Compilers: Analysis Phase

Module - 5

Topics to Cover

Introduction to compilers

Phases of compilers

Lexical Analysis- Role of Finite State Automata in Lexical Analysis, Design of Lexical analyzer, data structures used.

Syntax Analysis- Role of Context Free Grammar in Syntax analysis

Types of Parsers:

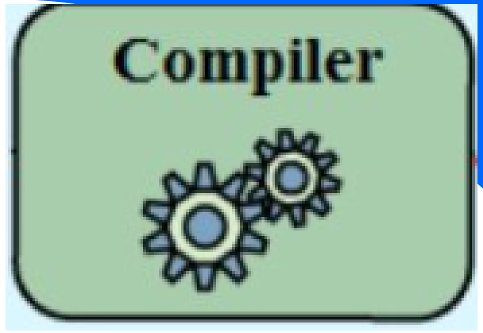
a. Top down parser- LL(1)

b. Bottom up parser- SR Parser,

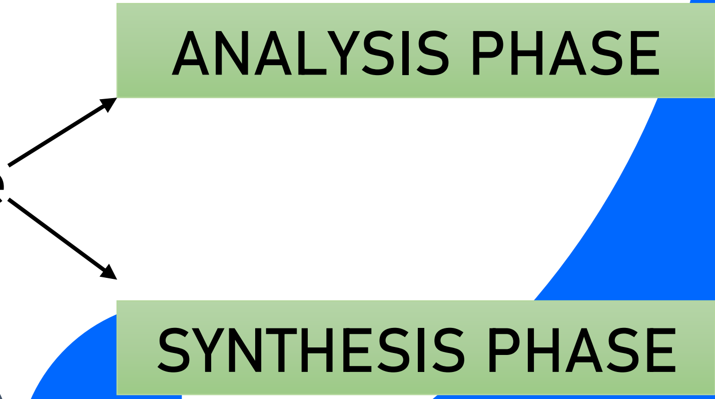
c. Operator precedence parser, SLR.

Semantic Analysis, Syntax directed definitions.

Introduction to Compiler



- ❖ The compiler is an application that can convert programs written in higher level language to lower level language that the computer can understand and execute. The first compiler was thought to have been designed way back in 1952.
- ❖ Thus, A compiler typically maps a source program into a semantically equivalent target program.
- ❖ Ideally, the compiler does this mapping in two phase



PHASES OF COMPILER

There are two main phases in the compiler.

1. Analysis - Front end of a compiler and is Language Dependent
2. Synthesis - Back end of a compiler and is Language Independent.

Analysis Phase

1. This phase breaks up the source program into its constituent pieces and imposes a Grammatical Structure on them.
2. It then uses this structure to create an **Intermediate Representation** of the source program.
3. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages so that user ca take corrective action.
4. The analysis part also collects information about the source program and stores it in a data structure called a **SYMBOL TABLE**, which is passed along with the intermediate representation to the synthesis Phase.

PHASES OF COMPILER

There are two main phases in the compiler.

1. Analysis - Front end of a compiler and is Language Dependent
2. Synthesis - Back end of a compiler and is Language Independent.

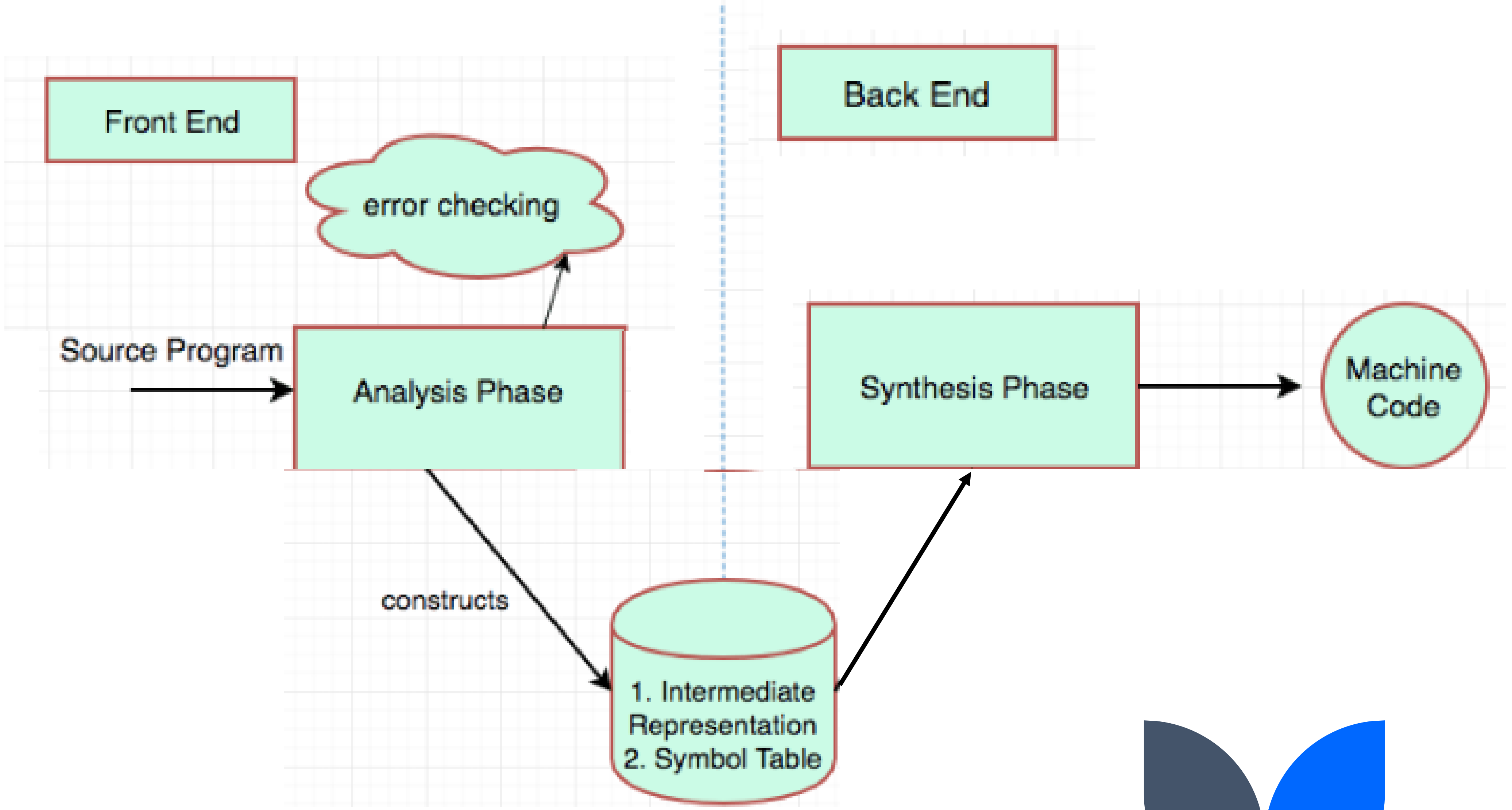
Synthesis Phase

1. This phase constructs the desired target program from the

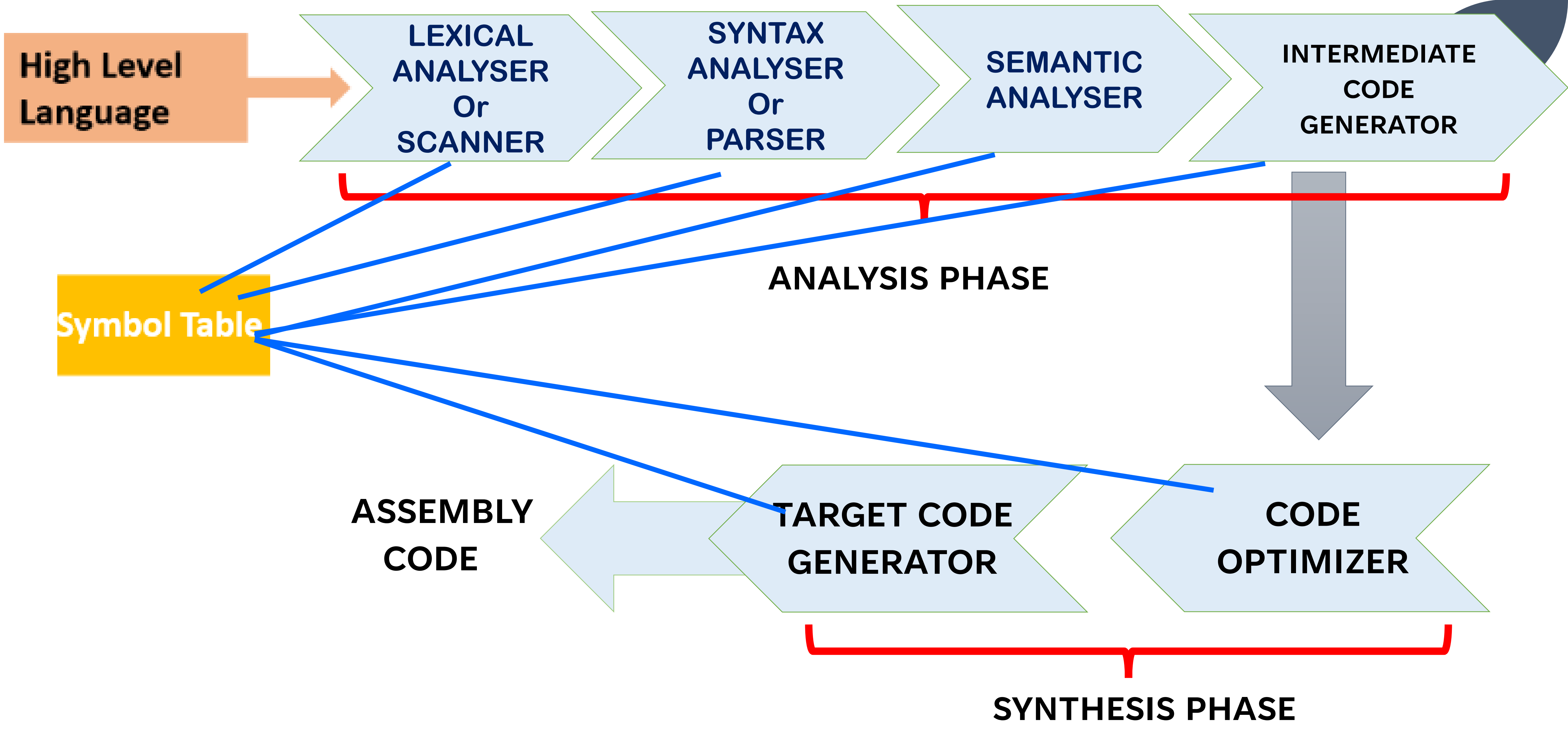
    Intermediate Representation

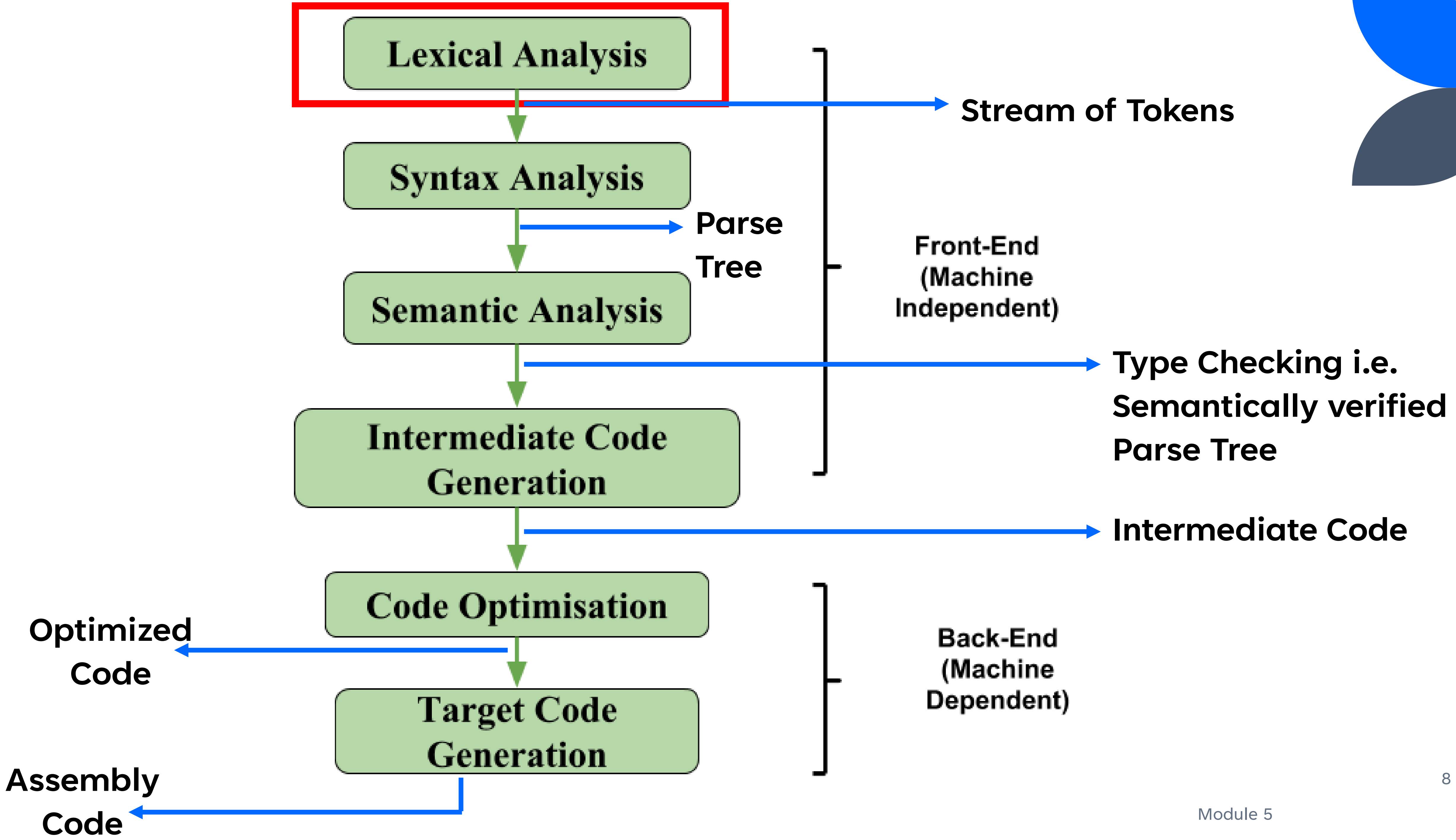
    Information in the symbol table

Analysis and Synthesis Phase



Phases of Compiler




Lexical Analysis
Stream of Tokens
Syntax Analysis
Parse Tree
Semantic Analysis
Front-End (Machine Independent)
Type Checking i.e. Semantically verified Parse Tree
Intermediate Code Generation
Intermediate Code
Code Optimisation
Optimized Code
Back-End (Machine Dependent)
Target Code Generation
Assembly Code

A lexeme is a sequence of characters of a program that is grouped together as a single unit. When a compiler or interpreter reads the source code of a program, the compiler breaks it down into smaller units called lexemes.

Lexical Analysis is the first phase when compiler scans the source code. This phase is handled by the **Lexical Analyzer** which is also **called as Scanner**.

This process is from left to right, character by character, and these characters are grouped into tokens. Thus, the character stream from the source program is grouped in meaningful sequences called **LEXEMES.**

For each Lexeme, the Lexical Analyzer produces as output a TOKEN of the form

(token-name, attribute-value)

The **Symbol table** is generated in this phase and populated with tokens generated.

A **symbol table** is typically a data structure that holds a record for each identifier in the source code.

The output of this phase is Stream of Tokens

Types of Lexemes

Operators: These are the symbols that can be used to perform mathematical or logical operations such as addition (+), subtraction (-), multiplication (*), division (/), etc.

Identifiers: These are the name of variables, methods, and functions. For example, If there is a statement ‘int a = 5;’ in C++. Here ‘a’ is an Identifier.

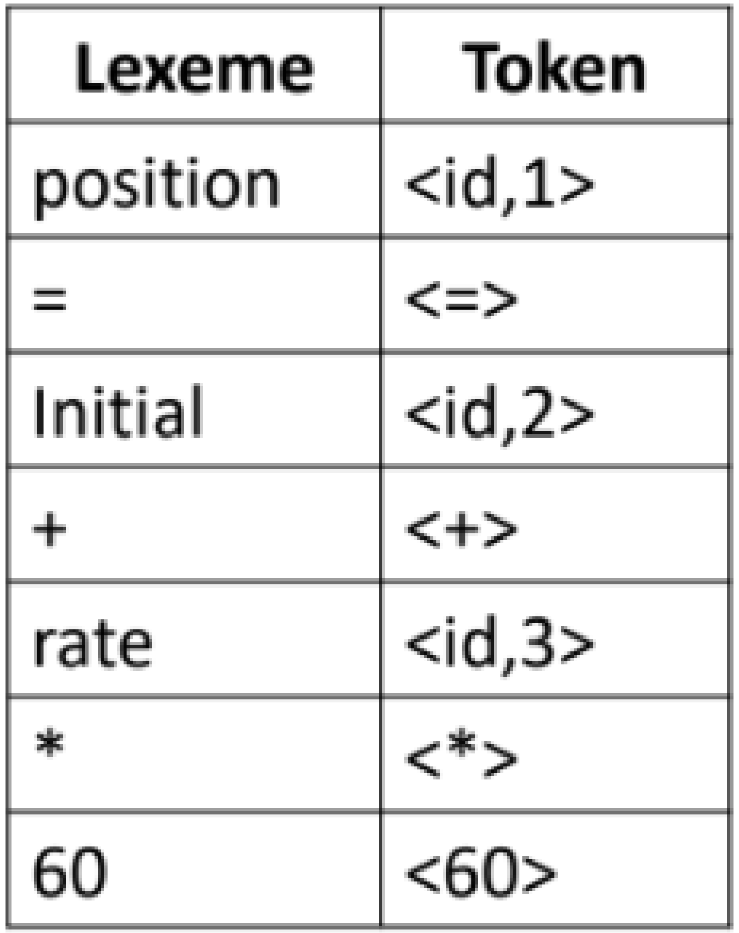
Punctuation: These are the symbols that are used to separate and group different parts of the program. For example, semicolon (;), comma (,), braces ({}), etc.

Keywords: These are used to identify programming constructs such as loops, conditionals, and functions. For example, ‘if’, ‘else’, ‘while’, etc.

Literals: These are the values that can be used in the source code. For example, If there is a statement ‘int a = 5;’ in C++. Here ‘5’ is literal. Literals can be of any given datatype.

EXAMPLE

position = initial + rate * 60



Lexeme	Token
position	<id,1>
=	<=>
Initial	<id,2>
+	<+>
rate	<id,3>
*	<*>
60	<60>

a = b / 2

Identifier: ‘a’

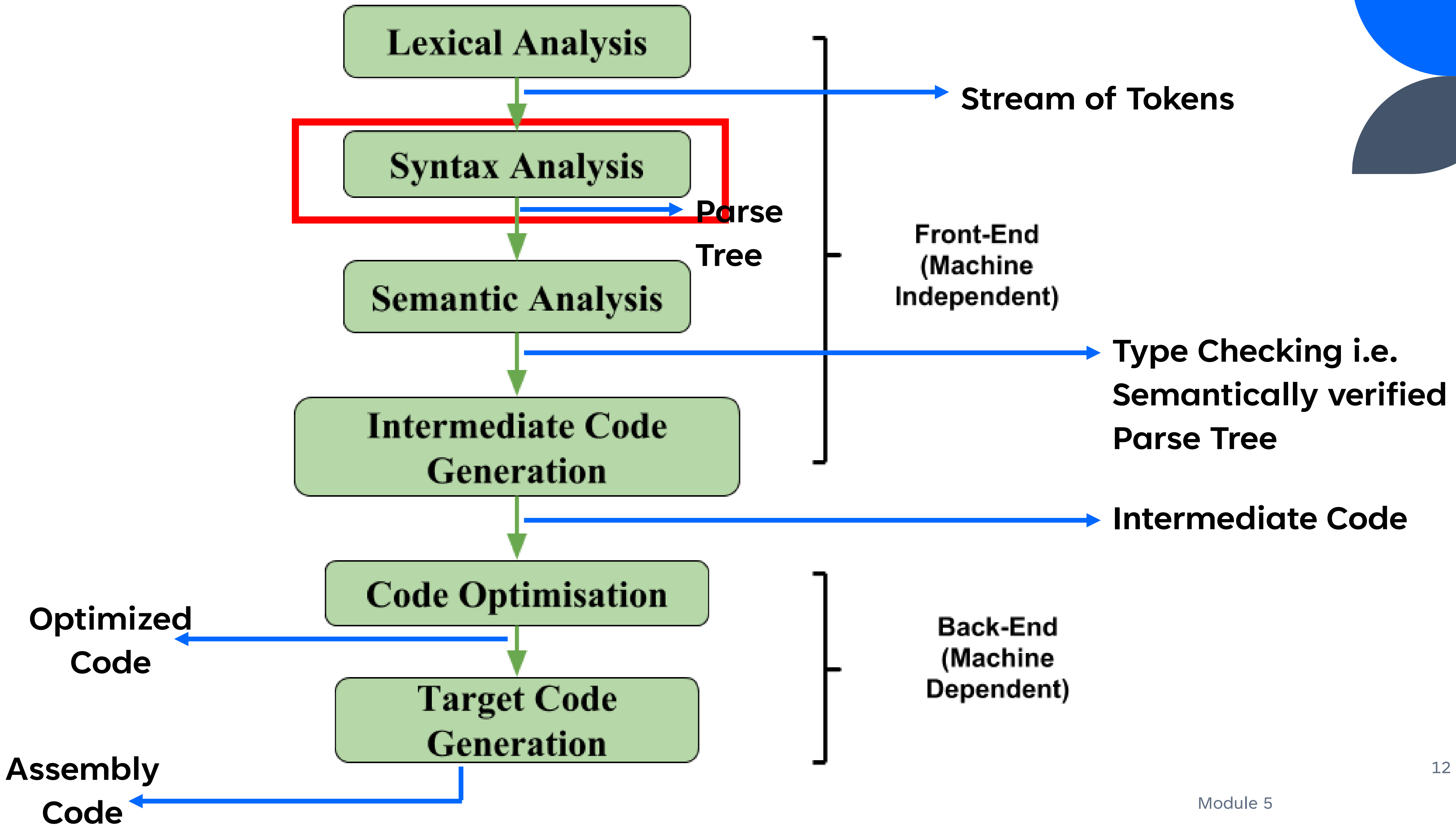
Operator: ‘=’

Identifier: ‘b’

Operator: ‘/’

Literal: ‘2’

Lexeme	Token
a	<id, 1>
=	< = >
b	<id, 2>
/	< / >
2	< 2 >


Lexical Analysis
Stream of Tokens
Syntax Analysis
Parse Tree
Semantic Analysis
Front-End (Machine Independent)
Type Checking i.e. Semantically verified Parse Tree
Intermediate Code Generation
Intermediate Code
Code Optimisation
Optimized Code
Back-End (Machine Dependent)
Target Code Generation
Assembly Code

Syntax Analysis

1. The next phase of the compiler is Syntax Analysis or Parsing. This phase is handled by the **Syntax Analyzer. Or Parser**

2. The Parser uses the components of the tokens produced by Lexical Analyzer to create a tree (Syntax Tree) – like **Intermediate Representation** that **depicts the grammatical structure of the token stream.**

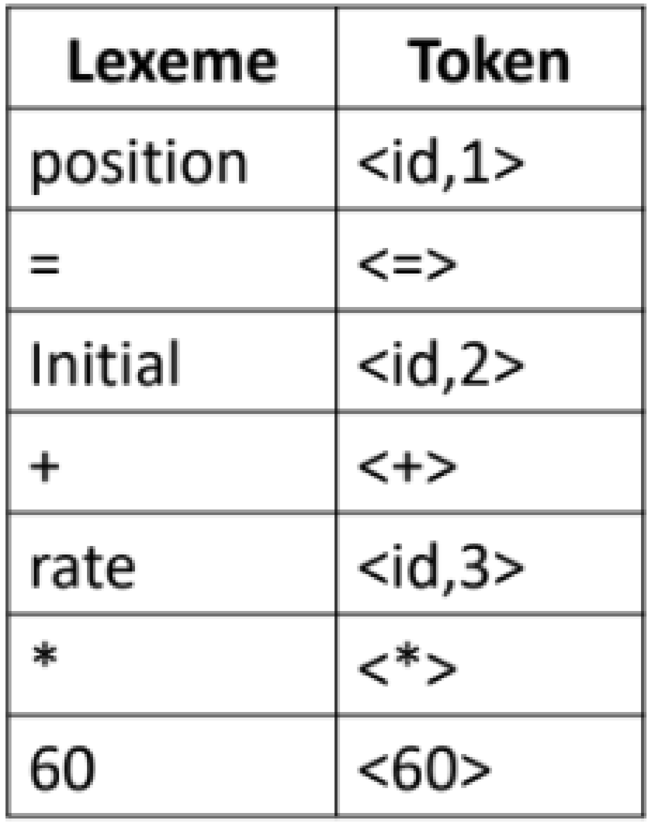
3. In a Syntax Tree a.k.a Parse Tree shows the order in which the operations are performed where;

   Interior node: Represents an operation
   Children nodes: Represents argument of that operation

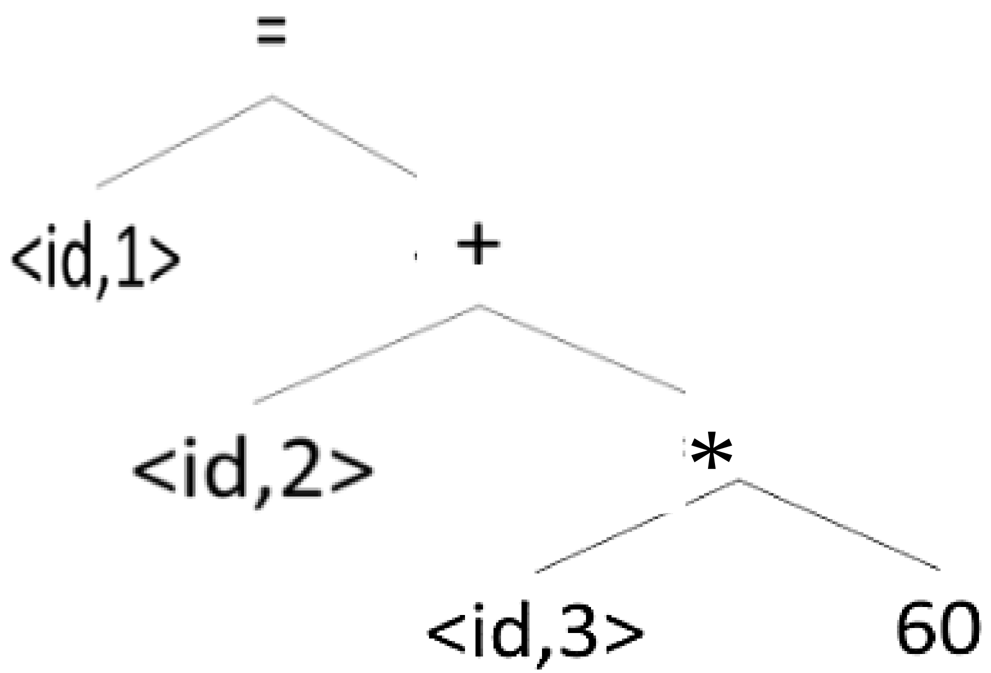
4. This is primarily done to ensure that the syntax of the given statements is correct and adheres to the language's pre-defined rules. If the syntax is incorrect, it generates a syntax error.

LEXICAL ANALYSER Or SCANNER → **STREAM OF TOKENS** → **SYNTAX ANALYSER Or PARSER** → **SYNTAX TREE / PARSE TREE**

position = initial + rate * 60

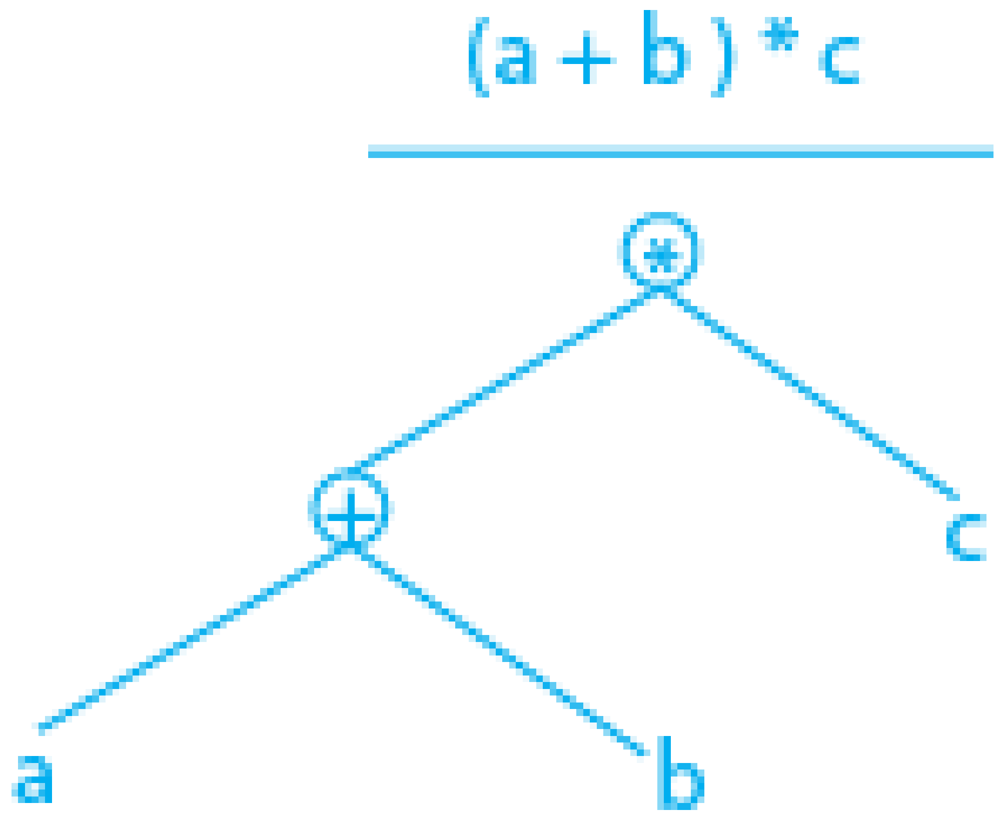


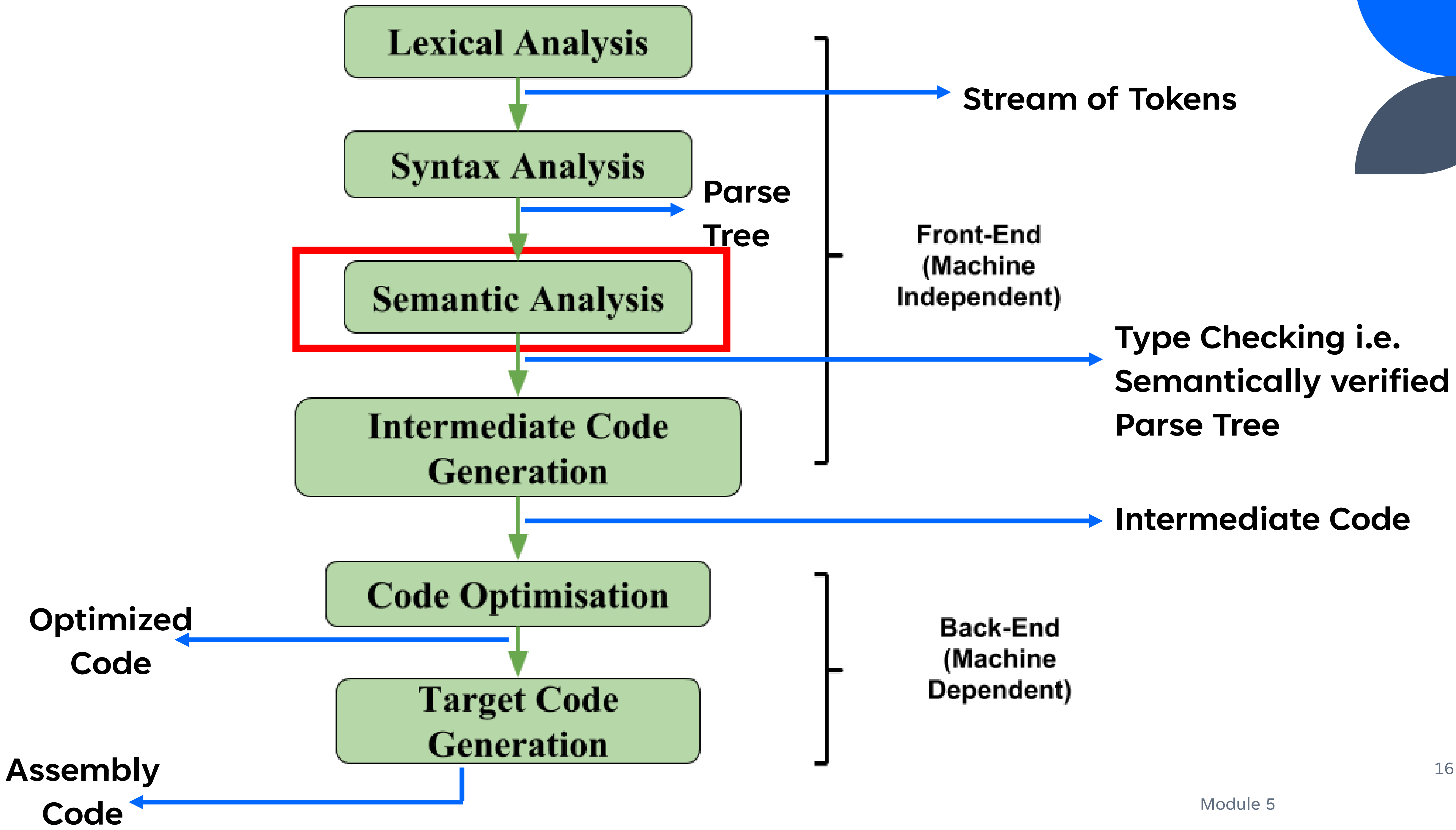
Lexeme	Token
position	<id,1>
=	<=>
Initial	<id,2>
+	<+>
rate	<id,3>
*	<*>
60	<60>



EXAMPLE: -

(a+b)*c




Lexical Analysis
Stream of Tokens
Syntax Analysis
Parse Tree
Semantic Analysis
Front-End (Machine Independent)
Type Checking i.e. Semantically verified Parse Tree
Intermediate Code Generation
Intermediate Code
Code Optimisation
Optimized Code
Back-End (Machine Dependent)
Target Code Generation
Assembly Code

SEMANTIC ANALYSIS

1. The **semantic analyzer** checks if the source code is semantically consistent, that is, if it conveys the intended meaning, using the syntax tree from the previous phase and the symbol table.

2. It also gathers TYPE INFORMATION and saves it in either SYNTAX TREE or the SYMBOL TABLE, for subsequent use for during Intermediate Code Generation.

3. Type-checking is one of the most important functions of the semantic analyzer where the compiler checks that each operator has matching operands.

4. For example, many programming languages 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.

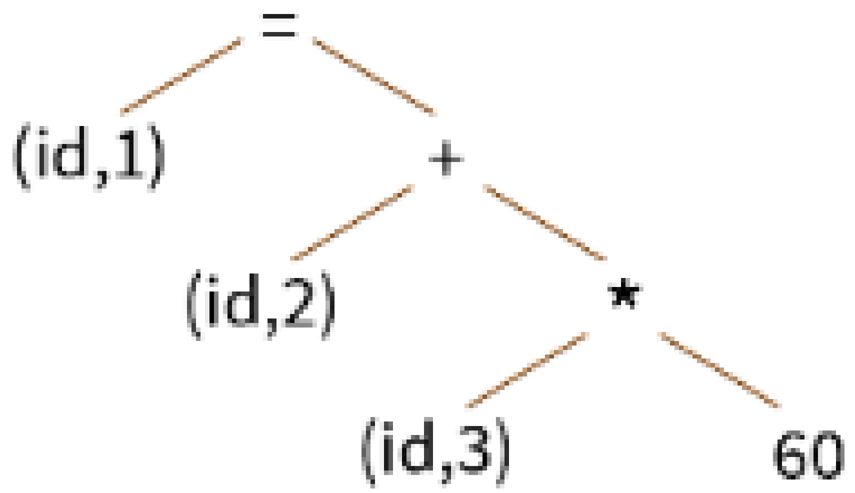
E.g. The semantic analyzers give an error in semantics when an integer and a string are added.

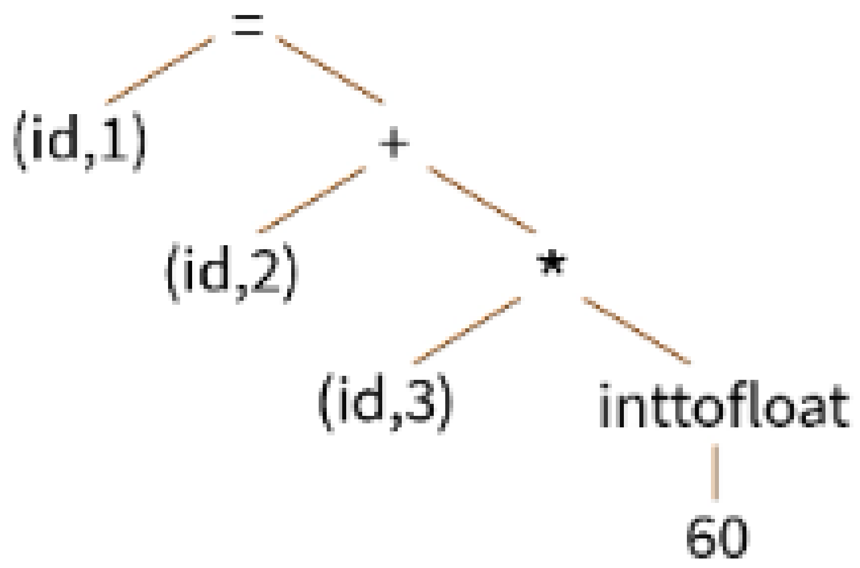
SEMANTIC ANALYSIS

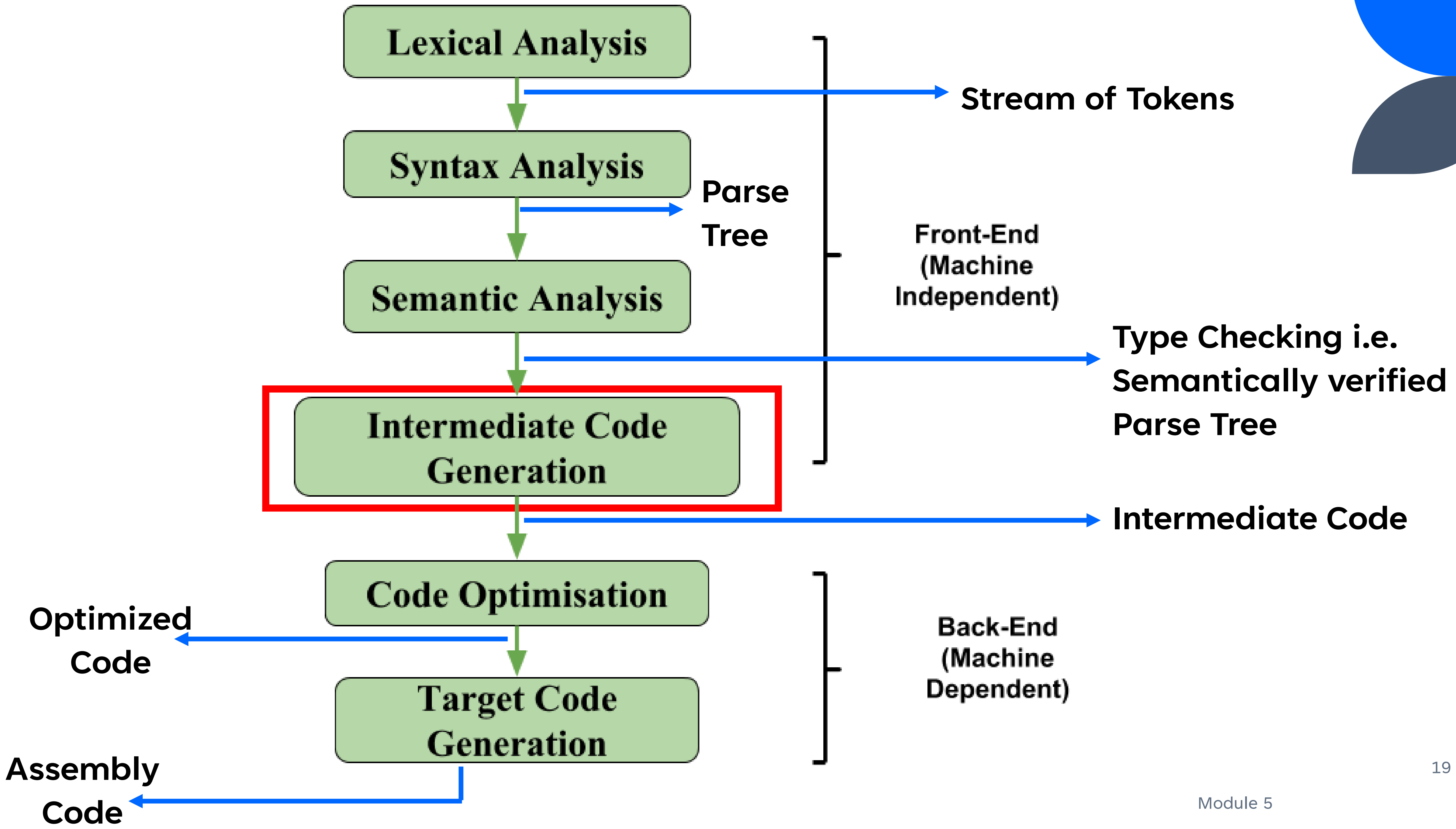
The **Language Specification** permits some type conversions called as **COERCIONS.**

For e.g. a binary operator may be applied to either a pair of integers or pair of floating-point numbers

If the operator is applied to a floating-point number and an integer, the compile coerce the integer into a floating-point number.






Lexical Analysis
Stream of Tokens
Syntax Analysis
Parse Tree
Front-End (Machine Independent)
Semantic Analysis
Type Checking i.e. Semantically verified Parse Tree
Intermediate Code Generation
Intermediate Code
Code Optimisation
Optimized Code
Back-End (Machine Dependent)
Target Code Generation
Assembly Code

Intermediate Code Generation

Intermediate code generation is the fourth phase in the process of compilation.

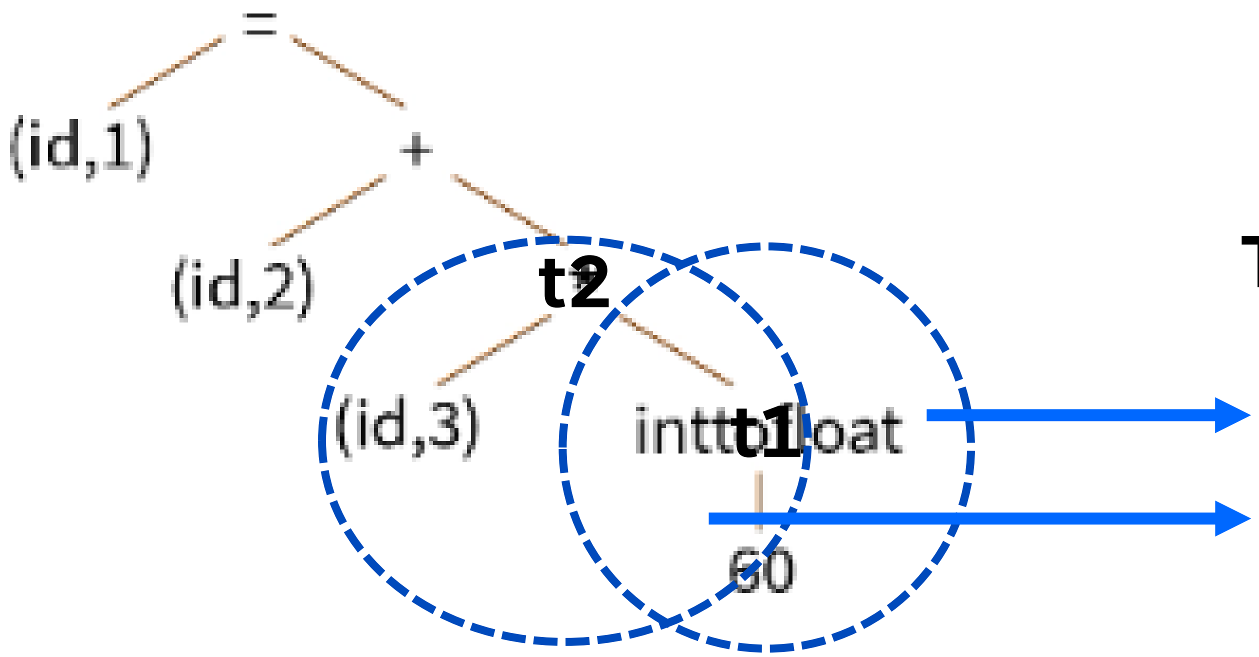
After Syntax and Semantic Analysis, compilers generate a machine-like Intermediate Code for the target machine.

It's a program for some kind of abstract machine.

Between high-level and machine-level languages is intermediate code. It is necessary to create this intermediate code in a way that makes it simple to convert it into the target machine code.

The Output of Intermediate Code Generator is **THREE ADDRESS CODE**.

EXAMPLE



Three Address Code – 3AC

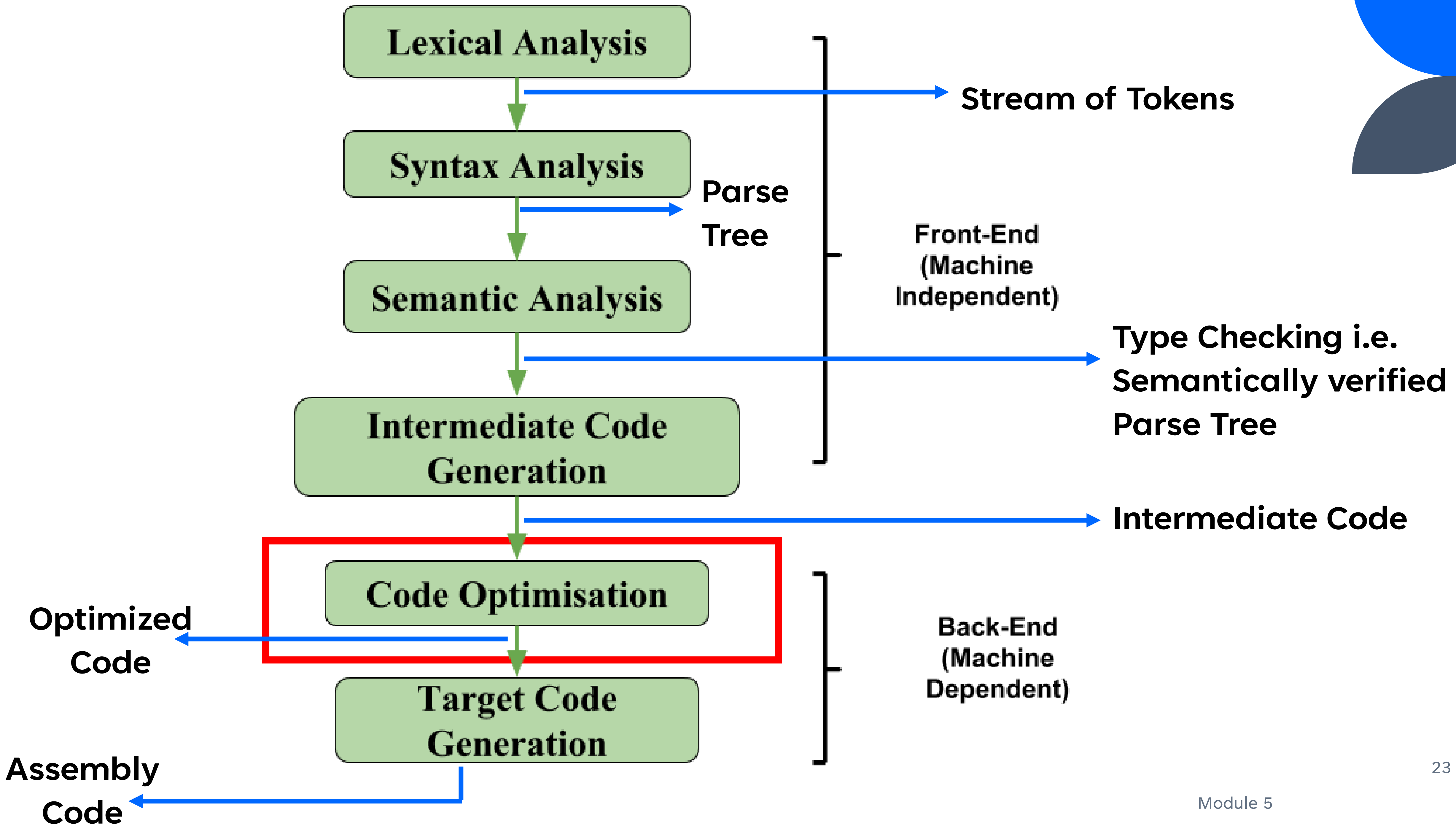
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Some points worth noting about 3AC

1. Each 3AC assignment instruction has at most one operator on RHS. Thus, these instructions fix the order in which operations are to be done.

2. The compiler must generate a temporary name to hold the value computed by three-address instruction

3. Some "three-address instructions" like the 1st and last instruction have fewer than three operands.


Lexical Analysis
Stream of Tokens
Syntax Analysis
Parse Tree
Semantic Analysis
Front-End (Machine Independent)
Type Checking i.e. Semantically verified Parse Tree
Intermediate Code Generation
Intermediate Code
Code Optimisation
Optimized Code
Back-End (Machine Dependent)
Target Code Generation
Assembly Code

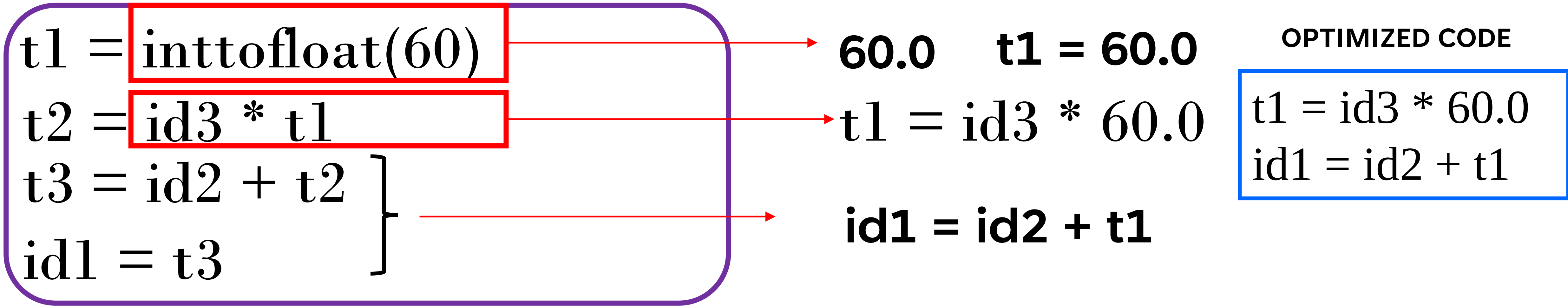
Code Optimization

SYNTHESIS PHASE

1. This is the fifth phase of the compiler and is machine-independent.

2. It attempts to improve the intermediate code so that better target code will result.

3. Optimization can be defined as removing unnecessary code lines and rearranging statement sequences to speed up programme execution without wasting resources.

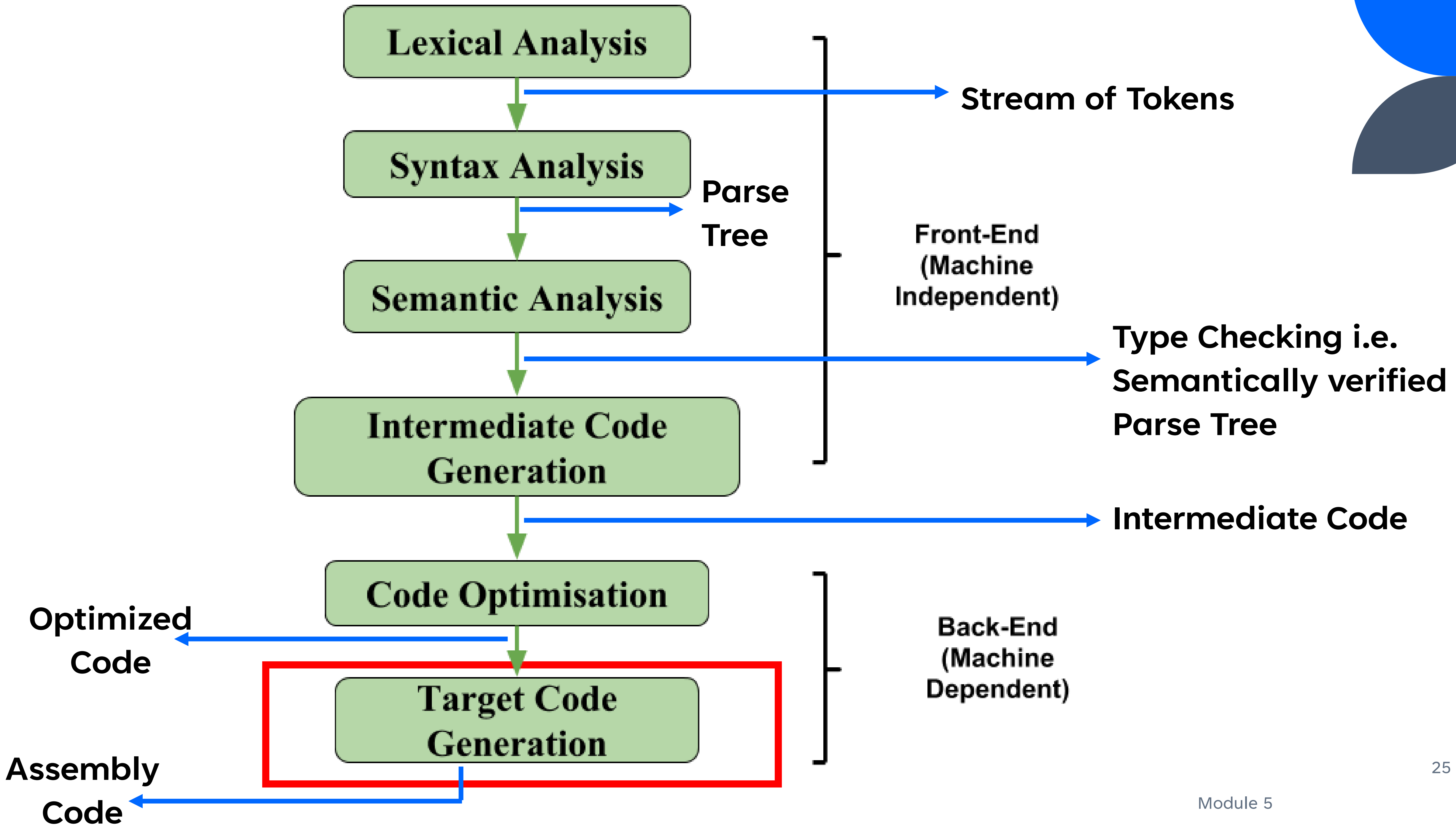


```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

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

OPTIMIZED CODE

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


Lexical Analysis
Stream of Tokens
Syntax Analysis
Parse Tree
Semantic Analysis
Front-End (Machine Independent)
Type Checking i.e. Semantically verified Parse Tree
Intermediate Code Generation
Intermediate Code
Code Optimisation
Optimized Code
Back-End (Machine Dependent)
Target Code Generation
Assembly Code

TARGET CODE GENERATOR

1. The code generator takes as input an intermediate representation of the source program and maps it into the target language.

2. The objective of this phase is to allocate storage and generate relocatable machine code.

3. It also allocates memory locations for the variable. The instructions in the intermediate code are converted into machine instructions.

4. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

TARGET CODE GENERATOR – EXAMPLE

OPTIMIZED CODE

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

EQUIVALENT

TARGET CODE

Loads the content of address id3 into register R2

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

60.0 has to be treated as Immediate Constant

The 1st operand of each instruction specifies a destination.

The F in each instruction tells us that it deals with floating-point numbers.

SYMBOL TABLE MANAGEMENT

1. 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.
2. These attributes provide information about;
   - ✓ The storage allocated
   - ✓ Its type
   - ✓ Its scope (i.e. where in the program its value may be used)
3. In case of procedure names, things such as;
   - ✓ Number and types of arguments
   - ✓ Method of passing each argument (Call by value or Call by Reference)
   - ✓ The type returned
4. The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name
5. The data structure should be designed to allow the compiler to find the record for each name quickly and to store and retrieve data from that record quickly

ERROR HANDLING ROUTINE

1. One of the most important functions of a compiler is the detection and reporting of errors in the source program.

2. The error message should allow the programmer to determine exactly where the errors have occurred.

3. Errors may occur in all or the phases of a compiler.

4. Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message.

ERROR HANDLING ROUTINE

In the compiler design process error may occur in all the below-given phases:

a) **Lexical analyzer:** Wrongly spelled tokens

b) **Syntax analyzer:** Missing parenthesis

c) **Intermediate code generator:** Mismatched operands for an operator

d) **Code Optimizer:** When the statement is not reachable

e) **Code Generator:** When the memory is full or proper registers are not allocated

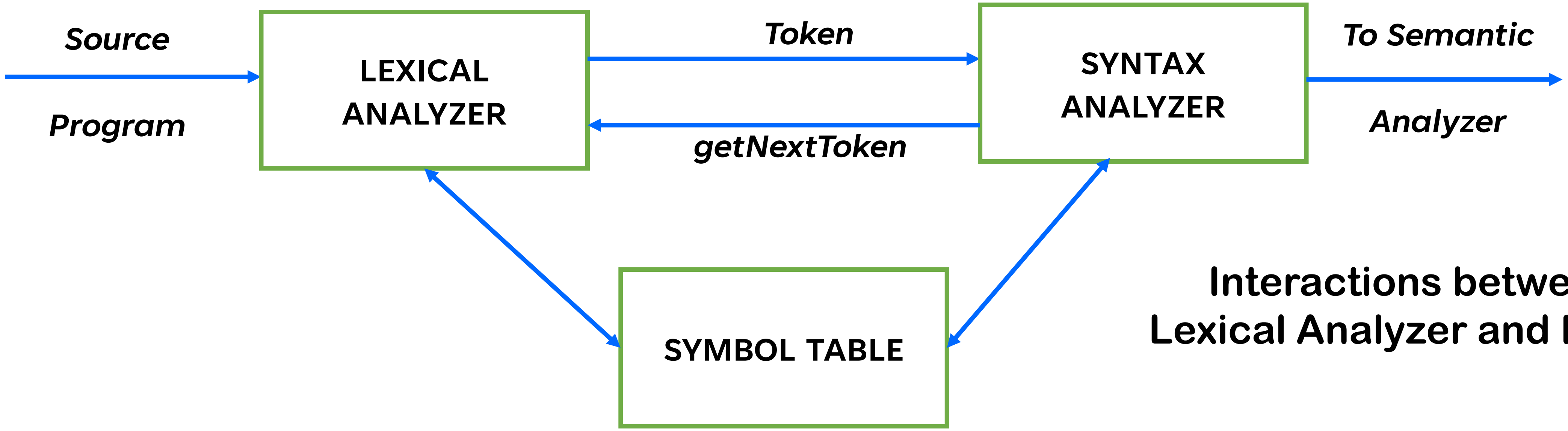
f) **Symbol tables:** Error of multiple declared identifiers

The most frequent mistakes are incorrect token sequences in type, invalid character sequences in scanning, scope errors, and parsing errors in semantic analysis.

LEXICAL ANALYSIS

ROLE OF LEXICAL ANALYZER

1. The main task of Lexical Analyzer is to read input characters of the source program, group them into lexemes and produce output a sequence of tokens for each lexeme.
2. When Lexical analyzer discovers a lexeme constituting an identifier, it inserts the same into symbol table.



Interactions between Lexical Analyzer and Parser

3. Besides Identification of Lexemes, Lexical Analyzer performs the following tasks;
   a) Stripping out comments and whitespaces (Blank, New line etc)
   b) Correlating error messages with source program

LEXICAL ANALYSIS

Tokens, Patterns, Lexemes

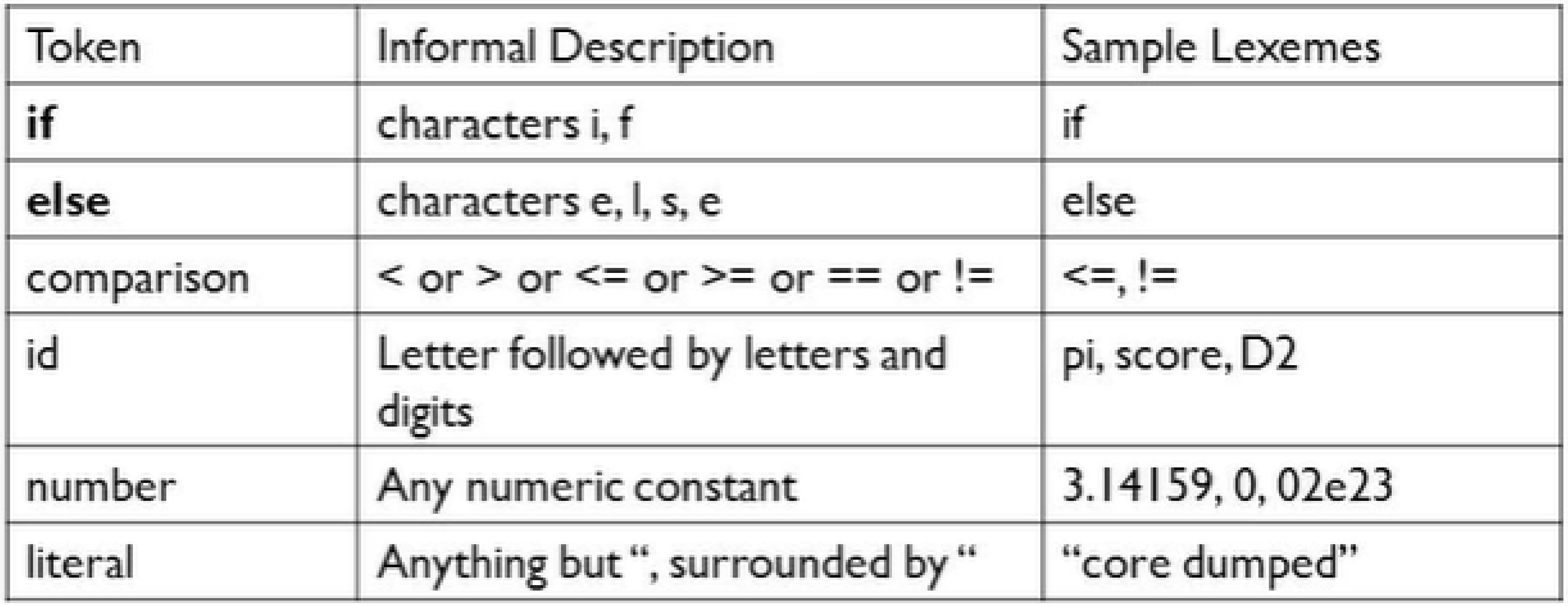
1. Token – Is a pair consisting of token name and an optional attribute value. Token names are abstract symbol representing a kind of lexical unit. For e.g. A particular keyword, or a sequence of input characters denoting an Identifier

2. Pattern–It specifies a set of rules that a scanner follows to create a token. It is a description of the form that lexemes of a token may take.

   For e.g. In the case of KEYWORD as a TOKEN, "The pattern is just the sequence of characters that form a keyword".

3. Lexeme – Is a sequence of characters in the source program that matches the pattern for a token and is identified as an instance of that token.

EXAMPLE OF TOKENS



Token	Informal Description	Sample Lexemes
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letters and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 02e23
literal	Anything but “, surrounded by “	“core dumped”

Divide the following C + + program into appropriate lexemes

```
float limitedSquare(x) float x {
    /* returns x-squared, but never more than 100 */
return (x<=-10.0 || x>=10.0)?100:x*x;     }
```

LEXICAL ANALYSIS

As a result, **a two-buffer system** is used to safely manage huge lookaheads.

- Scanner is the only part of the compiler which reads a complete program.
- Lexical Analyzer is the only phase of the compiler that reads the source program character by character. This phase consumes considerable amount of time, due to which, compilation time goes low.
- Hence, speed of Lexical Analysis is major concern!!
- Thus, some ways must be examined to **SPEED UP** the most simple but important task of **reading the source program**.

This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme

LEXICAL ANALYSIS

A buffer contains data that is stored for a short amount of time, typically in computer's memory (RAM). The purpose of buffer is to hold the data right before it is used

INPUT BUFFERING

1. Input buffering is the process of storing all of the source code in memory before beginning to compile it.

2. By having all of the data readily available, compilers can read and execute instructions more quickly, resulting in overall improved program performance and faster build times.

3. There are two methods used in this context: **One Buffer Scheme, and Two Buffer**

ONE BUFFER SCHEME : In this scheme, only one buffer is used to store the input string

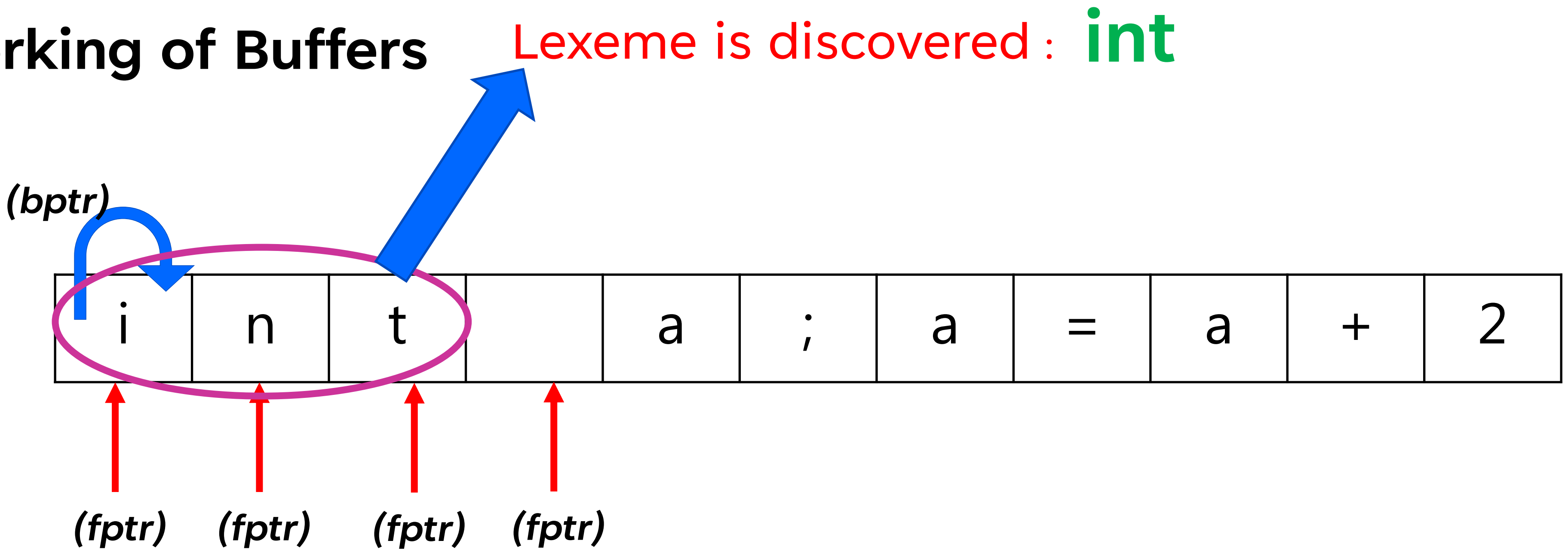
Size of the Buffer = N Characters;

Where N = Number of Character on one Disk block (1024 or 4096)

DISADVANTAGE:- The problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.

LEXICAL ANALYSIS

Working of Buffers



Initially both the pointers point to the first character of the input string as shown above

lexemeBegin (bptr) - points to the beginning of the current lexeme which is yet to be found.
forward (fptr) - moves ahead until a match for a pattern is found.

When a lexeme is discovered, lexeme begin is set to the character immediately after the newly discovered lexeme, and forward is set to the character at the right end of the lexeme.

LEXICAL ANALYSIS

ONE BUFFER SCHEME

DISADVANTAGE:- The problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.

To overcome the drawback of One Buffer Scheme **Two Buffer Scheme(Buffer Pairs)** **is introduced.**

Buffer Pairs

1. It consists of two buffers, each of which has an N-character size and is alternately reloaded.
2. The first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled.
3. Each buffer is of the same size N, and N is usually the size of a disk bock (e.g. 4096 bytes).
4. If (Number of characters in the Input File < N) then a **SPECIAL CHARACTER like** ***eof*** marks the end of source file

i	n	t		a	;	a	=	a	+	2	*eof*			

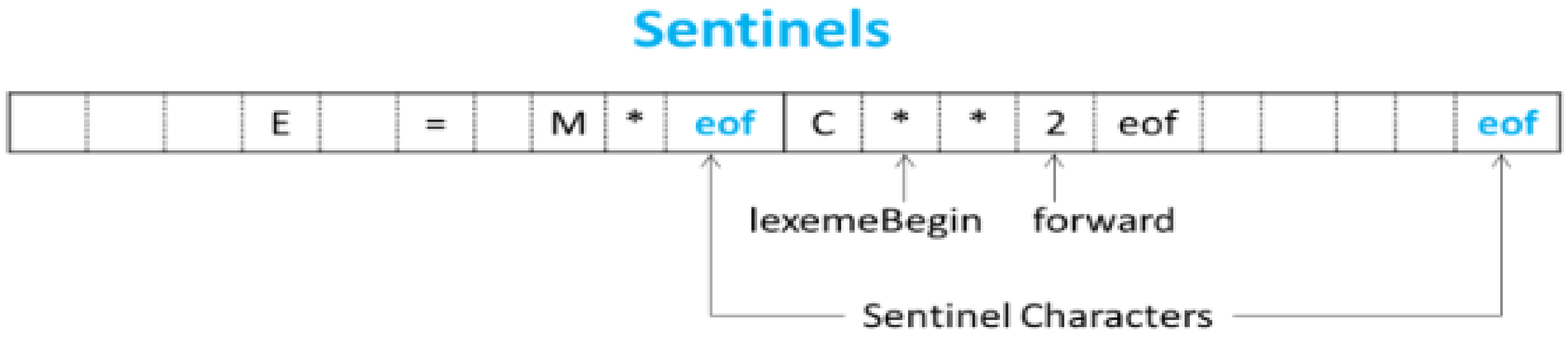
SENTINELS

1. Sentinels play a vital role in managing buffer pairs effectively.
2. A sentinel is a special character, often used at the end of the input buffer, to mark its boundaries.
3. In Two Buffer Technique, we must check, each time we advance forward pointer ***(fptr),*** that we have not moved off one of the buffers; if we do, then we must also reload the other buffer.
4. Thus, for each character read, we make two tests:

    Test 1: For end of buffer.

    Test 2: To determine what character is read.

Thus, The sentinel is a special character that cannot be part of the source program



Specification & Recognition of tokens

For specifying Lexeme Pattern – Regular Expressions are used.

While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for Tokens.

REGULAR EXPRESSIONS

Let Σ = {a, b}

1. a | b = {a, b}
2. (a | b) (a | b) = {aa ,ab, ba, bb}
3. a* = Can consist of strings having zero's or more a's i.e. $\{\epsilon$, a, aa, aaa, aaaa}
4. (a | b)* = $\{\epsilon$, a, b, aa, ab, ba, bb, aaa, ...}

Following Patterns are used for identifying tokens

digit → [0 – 9]

digits → digit$^+$

number → digits (.digits)? (E[+ -]? digits)?

letter → [A – Za – z]

id → letter (letter | digit)*

if → if

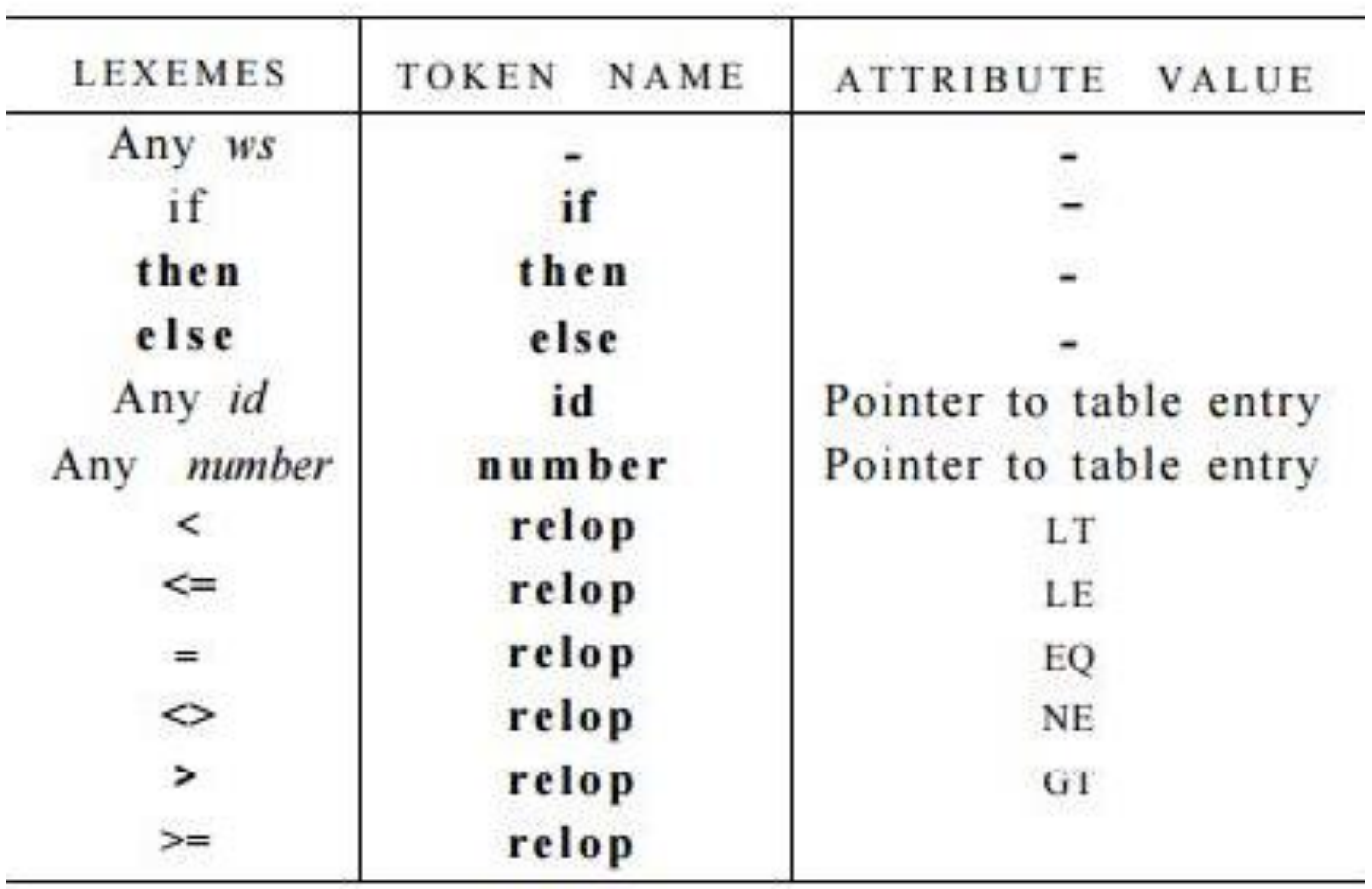
then → then

relop → < | > | <= | >= | = |< >

ws → (blank|tab|newline)+

Tokens, their patterns and attribute values

Goal for the lexical analyzer is summarized in the table below.

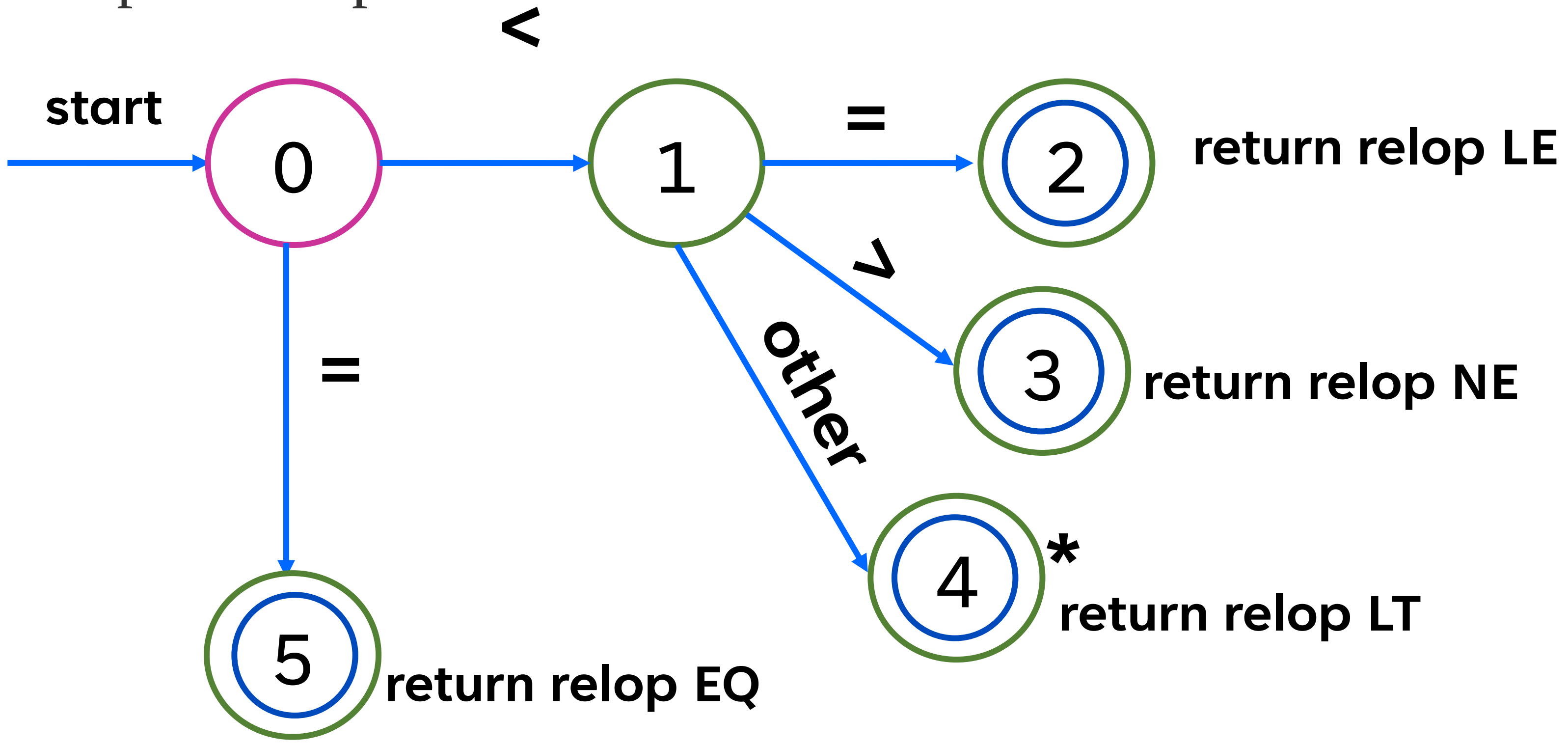


LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	-	-
if	**if**	-
then	**then**	-
else	**else**	-
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	

The table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value

TRANSITION DIGRAM

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams." Thus Regular-Expression patterns are converted into TRANSITION DIAGRAM



TRANSITON DIAGRAN FOR *RELOP*

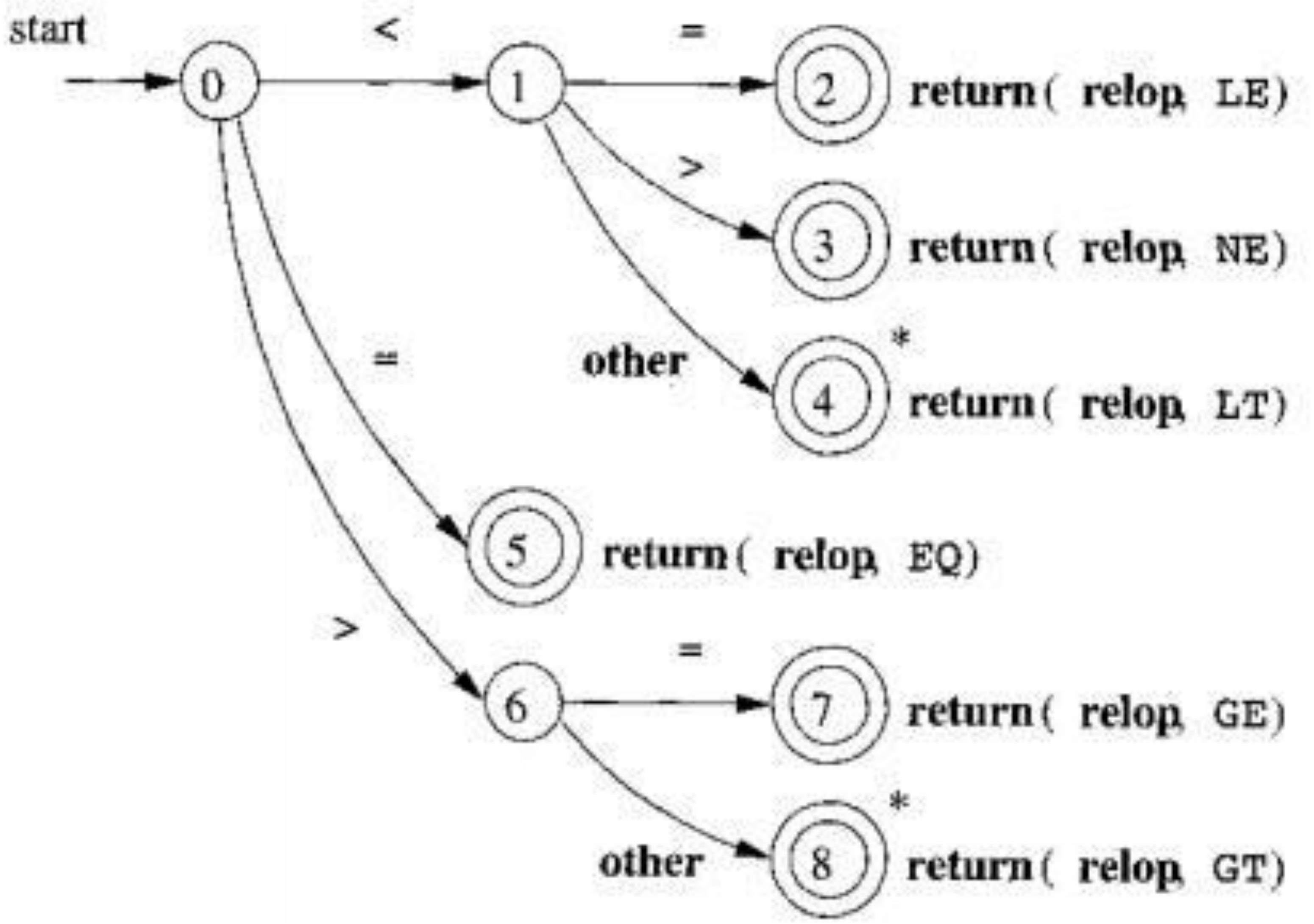


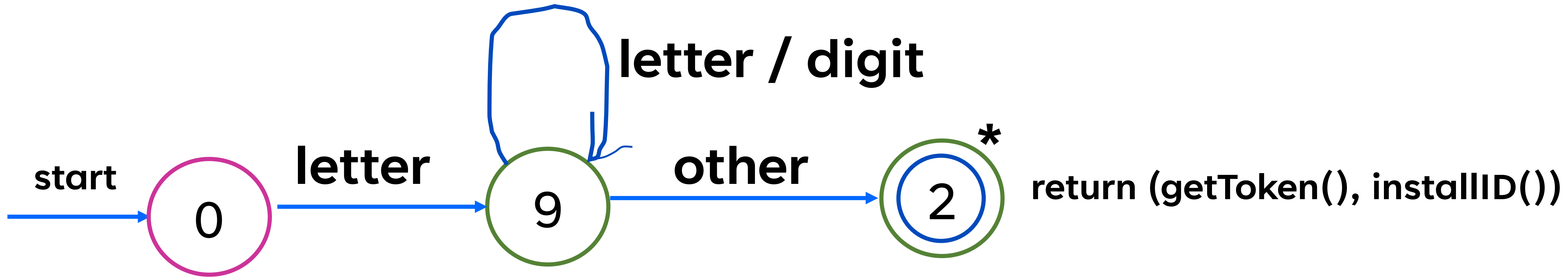
Figure 3.13: Transition diagram for **relop**

TRANSITON DIAGRAN FOR *Identifier*

Regular Expression for Identifier

id $\longrightarrow$ letter (letter | digit)*

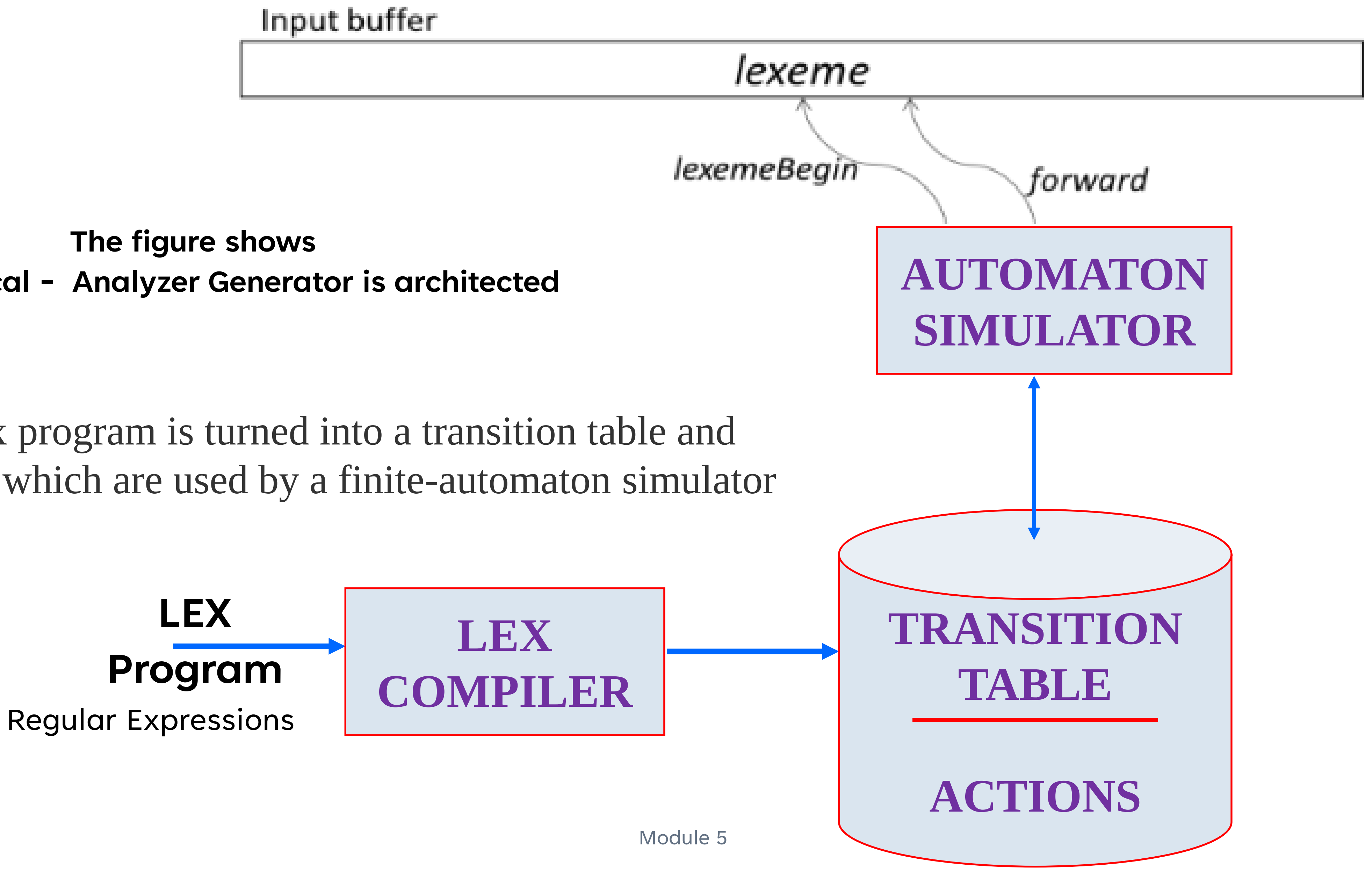
digit $\rightarrow$ [0 – 9]
letter $\rightarrow$ [A – Za – z]



DESIGN OF LEXICAL ANALYZER GENERATOR

The figure shows
How a Lexical - Analyzer Generator is architected

A Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator



DESIGN OF LEXICAL ANALYZER GENERATOR

Following are the components that are created from the Lex program by Lex itself.

1. A transition table for the automaton.
2. Those functions that are passed directly through Lex to the output
3. The actions from the input program, which appear as fragments of code to be invoked at the appropriate time by the automaton simulator.

To construct the automaton, take each regular expression pattern in the Lex program and convert it to an NFA

We need a single automaton that will recognize lexemes matching any of the patterns in the program, so we combine all the NFA's into one by introducing a new start state with e-transitions to each of the start states of the NFA's

Pattern Matching Based on NFA's

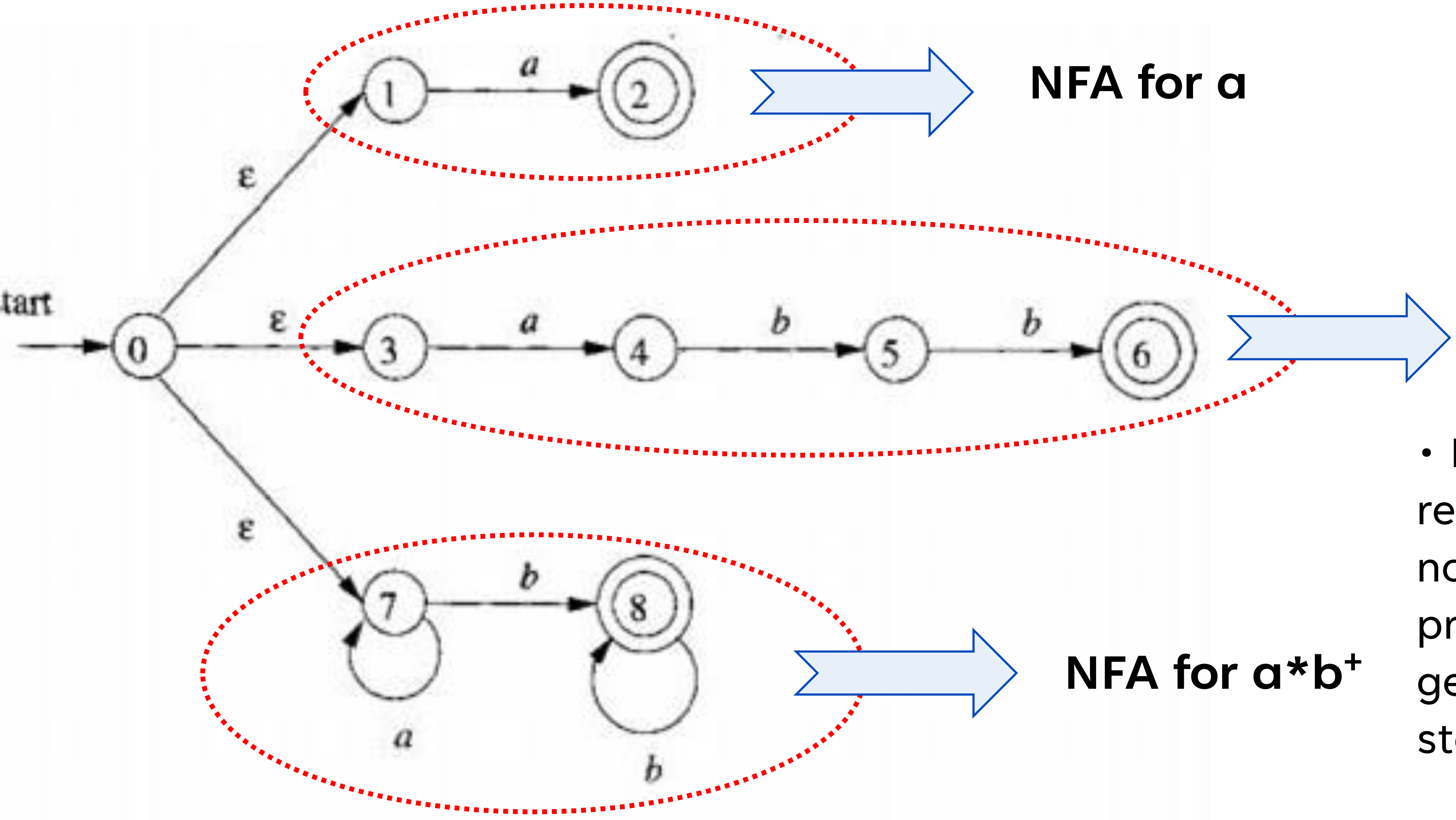


Figure 3.52: Combined NFA

- Eventually, NFA simulation reaches a point where there are no next states i.e. no longer prefix of the input would ever get the NFA to an accepting state.

- Thus, We can say that a longer prefix is a lexeme matching some pattern

SYNTAX ANALYSIS

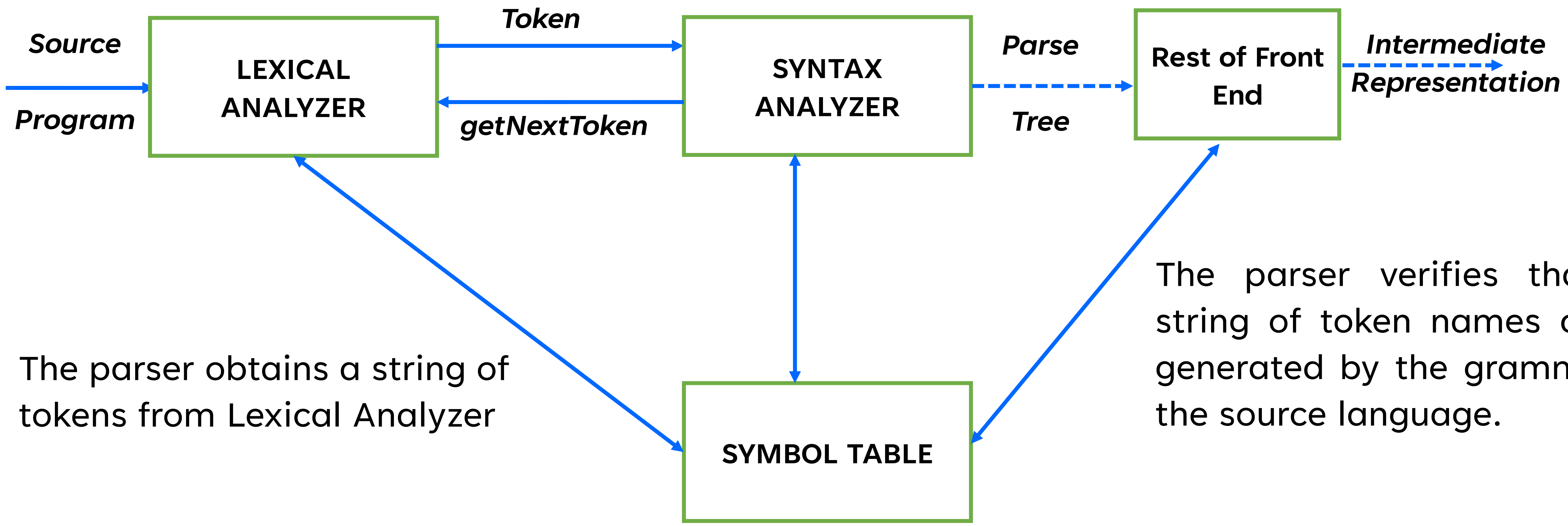
- Role of Context Free Grammar in Syntax analysis

- **Types of Parsers:**

    a. Top down parser- LL(1)

    b. Bottom up parser- SR Parser,

    c. Operator precedence parser, SLR.



The parser obtains a string of tokens from Lexical Analyzer

The parser verifies that the string of token names can be generated by the grammar for the source language.

POSITION OF PARSER IN COMPILER MODEL

The parser constructs a PARSE TREE for conceptually well-formed programs and passes it to the rest of the compiler for further processing

The parse tree is formed by matching the input string with the pre-defined grammar. If the parsing is successful, the given string can be formed by the grammar, else an error is reported.

Importance of Syntax Analysis

1. It is used to check if the code is grammatically correct or not.
2. It helps us to detect all types of syntax errors.
3. It gives an exact description of the error.
4. It rejects invalid code before actual compiling.
5. Helps you to make sure that each opening brace has a corresponding closing balance
6. Each declaration has a type and that the type must be exists

Parse Tree

Parse tree is a graphical representation of the derivation. It is used to see how the given string is derived from the start symbol. The start symbol is the root of the parse tree, and the characters of the input string become the leaves.

PARSING TECHNIQUES

The parsing techniques can be divided into two types:

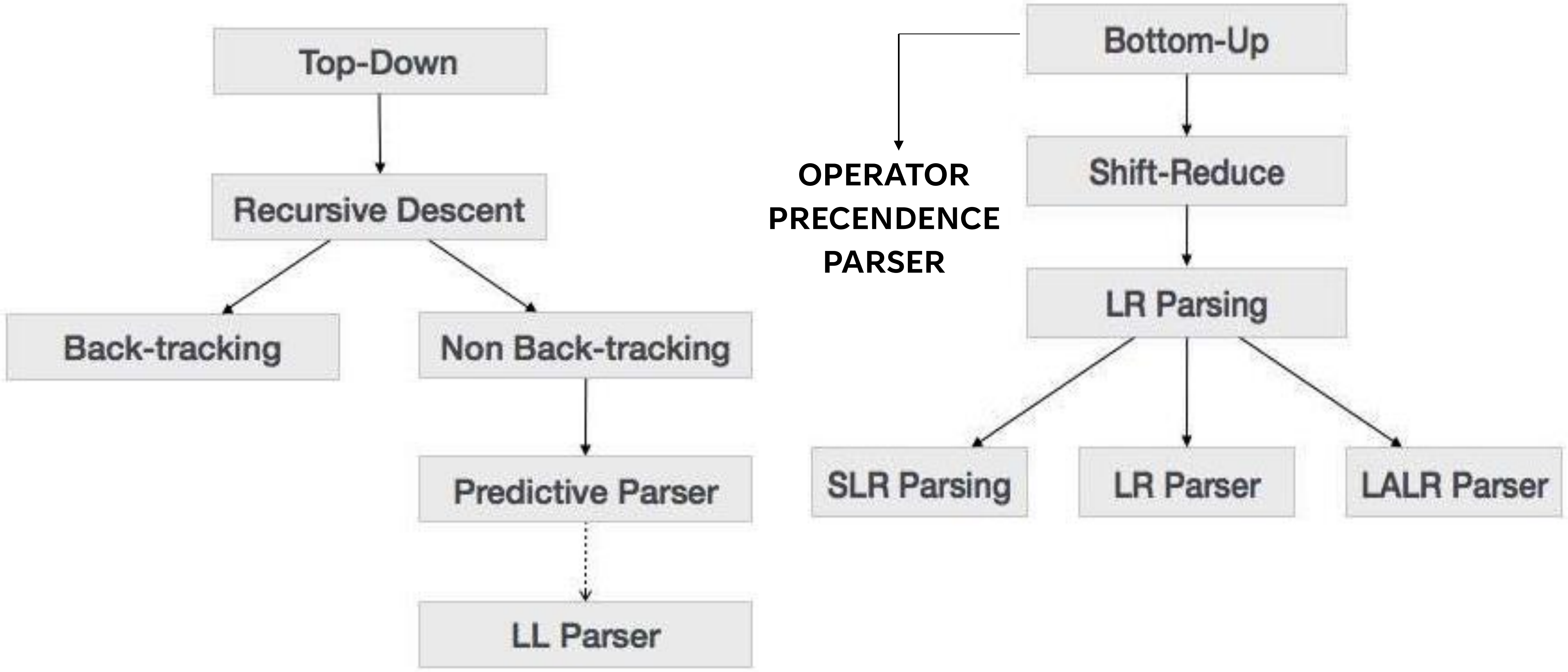
Top-down parsing: The parse tree is constructed from the root to the leaves in top-down parsing. Some most common top-down parsers are:-

Recursive Descent Parser and LL parser (a.k.a. Predictive Parser)

Bottom-up parsing: The parse tree is constructed from the leaves to the tree’s root in bottom-up parsing. Some examples of bottom-up parsers are:-

The LR parser, SLR parser, CLR parser, etc.

TYPES OF PARSER



TOP DOWN PARSING

1. Top-down parsing is a parsing-method where a sentence is parsed starting from the root of the parse tree (with the “Start” symbol), working recursively down to the leaves of the tree (with the terminals).

2. In practice, top-down parsing algorithms are easier to understand than bottom-up algorithms.

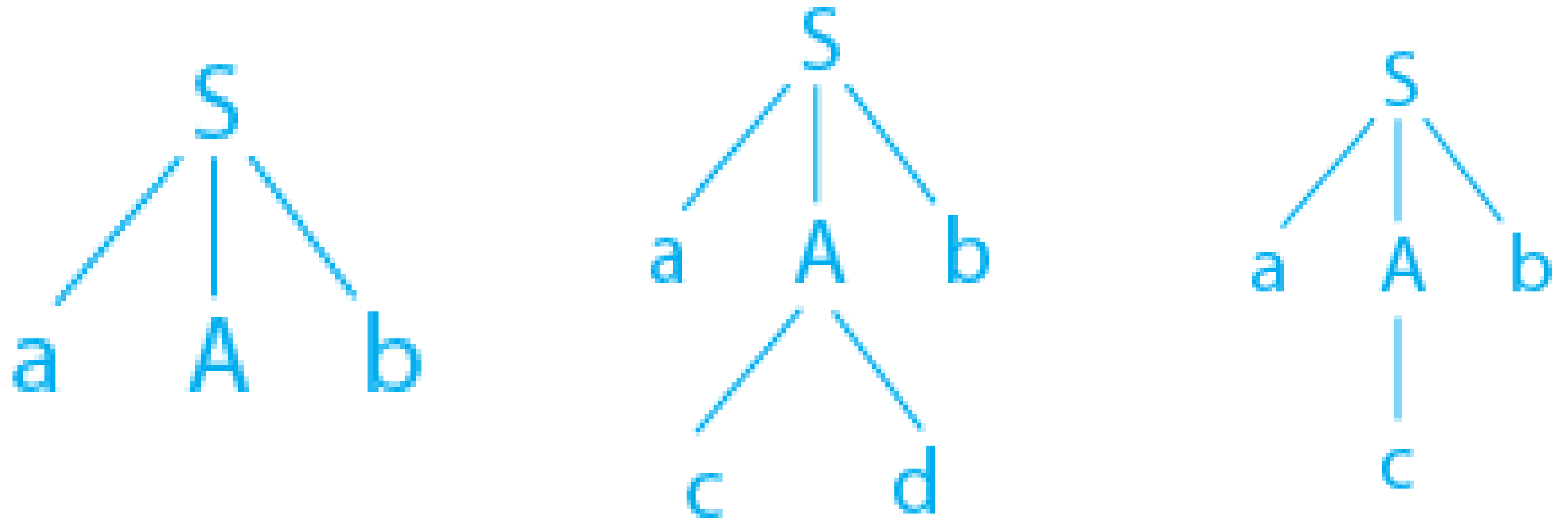
3. Not all grammars can be parsed top-down, but most context-free grammars can be parsed bottom-up.

TOP DOWN PARSING

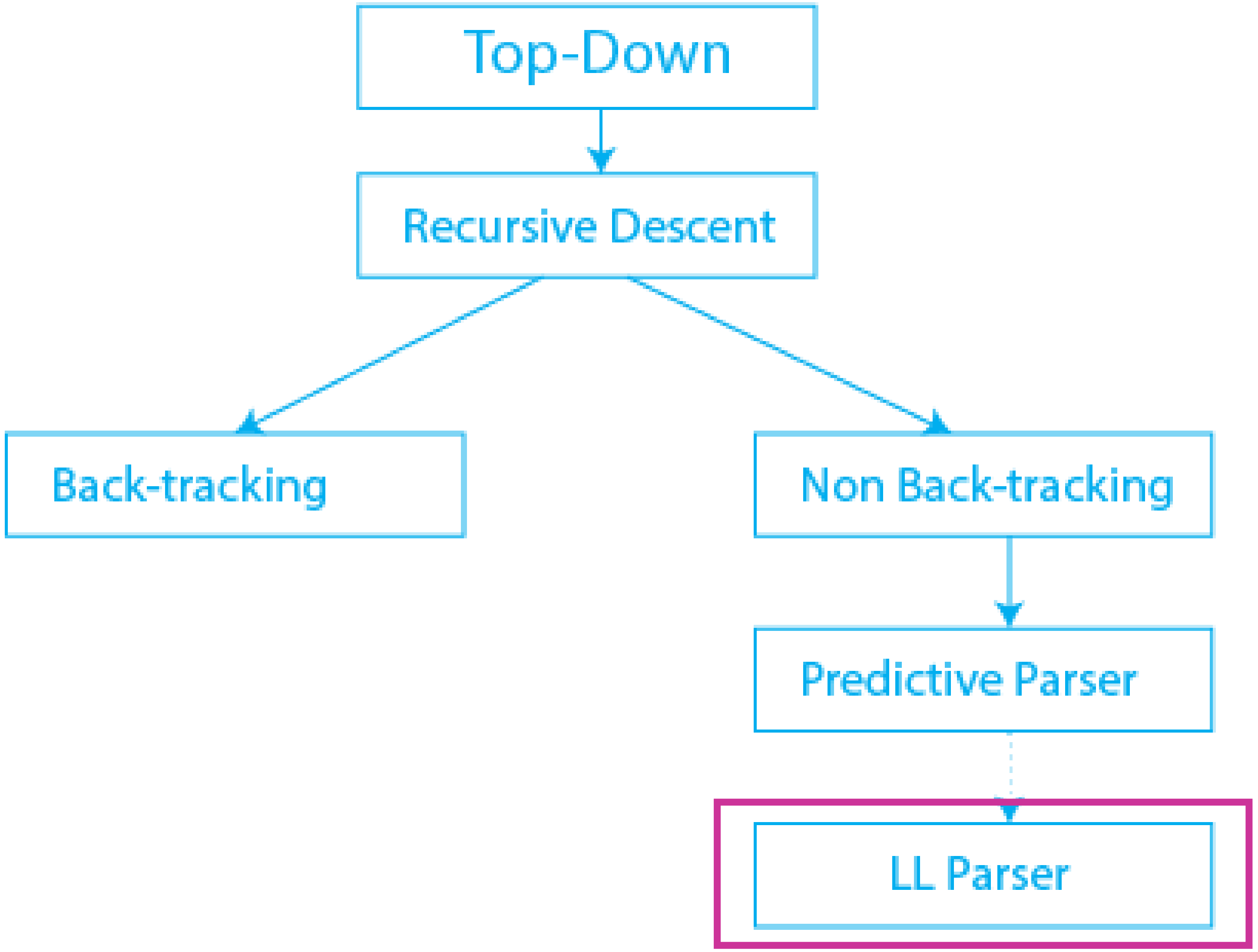
Consider the lexical analyzer's input **string 'acb'** for the following grammar by using left most deviation.

S->aAb

A->cd | c

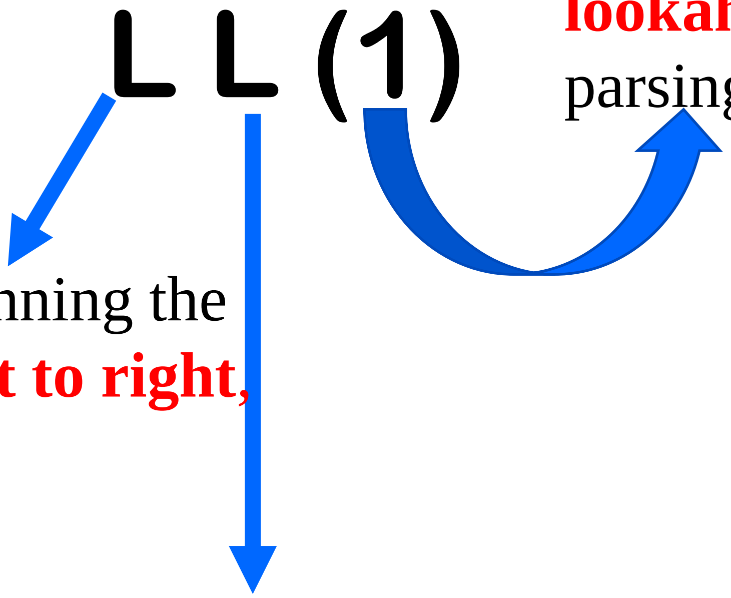


Classification of TOP DOWN PARSER



LL(1) Parser

LL(1) parsing is a top-down parsing method in the syntax analysis phase of compiler design.



stands for using **one input symbol of lookahead** at each step to make parsing action decision.

stands for scanning the input from **left to right,**

stands for producing a **leftmost derivation**

Prime Requirement of LL(1)

The following are the prime requirements for the LL(1) parser:

- The grammar must have no left factoring and no left recursion.
- FIRST() & FOLLOW()
- Parsing Table
- Stack Implementation
- Parse Tree

NUMERICALS On LL (1) is provided in Hand Written PDF document

UNIVERSITY QUESTIONS

- Explain the phases of compiler. Discuss the action taken in various phases to compile the statement:

  a = b * c + 10   where a, b, c are of real type

  a = b + c – d * 5

  a = a * b – 5 * 3 / c   where c = 1

- Test whether following grammar is LL(1) or not. If it is LL(1), construct Parsing table for the same

  S – 1AB | ε

  A - 1AC | 0C

  B – 0S

  C - 1

- Compare Pattern, Lexeme and Token with example

UNIVERSITY QUESTIONS

- Test whether following grammar is LL(1) or not. If it is LL(1), construct Parsing table for the same

  S – AB | gDa

  A – ab | c

  B – dC

  C – gC | g

  D – fD | g

- Find FIRST and FOLLOW for the following grammar

  S – Bb | Dd

  B – aB | ε

  D – cD | ε

- Eliminate Left Recursion for the following Grammar

  S – (L) | x ; L – L, S| S

UNIVERSITY QUESTIONS

- For following grammar construct LL(1) Parsing table and parse the string (a-a)

  S – F

  S – (S-F)

  F – a

- Compare Bottom-Up and Top-Down Parser
- Design LL(1) parsing table for the given grammar. Also state that whether the given grammar is LL(1) or not

  S – Ad

  A – aB | BC

  B – b

  C –e | ε

UNIVERSITY QUESTIONS

- Consider the following Grammar

  S – (A) | 0

  A – SB

  B - ,SB | ε
- Compare Bottom-Up and Top-Down Parsing Techniques
- Explain the role of Finite Automata in compiler theory
- Construct predictive parsing table for the grammar

  E → TE'

  E' → +TE' | ε

  T → FT'

  T' → *FT' | ε

  F → (E) | id

UNIVERSITY QUESTIONS

- What is Left Factoring? Find FIRST & FOLLOW for the following Grammar

  S – Aa

  A – BD

  B – b | ε

  D – d | ε

Thank you

Prepared By:

- Ms. Ankita Karia

Asst. Prof

CMPN Dept.

Email id: ankitakaria@sfit.ac.in