
Program
Compilation

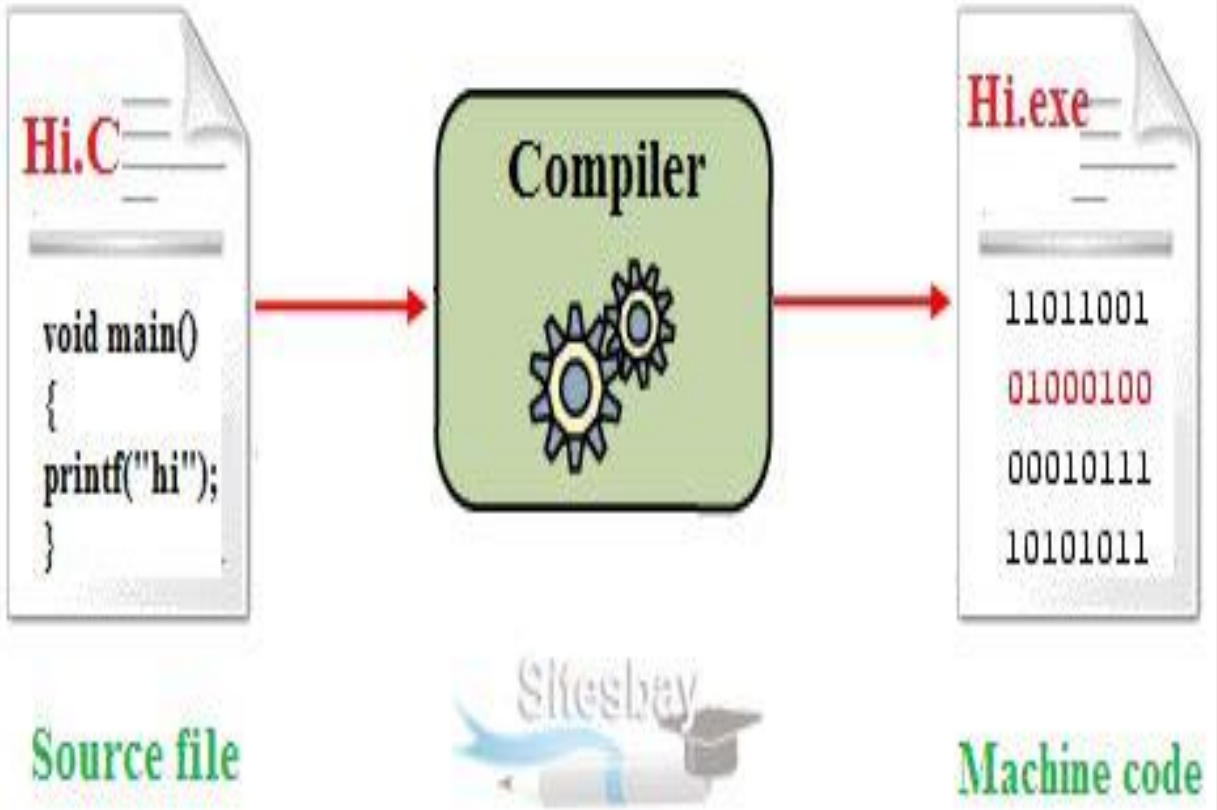
Compiler vs Interpreter ??

Compiler	Interpreter
• A compiler takes the entire program in one go.	• An interpreter takes a single line of code at a time.
• The compiler generates an intermediate machine code.	• The interpreter never produces any intermediate machine code.
• The compiler is best suited for the production environment.	• An interpreter is best suited for a software development environment.
• The compiler is used by programming languages such as C, C ++, C #, Scala, Java, etc.	• An interpreter is used by programming languages such as Python, PHP, Perl, Ruby, etc.

Objectives

At the end of the lesson students should be able to

- State what is meant by **program compilation**
- Give reasons for compiling a program
- Identify the **stages** of programme compilation
- Give the **purpose** of each identified stage
- Describe the activities that take place at each stage



Introduction

Compilation is the process of translating high-level source code (e.g., in Python or Java) into low-level machine code that a computer's CPU can execute.

A compiler!

A compiler is a program that converts a source language (HL) into object code (machine language)

Why compile?

Checks for errors before runtime.

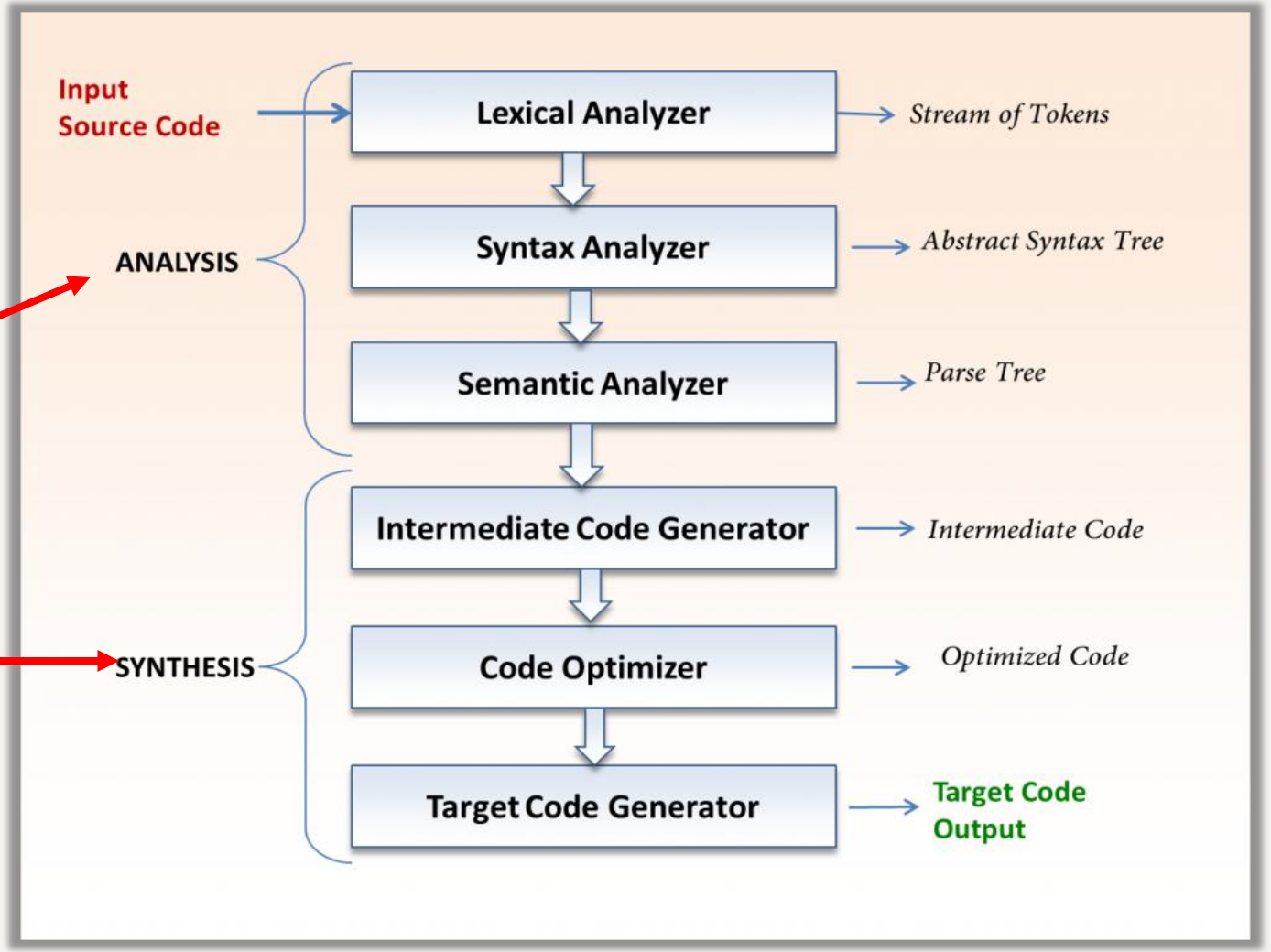
Optimizes for speed and efficiency.

Produces an executable file (e.g., .exe).

Components of the compiler and stages

The compiler is designed into two parts and compilation stages.

The first phase is the **analysis(breaking down the source code into the format can be processed)** phase while the second phase is called **synthesis(generation of target code based on the info from analysis).**




Lexical Analysis

What makes it readable and easy to understand?

```
$cat hello.cpp2

// left-to-right, order-independent,
// context-free, "import std;" default

main: () -> int = {
    hello("world\n");
}

hello: (msg: _) =
    std::cout << "hello " << msg;

$cppfront hello.cpp2 -p
hello.cpp2... ok (all Cpp2, passes safety checks)

hello.cpp

$hello
hello world
```

Lexical Analysis

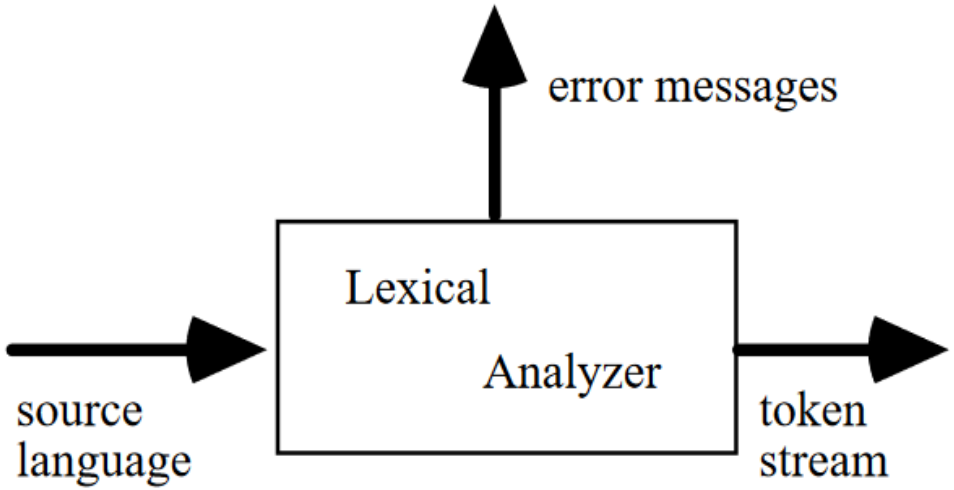
Lexical Analysis is the first phase of the compiler also known as a **scanner**

Scans source code and breaks it into tokens (e.g., keywords like "if", identifiers like variable names, operators like "+").

Removes whitespace, comments, and identifies errors (e.g., invalid characters).

Output: Token stream (e.g., "int x = 5;" → tokens: INT, IDENTIFIER(x), OPERATOR(=), LITERAL(5), SEMICOLON).

```
while (i > 0)
  i = i - 2;
```

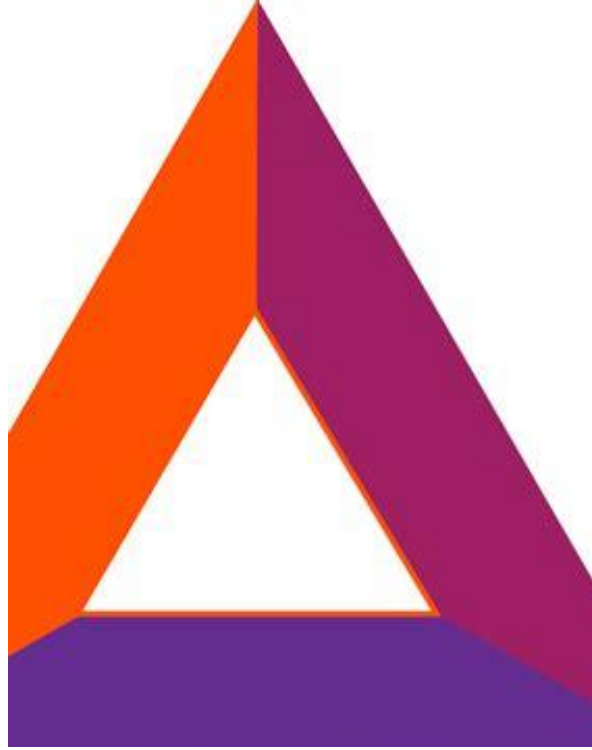


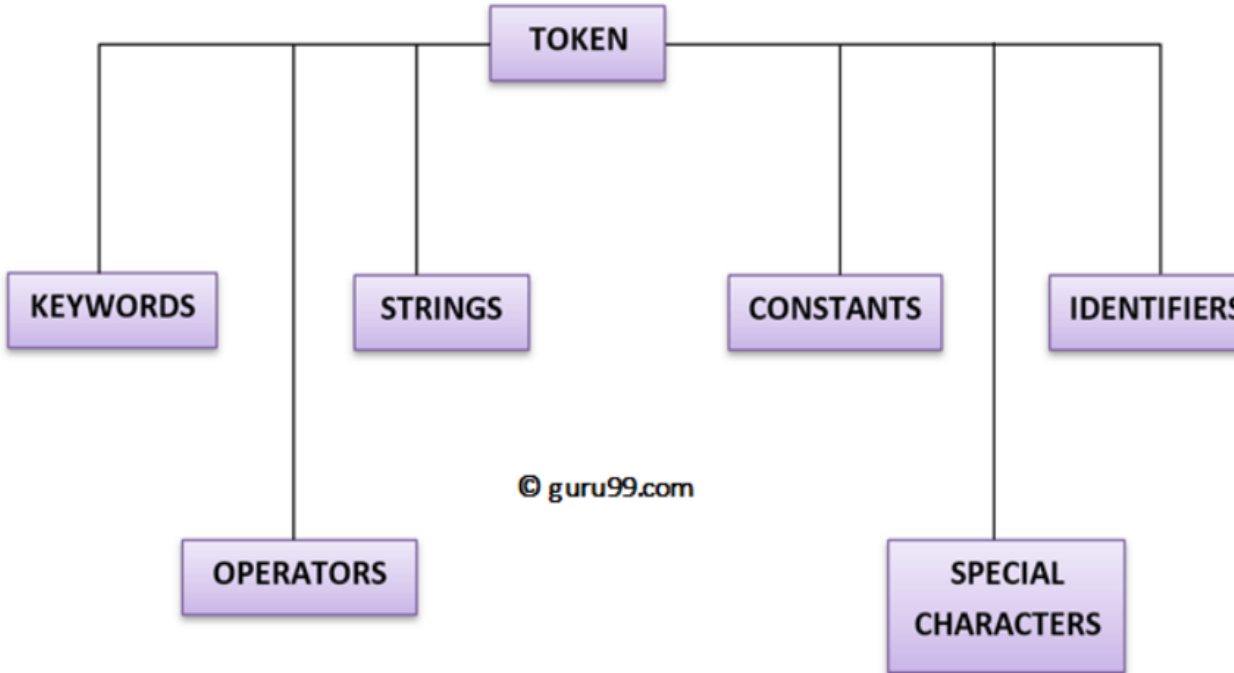

```
T_WHILE
T_LPAREN
T_IDENTIFIER
T_LESSTHAN
T_INTCONSTANT
T_RPAREN
T_IDENTIFIER
T_EQUALS
T_MINUS
T_INTCONSTANT
T_SEMICOLON
```

What are tokens?

Tokens are sequences of characters with a collective meaning. There are usually only a small number of tokens for a programming language:

- **constants** (integer, double, char, string, etc.)
- **operators** (arithmetic, relational, logical)
- **Punctuations/ Delimeter** (; , { } ())
- **Reserved words** (if, while, int, class**)**
- **Identifies:** (myVariable, calculateTotal, UserName)





Lexical Analysis: Tokens

Examples

Token	Pattern	Sample Lexeme
while	while	while
relation_op	= | != | < | >	<
integer	(0-9)*	42
string	Characters between " "	"hello"

Input string: size := r * 32 + c

Token	Lexeme
id	Size
assign	:=
id	r
arith_symbol	*
integer	32
arith_symbol	+
id	c

Lexeme

A lexeme is the actual character sequence forming a token, the token is the general class that a lexeme belongs to.

Some tokens have exactly one lexeme (e.g., the > character); for others, there are many lexemes (e.g., integer constants).

Demo

https://replit.com/@BatiMohoang/LexicalTokens

Activity 1

Exercise

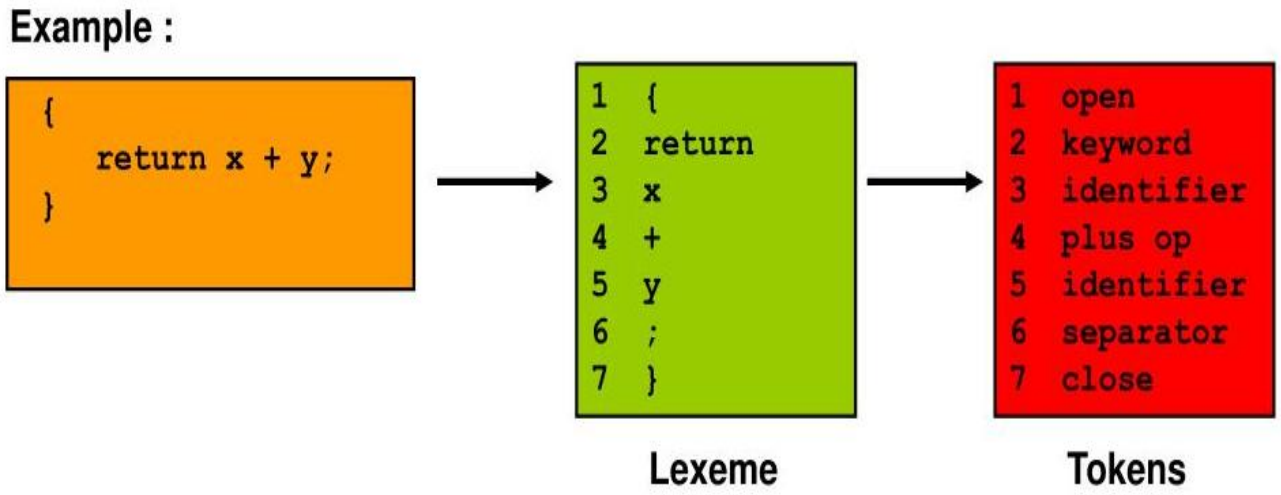
Create a table to show the token and lexemes for the following codes.

1. while (y >= t) y = y - 3

2.
```
{
    return x + y ;
}
```

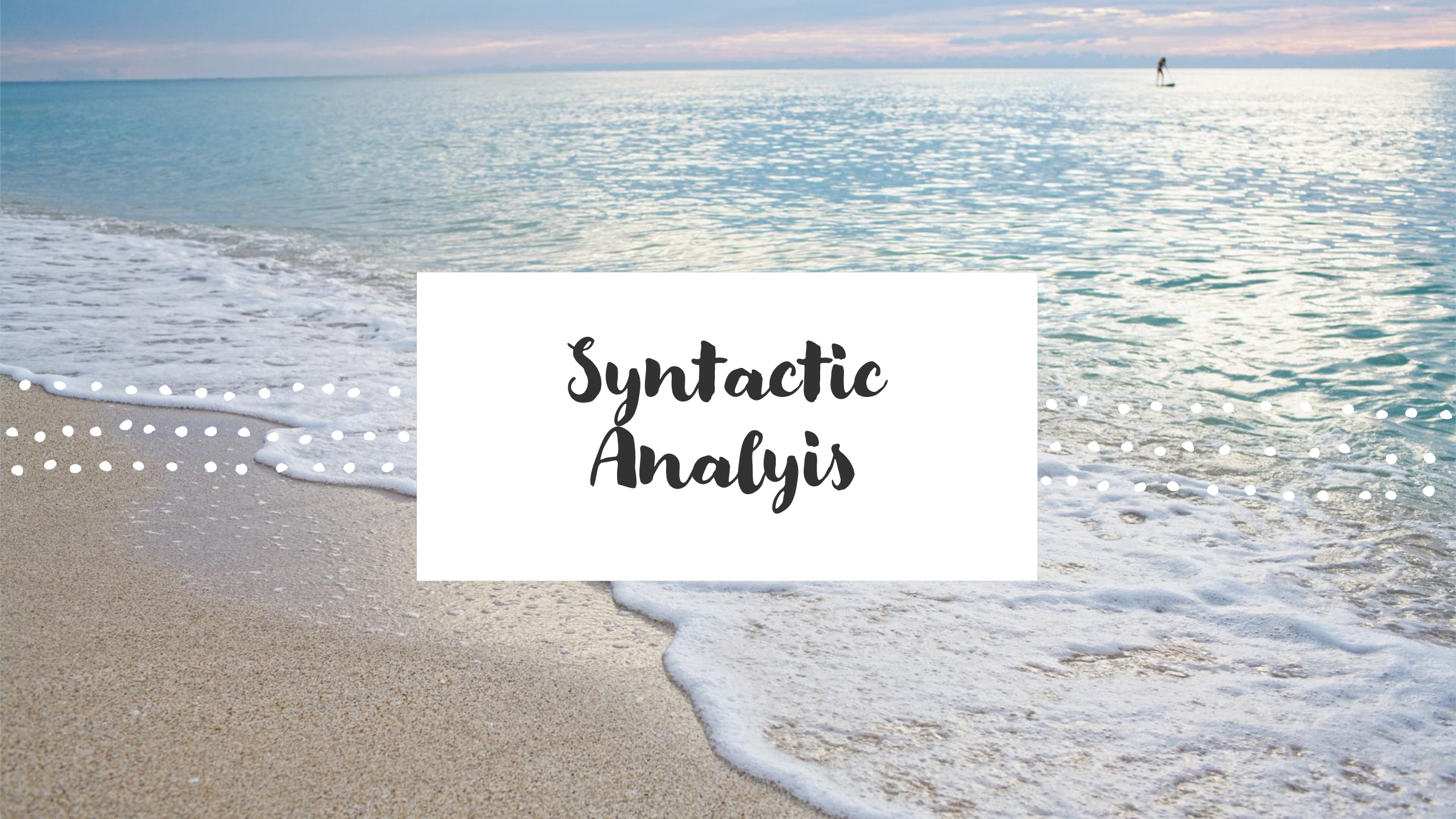
Solution

Lexeme	Token**
while	WHILE
(	LPAREN
y	IDENTIFIER
<=	COMPARISON
t	IDENTIFIER
)	RPAREN
y	IDENTIFIER
=	ASSIGNMENT
y	IDENTIFIER
-	ARITHMETIC
3	INTEGER
;	SEMICOLON



Lexical Analyser Summary

- ✓ All the comments and whitespace are removed from the program.
- ✓ High level code is turned into a series of **tokens**
- ✓ A **symbol table** is created which stores the names and addresses of all variables, constants
- ✓ Variables are checked to make sure they have been declared and to determine the data types used.
- ✓ Token is passed to the next stage



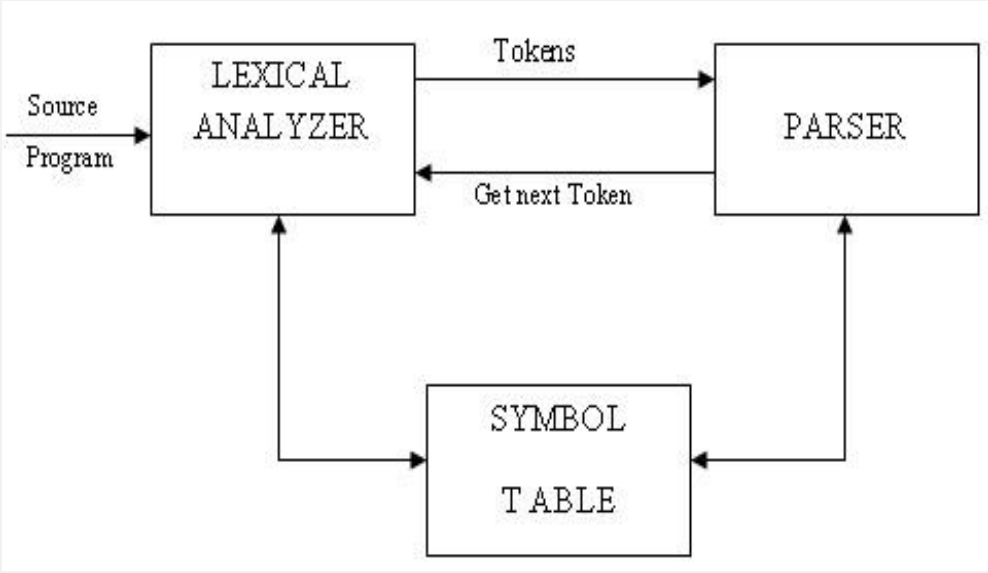
Syntactic Analyis

Syntax Analyis

The stage is also known as the parsing.

This stage the syntax analyzer (**the Parser)**

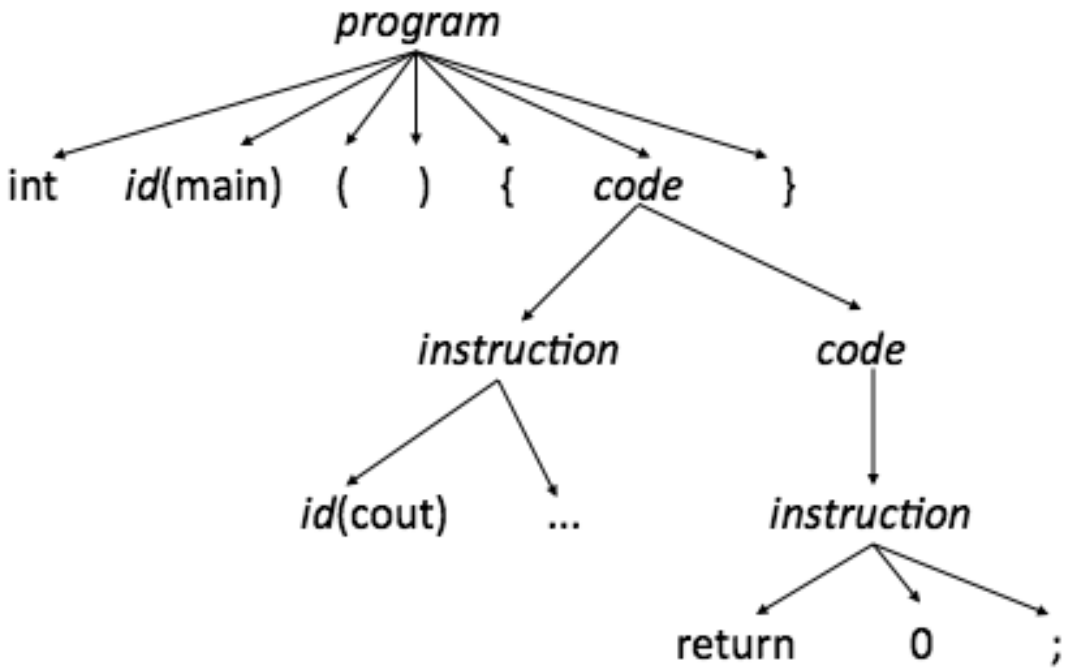
- ✓ Checks the syntax of the statements to ensure they conform to the rules of grammar for the computer language in question.
- ✓ checks that the tokens are in the correct order and follow the rules of the language.
- ✓ A **Parse tree** s then created



What is a Parse Tree?

A parse tree contains **all the tokens** that appeared in the program and possibly, a set of **intermediate rules**

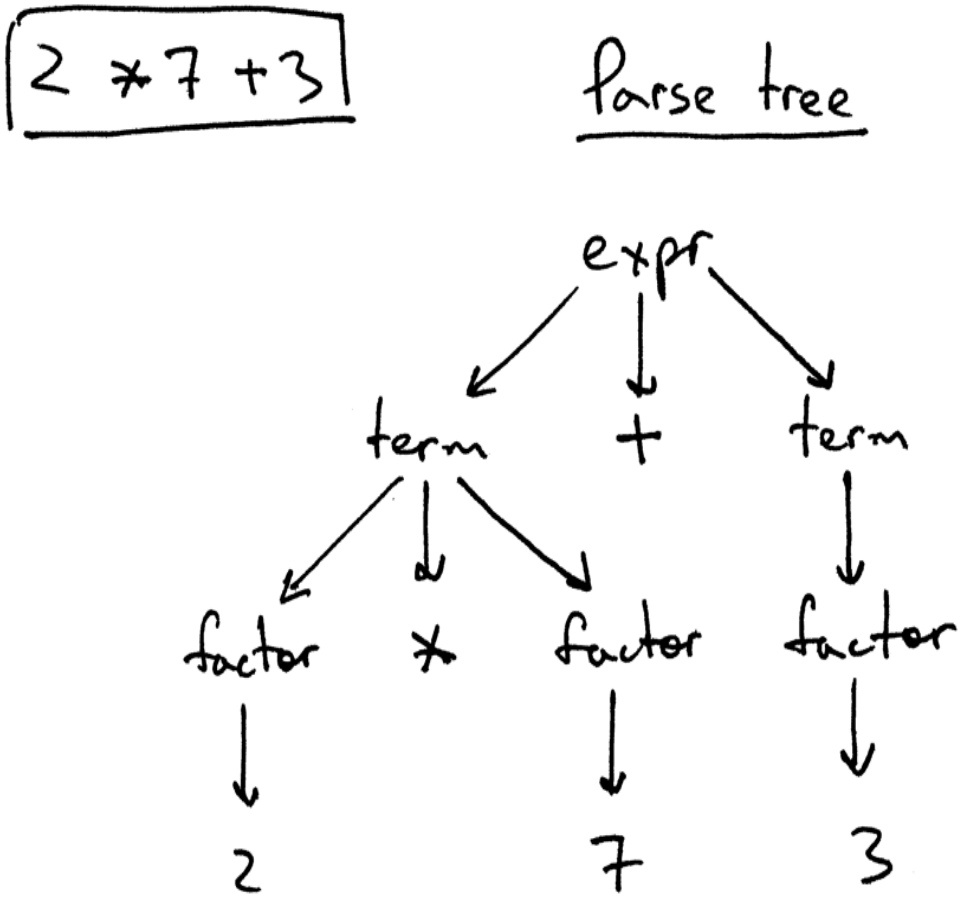
Parse tree is easier to produce for the parser since it is a direct representation of the parsing process.



Parse Tree Example

The diagram shows that the parse tree:

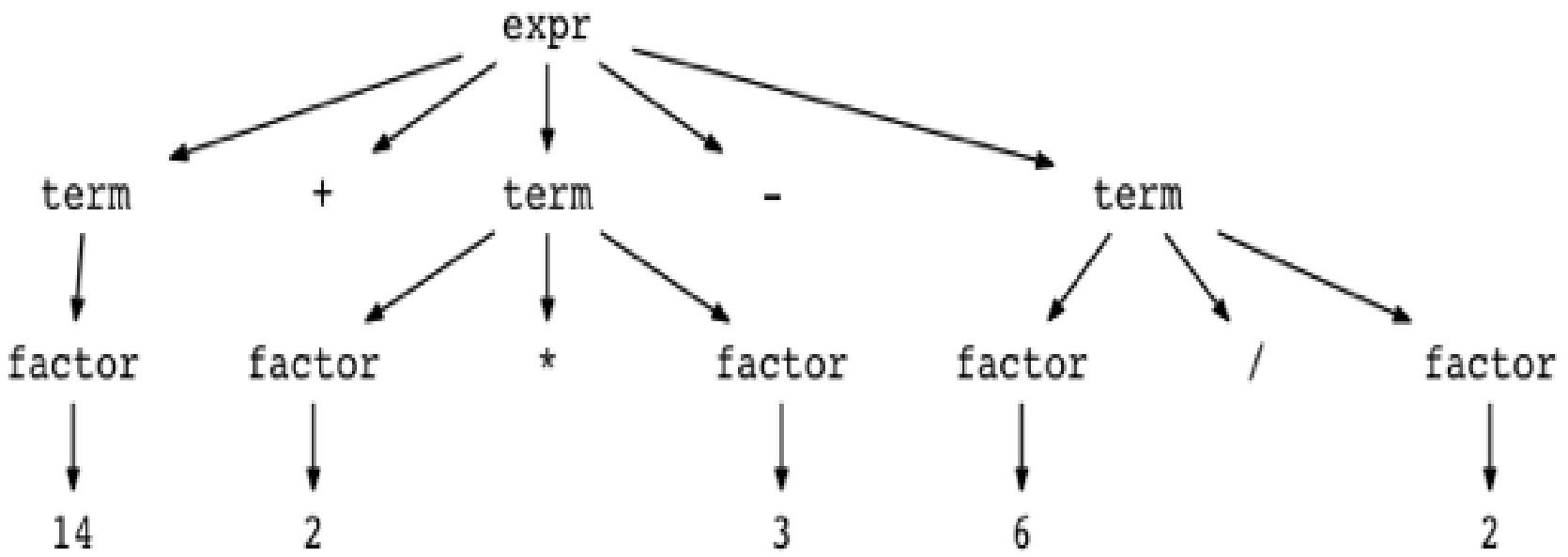
- ✓ records a sequence of rules the parser applies to recognize the input.
- ✓ The root of the tree is labeled with the grammar start symbol.
- ✓ Each **interior node** represents a grammar rule application, like *expr*, *term*, or *factor* in our case.
- ✓ Each leaf node represents a **token**



Parse Tree Example

Represent the following expression as a parse tree

14 + 2 * 3 - 6 / 2:



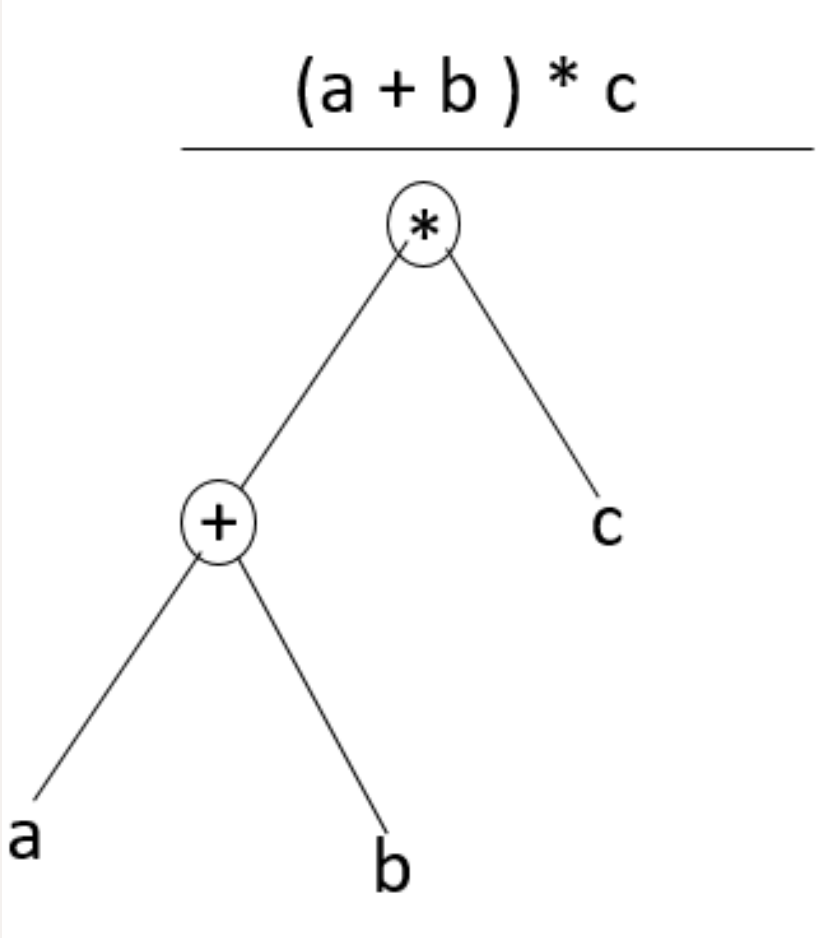
what is an abstract syntax tree? (AST)

The **abstract syntax tree (AST)** is created during the syntax stage.

This maps the structure of the program, first dropping the brackets, semicolons, etc, that were used by the programmer.

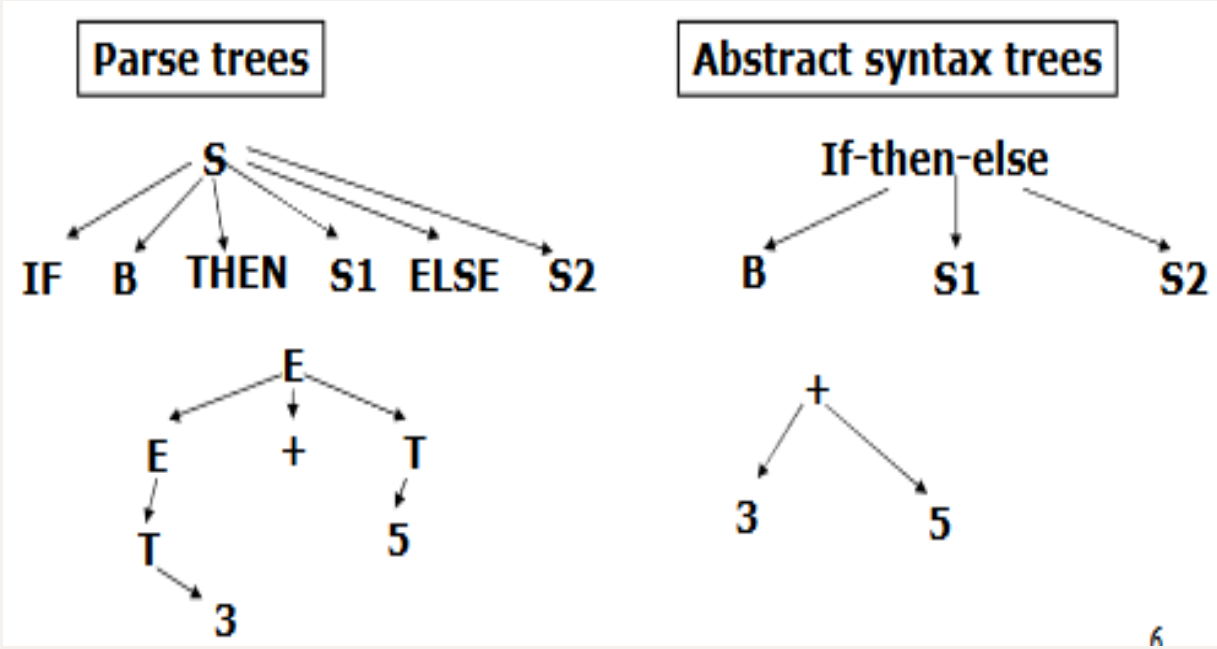
If required tokens are missing from the tree, or in the wrong place, the compiler will report an error.

The AST, instead, is a polished version of the parse tree, in **which only the information** relevant to understanding the code is maintained.



Difference between the AST and Parse Tree

- ✓ ASTs uses operators/operations as root and interior nodes and it uses operands as their children.
- ✓ ASTs do not use interior nodes to represent a grammar rule, unlike the parse tree does.
- ✓ ASTs don't represent every detail from the real syntax (that's why they're called *abstract*) - no rule nodes and no parentheses, for example.
- ✓ ASTs are dense compared to a parse tree for the same language construct.



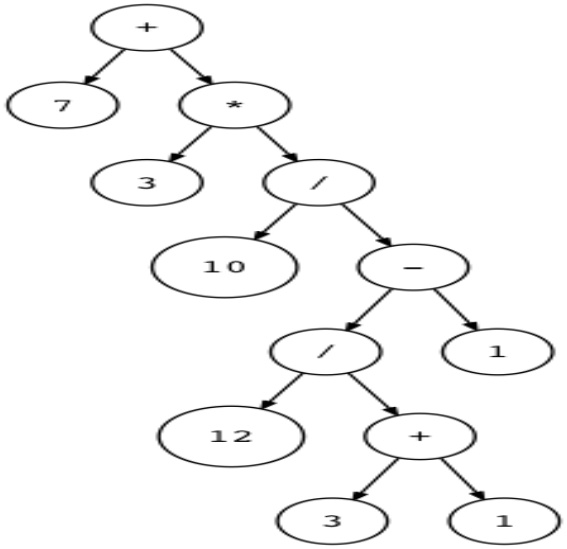
Activity

Create a Parser tree and AST from the following

1. a
2. a * b + c
3. (a+b) * c
4. "7 + 3 * (10 / (12 / (3 + 1) - 1))"

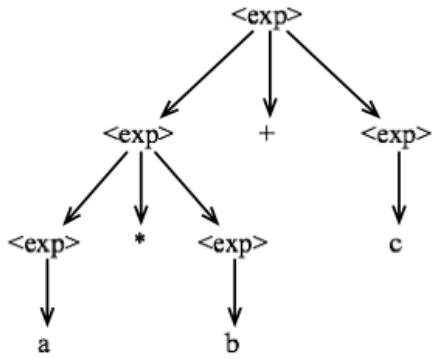
Solution

"7 + 3 * (10 / (12 / (3 + 1) - 1))"

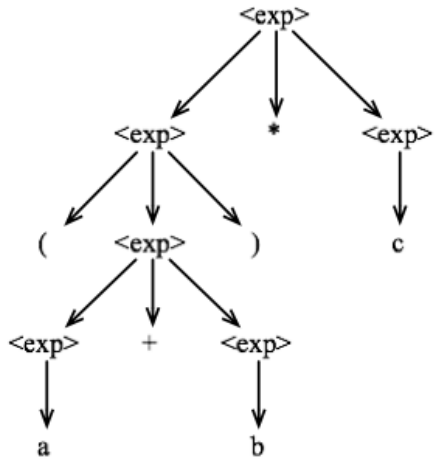


"a"

"a * b + c"



"(a + b) * c"



Syntax Analyser Summary

Checks code is written in a valid syntax

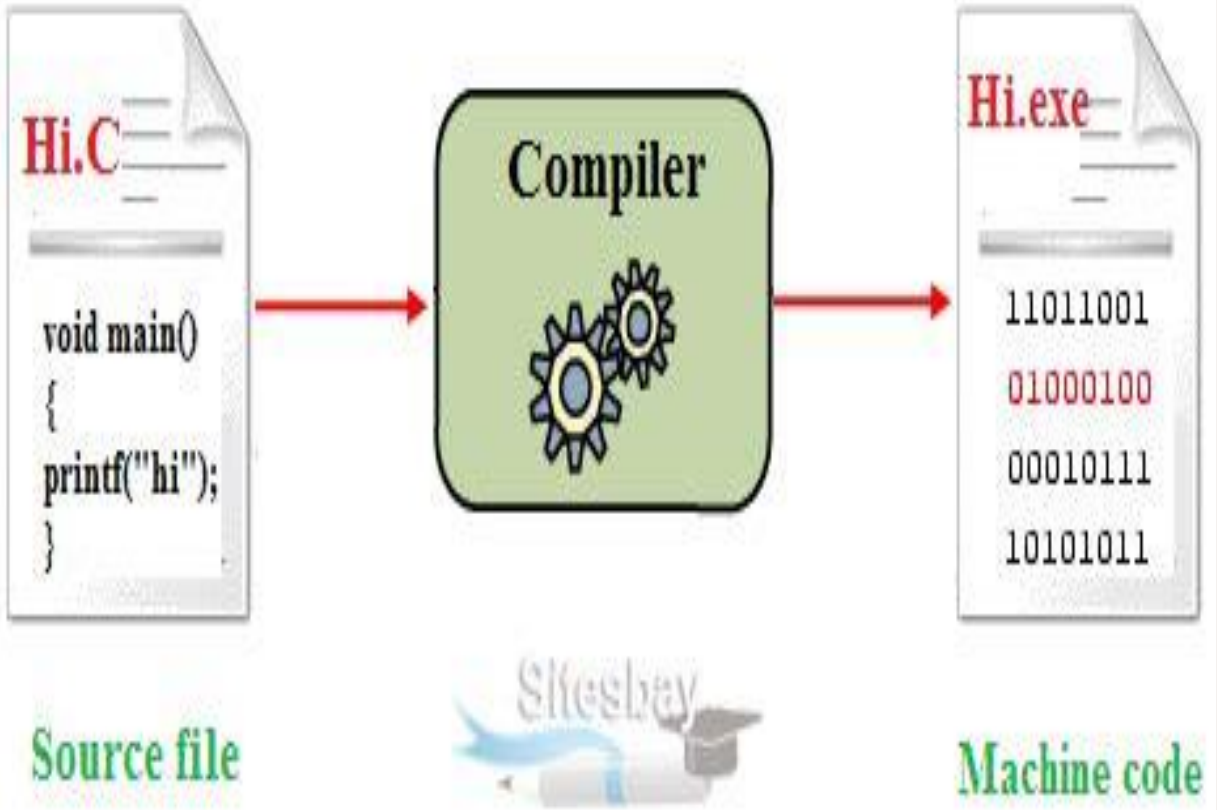
- ✓ Compiles a list of errors for user where code does not follow the languages rules
- ✓ Produces a parser tree
- ✓ It produces an abstract syntax tree, which represents the program.


Program
Compilation

Objectives

At the end of the lesson students should be able to

- State what is meant by program compilation
- Give reasons for compiling a program
- Identify the stages of programme compilation
- Give the purpose of each identified stage
- Describe the activities that take place at each stage



Review

In pairs discuss the following concepts together

- ✓ Describe the Lexical analysis stage
- ✓ Describe the Syntactic analysis stage

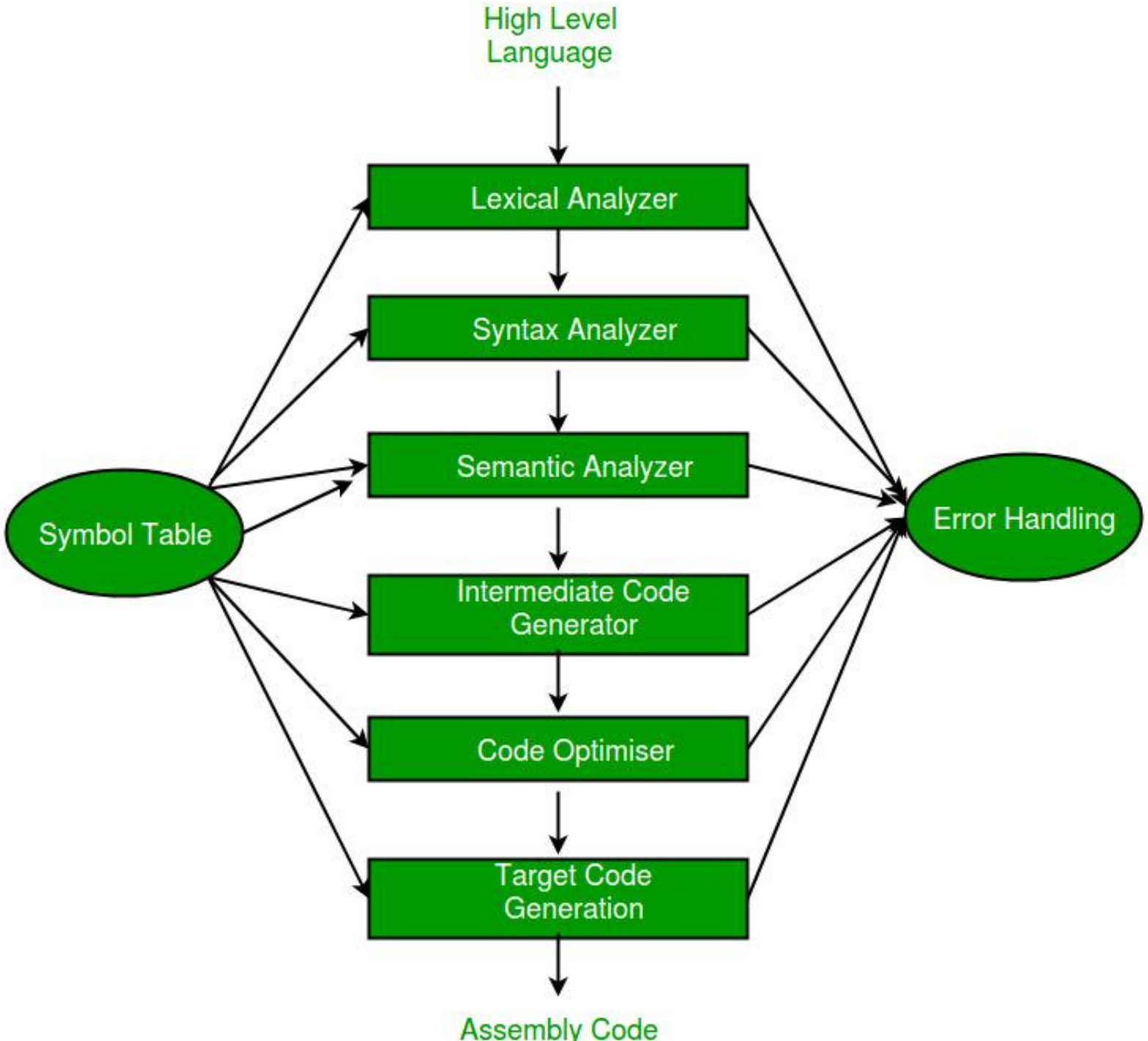
LEXICAL ANALYSIS

- ✓ All the comments and whitespace are removed from the program.
- ✓ High level code is turned into a series of **tokens,** using reserved words that the computer is able to pick out.
- ✓ A **symbol table** is created which stores the names and addresses of all variables, constants
- ✓ Variables are checked to make sure they have been declared and to determine the data types used
- ✓ Token is passed to the next stage

SYNTAX ANALYSIS

Checks code is written in a valid syntax

- ✓ Compiles a list of errors for user where code does not follow the languages rules
- ✓ Produces a parser tree
- ✓ It produces an abstract syntax tree, which represents the program.


High Level Language
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Code Optimiser
Target Code Generation
Assembly Code
Symbol Table
Error Handling


Semantic Analyis

Semantic Analyis

The semantic analyser is mainly concerned with what the program means and how it executes.

It checks

- ✓ the semantic consistency of the code.
- ✓ It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent.
- ✓ whether the code is conveying an appropriate meaning.
- ✓ Type mismatches, incompatible operands, a function called with improper arguments, an undeclared variable, etc

Type checking is an important aspect of semantic analysis where each operator should be compatible with its operands.

Semantic Analyis

Summary

- ✓ Helps you to store type information gathered and save it in symbol table or syntax tree
- ✓ Allows you to perform type checking
- ✓ In the case of type mismatch, where there are no exact type correction rules which satisfy the desired operation, a semantic error is shown
- ✓ Collects type information and checks for type compatibility
- ✓ Checks if the source language permits the operands or not
- ✓ Variables are checked to make sure they have been correctly declared and contain the correct data type.
- ✓ Updates to the symbol table

```
float x = 20.2;
float y = x*30;
```

In the above code, the semantic analyzer will typecast the integer 30 to float 30.0 before multiplication

What type of error?

Code	Error type
Y=(x*c) +(d/b));	
Y=(xc) +(d/b)	


Immediate Code
Generation

Code Generation

A separate program is created that is distinct from the original source code.

The code generated is the object code, which is the binary equivalent of the source code.

This is the executable version of the code, before linked libraries are included.

Code generation is a major distinguishing feature between compilation and interpretation; **interpreters do not produce a separate executable file.**

The symbol table store

During lexical and syntax analysis, a table of variables and labels has been built up which includes details of the variable name, its type and the block in which it is valid.

When a variable is encountered during code generation, the address of the variable is computed and stored in the symbol table

Intermediate code generation

This phase involves generating an intermediate code that can be translated into the final machine code.

Intermediate language can be many different languages and the designer of the compiler decides this.

Intermediate representation can be in various forms such as

- ✓ three-address code (language independent),
- ✓ byte code,
- ✓ stack code.

Activity

In three (4 groups) research on the following and present your finding to the class

- ✓ three-address code (language independent),
- ✓ byte code
- ✓ stack code
- ✓ Postfix notation

Intermediate Code Generation

Some programming language have well defined intermediate languages

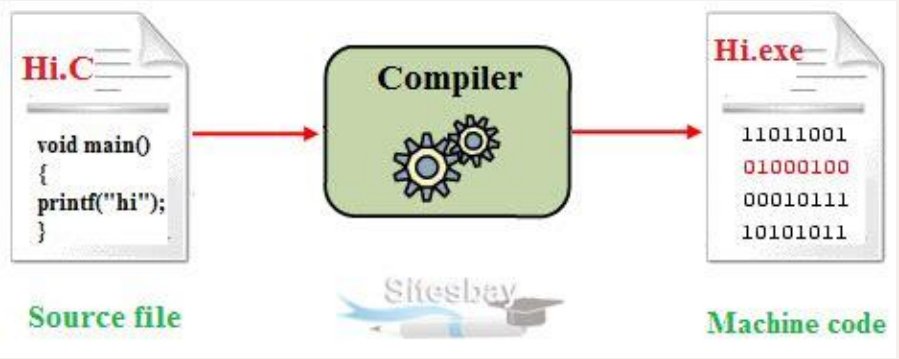
- ✓ Java-java virtual machine
- ✓ Prolog- warren abstract machine
- ✓ Any byte-code emulators to execute instructions in these intermediate languages

Code Optimization

This is an optional phase that improves or optimizes the intermediate code to enable the output to be run faster.

This phase eliminates unnecessary code lines and ensures that the output occupies less space.

Types of Code Optimization:

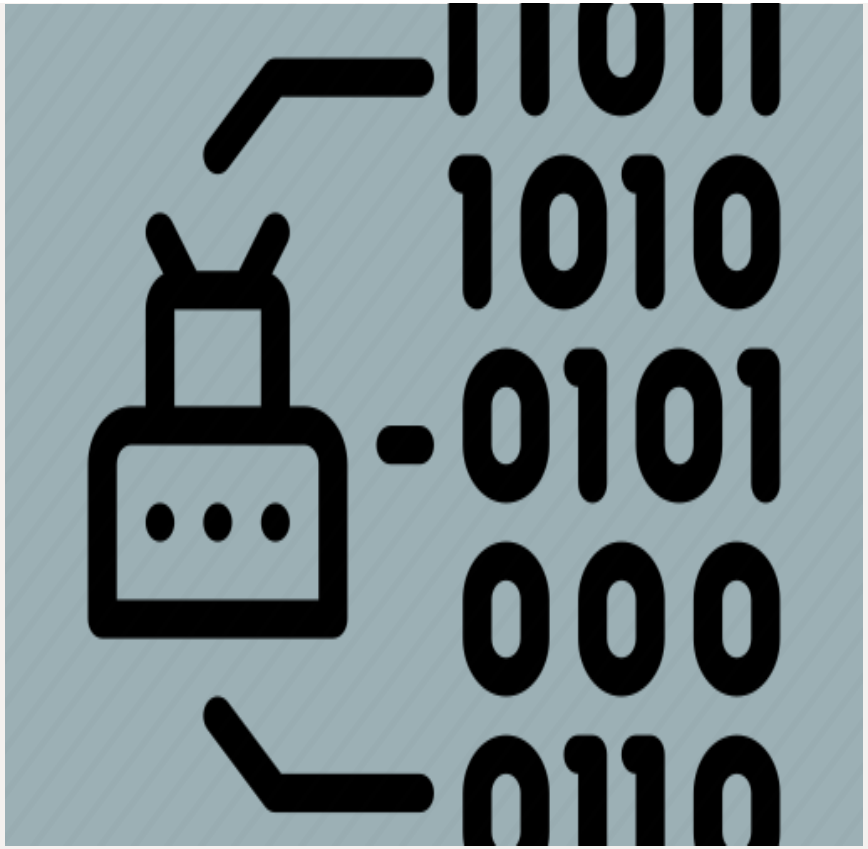


Machine Independent Optimization: This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.

1. **Machine Dependent Optimization:** Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

Code generation

This is the final stage that transforms the optimized code into the desired machine code.

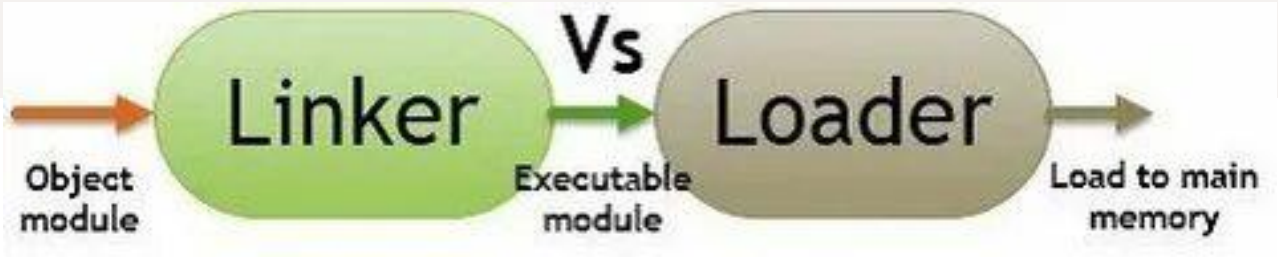


Linkers, loaders and library routines

Programs are usually built up from small, selfcontained blocks of code called **subprograms**, **procedures** or **modules**. These modules are stored in an area known as a program library and the modules made available are library routines.

Library routines have already been written and tested so the programmer is saved a lot of work; the code exists in compiled form so it does not need to be re-compiled when it is used.

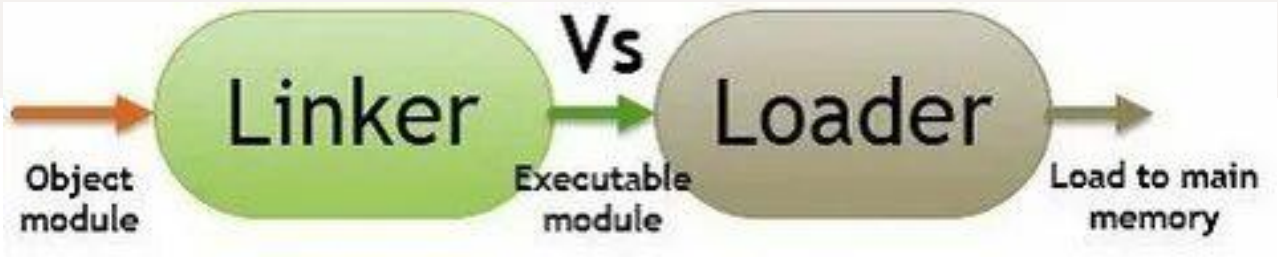
Each of these may be separately compiled so there is the separate compiled object code.



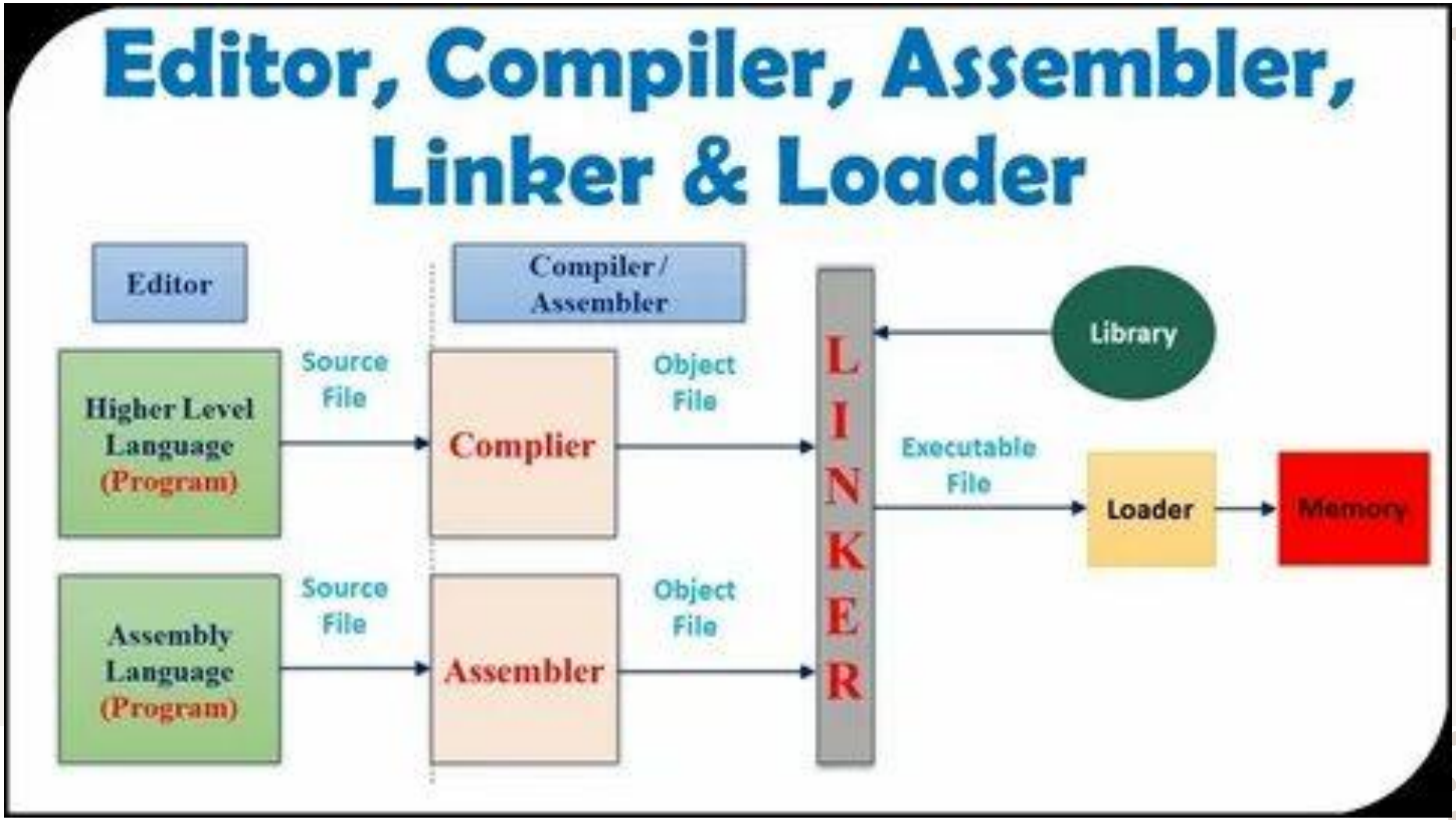
Linkers, loaders and library routines

However, a problem is variable names and memory addresses are different from one use of the library routine to the next.

These problems are solved by two **utility programs**. A **loader** loads all the modules into memory and sorts out difficulties such as changes of addresses for the variables. A **linker** links the modules together by making sure that references from one module to another are correct.

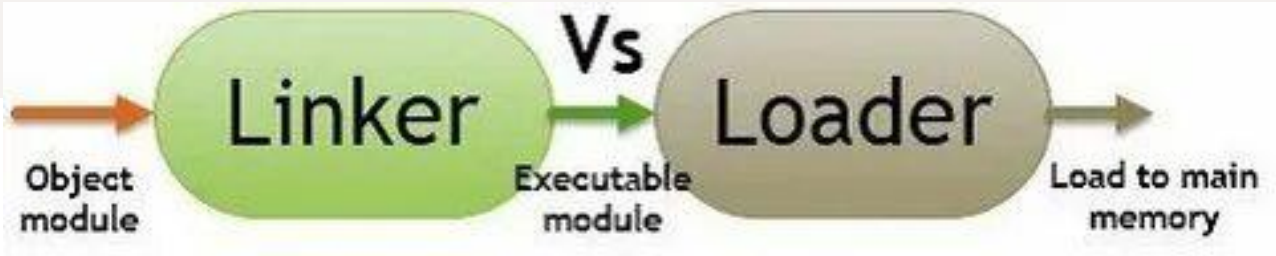


Linkers, loaders and library routines



Linkers, loaders and library routines

Linker and Loader are both important components in the software development process. The **Linker** is used during the compilation process to link object files into a single executable file, while the **Loader** is used at runtime to load the executable file into memory and prepare it for execution

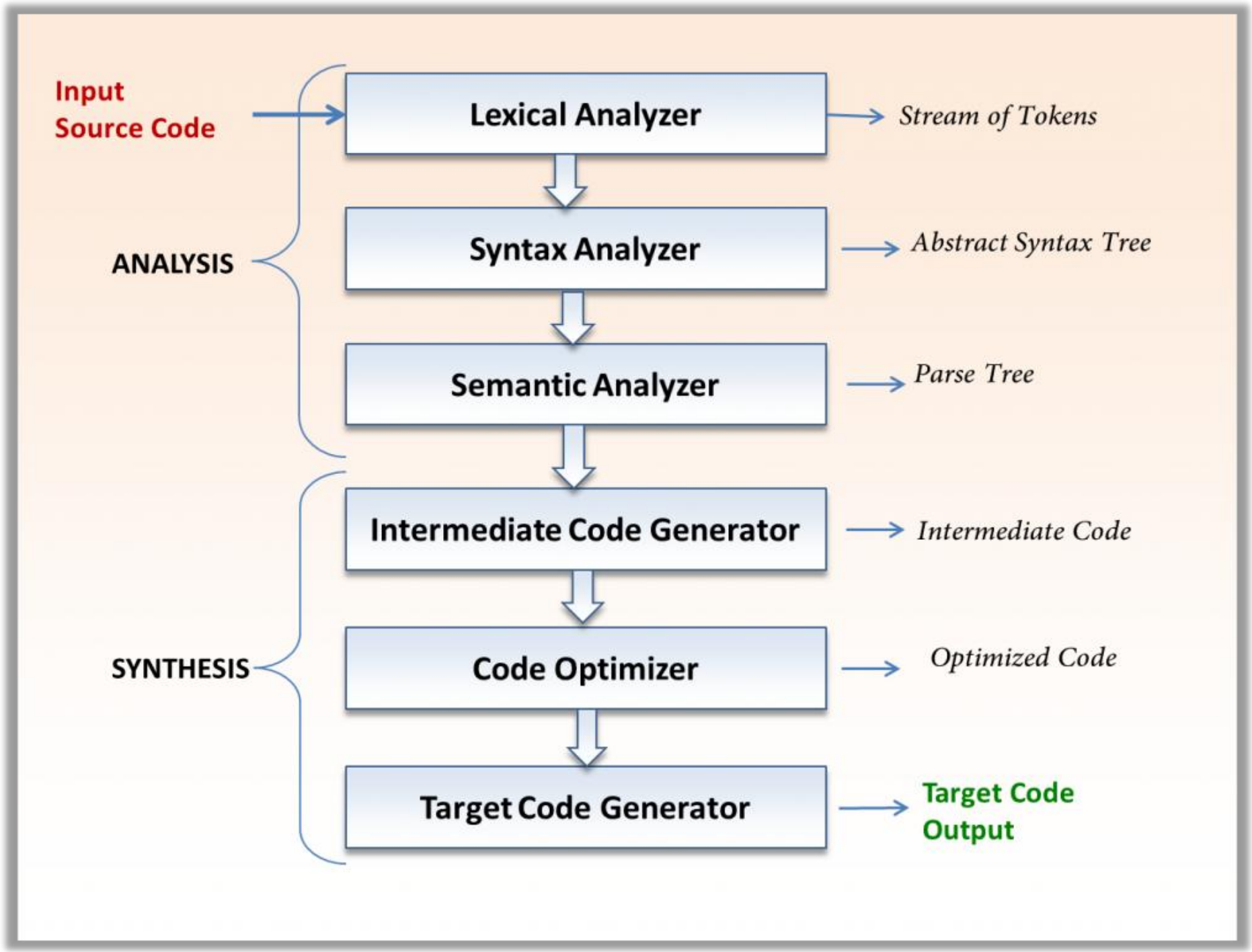


Recap

- **What is a compiler?**
- **List stages of program compilation**
- **Why do we need code optimization stage?**
- **What is a syntax error**

ACTIVITY: Role play stages of program compilation 30 Min

- https://www.guru99.com/compiler-design-phases-of-compiler.html


Input
Source Code
ANALYSIS
SYNTHESIS
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Code Optimizer
Target Code Generator
Stream of Tokens
Abstract Syntax Tree
Parse Tree
Intermediate Code
Optimized Code
Target Code
Output