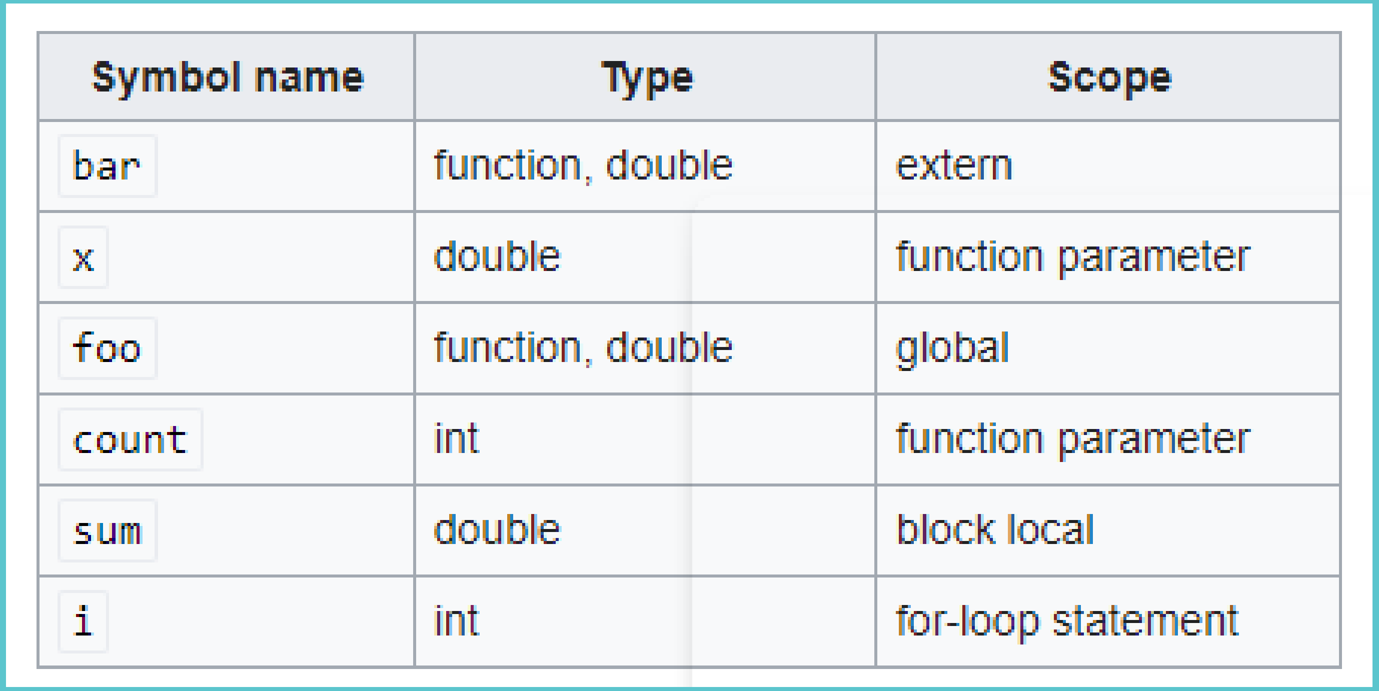
SYMBOL TABLE

- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.
- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

SYMBOL TABLE

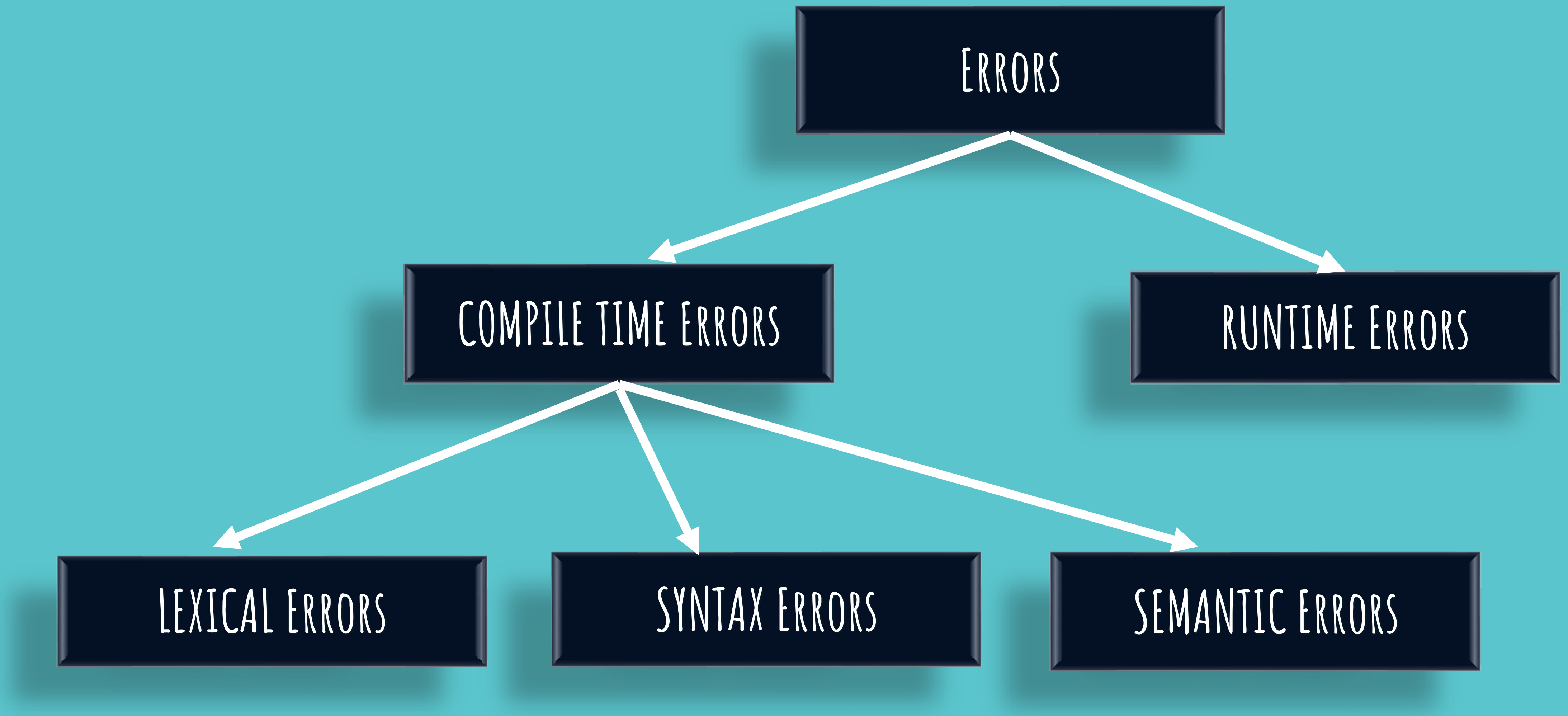


```
// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
    double sum = 0.0;

    // Sum all the values bar(1) to bar(count)
    for (int i = 1; i <= count; i++)
        sum += bar((double) i);
    return sum;
}
```



Symbol name	Type	Scope
bar	function, double	extern
x	double	function parameter
foo	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

ERROR DETECTION & REPORTING



ERROR DETECTION & REPORTING

- Each phase can encounter errors.
- However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- A compiler that stops when it finds the first error is not a helpful one.

ERROR DETECTION & REPORTING

- **Lexical Errors**
  - Exceeding length of identifier or numeric constants
  - Illegal characters
  - Unmatched string

ERROR DETECTION & REPORTING

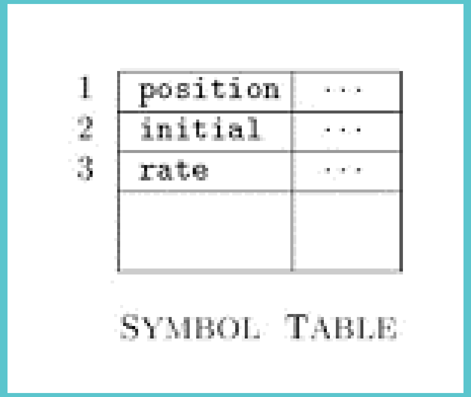
- **Syntax Errors**
  - Errors in structure
  - Missing operator
  - Misspelled keywords
  - Unbalanced parenthesis

ERROR DETECTION & REPORTING

- **Semantic Errors**
  - Incompatible type of operands
  - Undeclared variables
  - Non matching of actual arguments with formal arguments

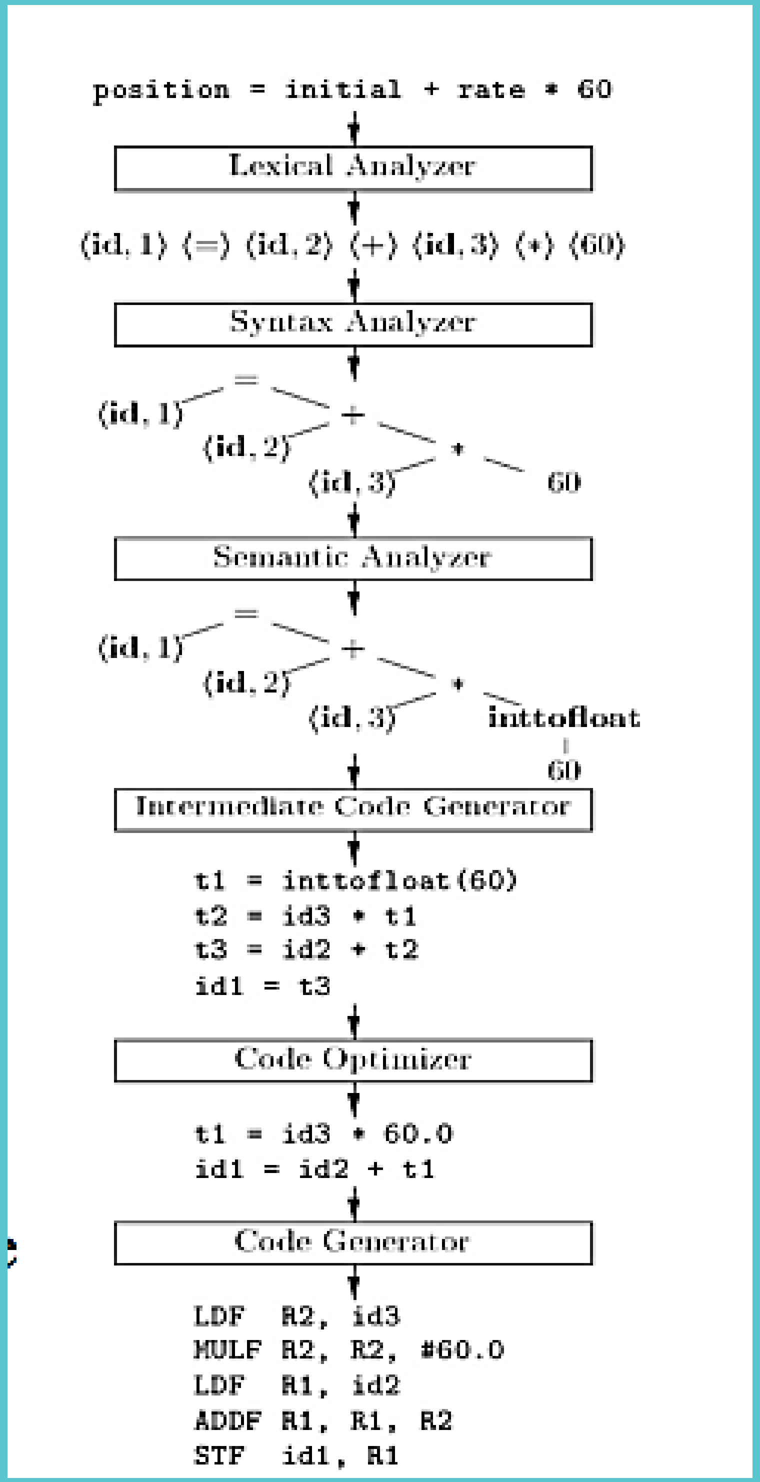
TRANSLATION OF AN ASSIGNMENT STATEMENT

$$position = initial + rate \ * 60$$



1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



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

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

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

TOKENS, LEXEMES & PATTERNS

- **Tokens**
  - Token is a sequence of characters that can be treated as a single logical entity.
  - Typical tokens are,
    - Identifiers
    - Keywords
    - Constants
    - Special symbols
    - Operators

TOKENS, PATTERNS & LEXEMES

- **Patterns**
  - A set of strings in the input for which the same token is produced as output.
  - This set of strings is described by a rule called a pattern associated with the token.

TOKENS, PATTERNS & LEXEMES

- **Lexemes**
  - A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

TOKENS, PATTERNS & LEXEMES

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, l, s, e	else
relation	<, >, <=, >=, = = , !=	<=, !=
id	letter followed by letters and digits	pi, D2
number	any numeric constant	3.14159, 7.13e5
literal	any characters between " and " except "	"core dumped"

COMPILER WRITING TOOLS

- Compiler writers use software development tools and more specialized tools for implementing various phases of a compiler.
- Some commonly used compiler construction tools include the following
  - **Scanner generators**
  - **Parser generators**
  - **Syntax directed translation engine**
  - **Data flow analysis engines**
  - **Automatic code generators**
  - **Compiler construction toolkits**

COMPILER WRITING TOOLS

- **Scanner generators**

Input : Regular expression description of the tokens of a language.

Output : Lexical analyzers.

- These automatically generate lexical analyzers, normally from a specification based on regular expressions.
- The basic organization of the resulting lexical analyzer is in effect a finite automaton.

COMPILER WRITING TOOLS

- **Parser generators**

Input : Grammatical description of a programming language.

Output : Syntax analyzers.

- These produce syntax analyzers, normally from input that is based on a context-free grammar.
- It is highly complex and consumes more compilation time.

COMPILER WRITING TOOLS

- **Syntax Directed Translation Engine**

Input : Parse tree.
Output : Intermediate code.

- These produce collections of routines that walk the parse tree, generating intermediate code.
- In this, each node of the parse tree is associated with one or more translations.

COMPILER WRITING TOOLS

- **Data flow engines**

- Data-flow analysis engine gathers the information about values transmitted from one part of a program to other parts.
- Data-flow analysis is a key part of code optimization.

COMPILER WRITING TOOLS

- **Automatic code generators**

Input : Intermediate language.
Output : Machine language.

- Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.
- A template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement using templates.

COMPILER WRITING TOOLS

- **Compiler Construction Toolkits**

  - It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.