جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

قسم الامن السيبراني

Department of Cyber Security

Subject:

Compiler Design

Class:

Third

Lecturer:

Asst. Prof. Dr. Ali Kadhum Al-Quraby

Lecture: (2)

The Compiler Phases

❖ LEXICAL ANALYSIS

The first phase of a compiler is called ***lexical analysis*** or ***Scanning*** or ***Lexer***. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called ***lexemes***. From each lexeme, the lexical analyzer produces as output a token of the form

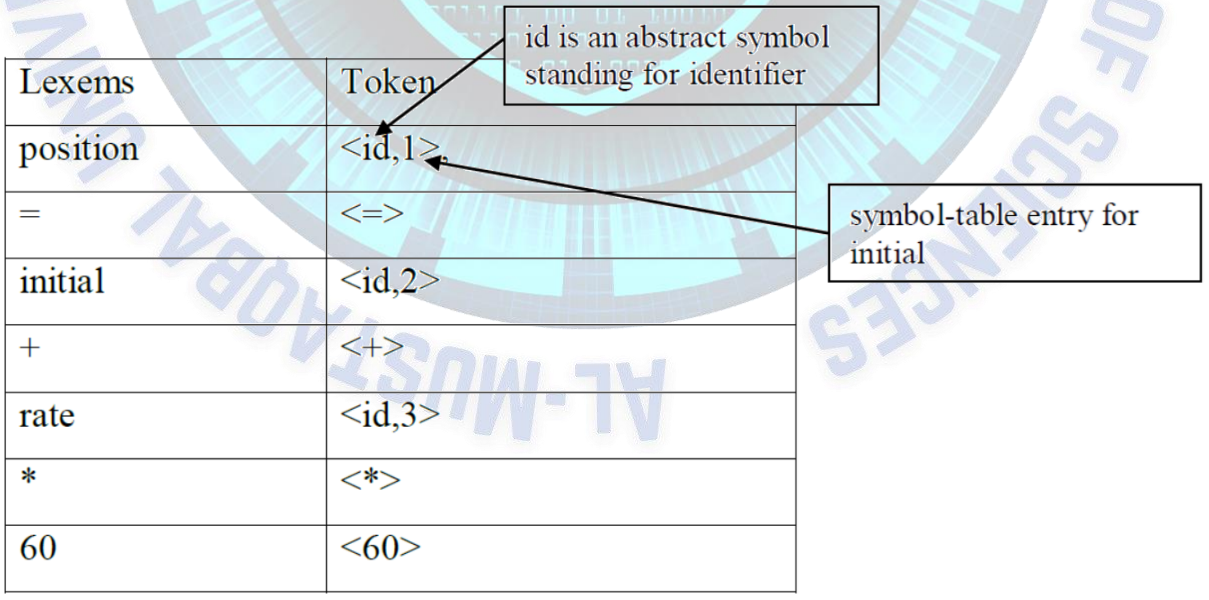
<token – name, attribute – value>

that it passes onto the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component ***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.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:



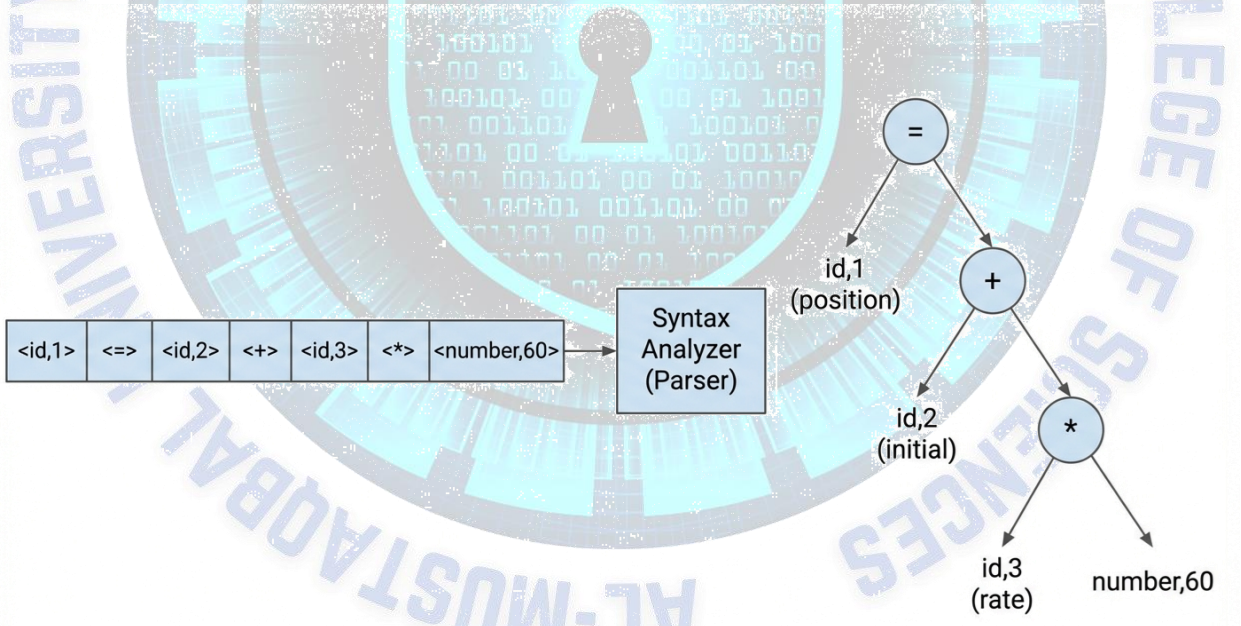
Lexems	Token
position	<id,1>
=	<=>
initial	<id,2>
+	<+>
rate	<id,3>
*	<*>
60	<60>

Note: - Blanks separating the lexemes would be discarded by the lexical analyzer.

❖ Syntax Analysis

The syntax analyzer groups the tokens together into syntactic structures. This phase is called ***syntax analysis*** or ***parsing***. The parser uses the first 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.

The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.



❖ Semantic Analysis

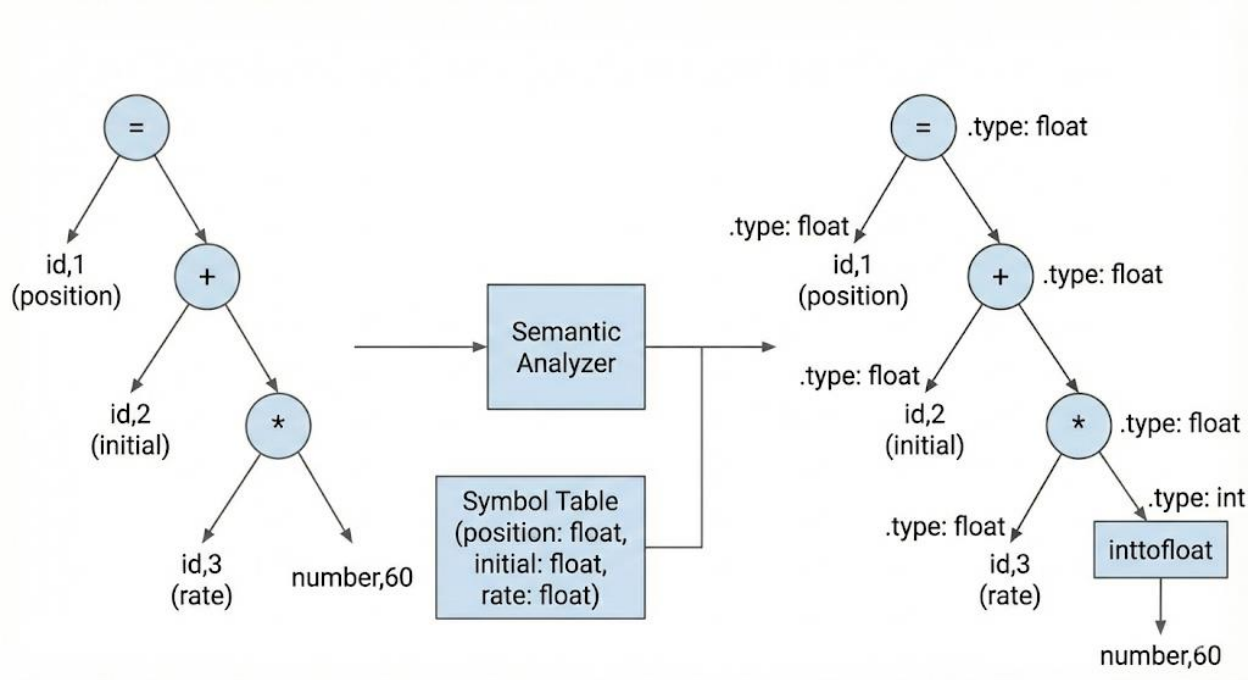
The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is ***type checking***, where the compiler checks that each operator has matching operands.

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.

The language specification may permit some type conversions called ***coercions***.

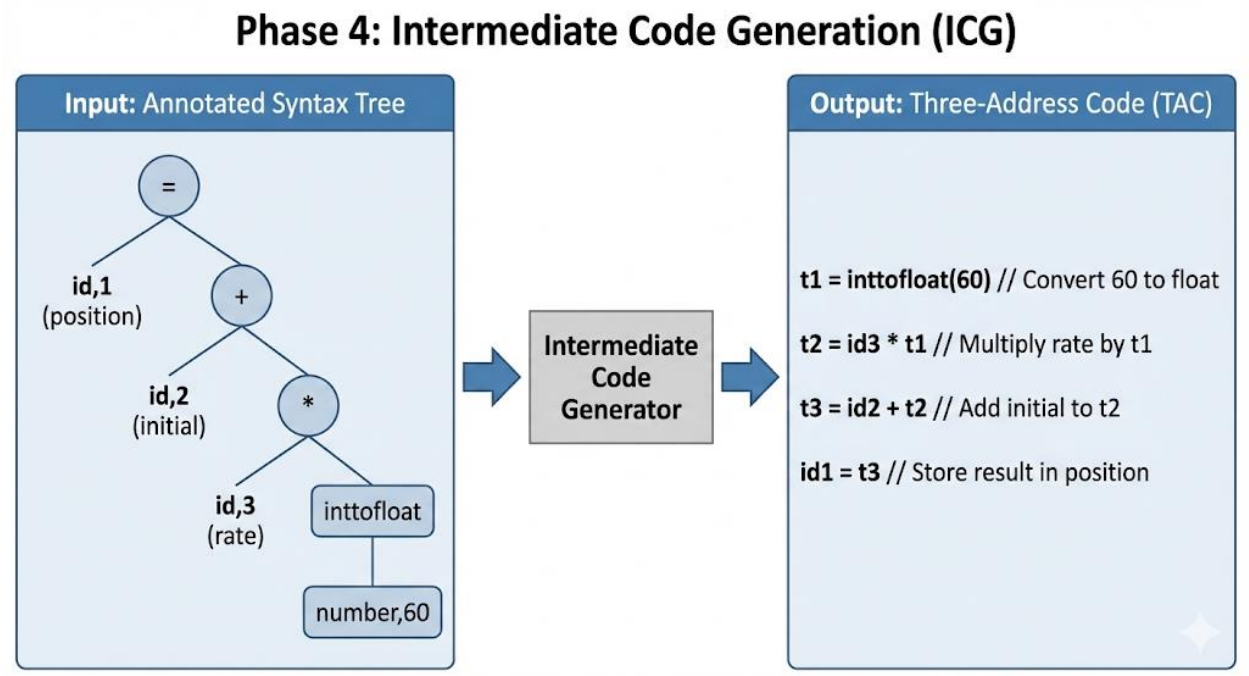
For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.



❖ Intermediate Code Generation

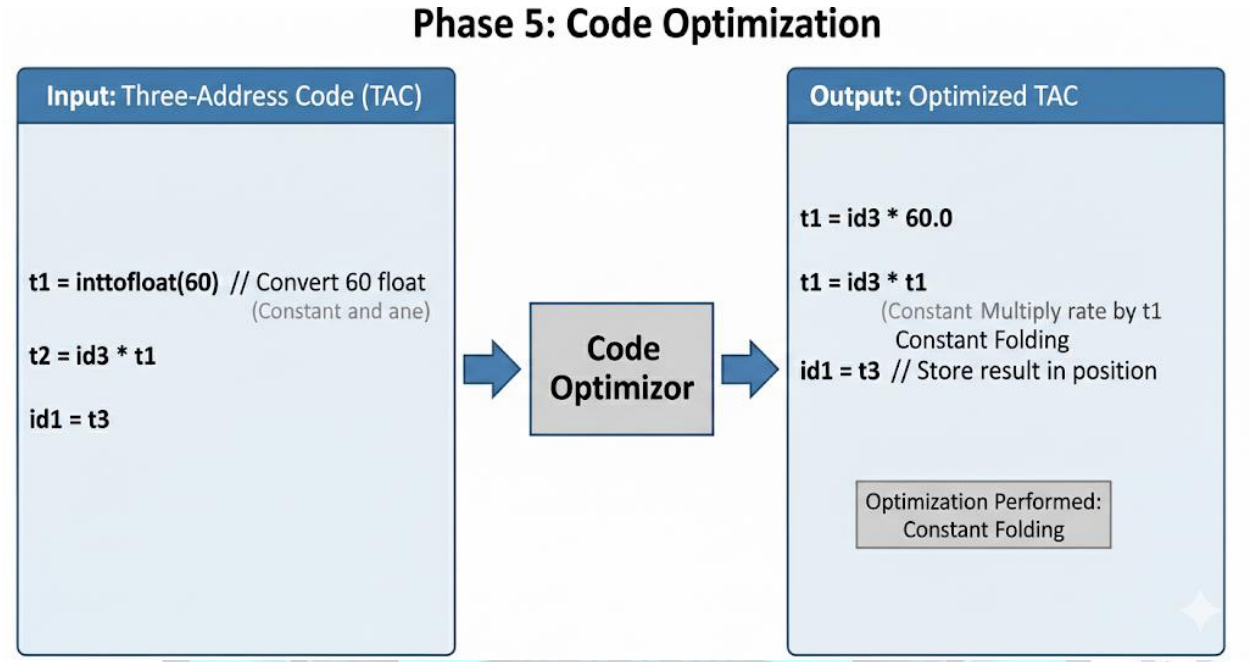
In this process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

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 easy to produce and it should be easy to translate into the target machine.



❖ CODE OPTIMIZATION

This is optional phase designed to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code Figure using an instruction for each operator in the tree representation that comes from the semantic analyzer. A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code.



❖ Code Generation

The code generator takes as input an intermediate representation of the source 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. A crucial aspect of code generation is the judicious assignment of registers to hold variables. Designing a code generator that produces truly efficient object programs is one of the most difficult parts of compiler design.

Target Code (Example in Assembly):

Code snippet

```
LDF R2, id3       // Load rate into register R2
MULF R2, R2, #60.0 // Multiply R2 by 60.0
LDF R1, id2       // Load initial into register R1
ADDF R1, R1, R2   // Add R2 to R1
STF id1, R1       // Store result in position
```