Compiler Construction 6001310-3

Abdelrahman Osman

aamosman@uqu.edu.sa

Textbook

- Compilers: Principles, Techniques, and Tools ,A. V. Aho, R. Sethi, J. D. Ullman; (c) 2010

Course information

- Lexical analysis, including regular languages and finite state acceptors;
- Syntactic analysis, including parsing techniques and grammars;
- Code generation
- Optimization.

- Prerequisite: 6001231-4 Programming Languages

Goals

- Understanding of the organization of a compiler
- Understanding of the concepts of scanning, parsing, and translation
- Understanding of Compiler writing tools

ABET

- (C): An ability to design, implement and evaluate a computer-based system, process, component or program to meet desired; Students are required design and implement a software project to meet a specification.
- (D): An ability to function effectively on teams to accomplish a common goal Projects are implemented in teams.
- (I): An ability to use the current techniques, skills, and tools necessary for computing practice.; Projects use current computing and modeling/design tools.

Topics To be Covered

- Introduction to compilers structure & goals
- Arithmetic expression processing using a stack
- Simple compiler structure
- Grammar, parse tree, and ambiguous grammar
- Translation schemes
- Context-free grammar & parsing
- Introduction to left recursion and right recursion
- Lexical analyzer (language, errors, pattern specifications)
- Operations on languages and regular expressions
- Finite automata
- Parsers and errors and sentential error
- Left recursion and left factoring
- FIRST, FOLLOW, and transition diagrams

Lecture 1

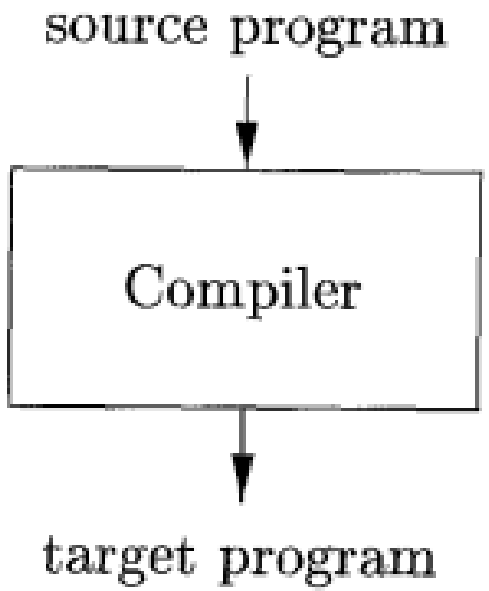
Introduction

- Programming languages are notations for describing computations to people and to machines.
- The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language.
- Before a program can be run, it first must be translated into a form in which it can be executed by a computer.

- The software systems that do this translation are called compilers.

Compiler

- 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

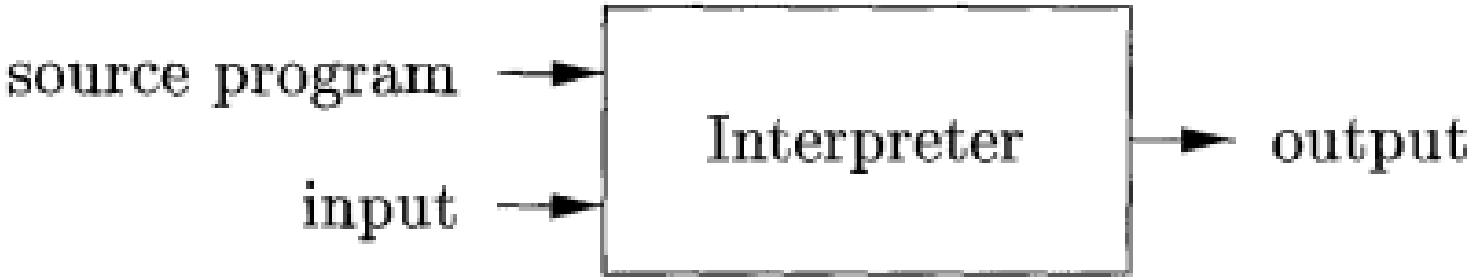


Running the target program

- If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

Interpreter

- An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

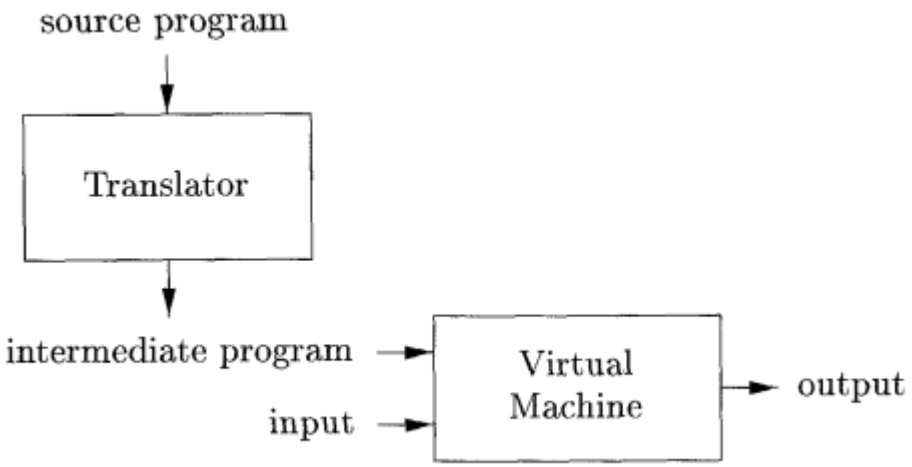


Difference between compiler and Interpreter

- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

A hybrid compiler

- Java language processors combine compilation and interpretation.



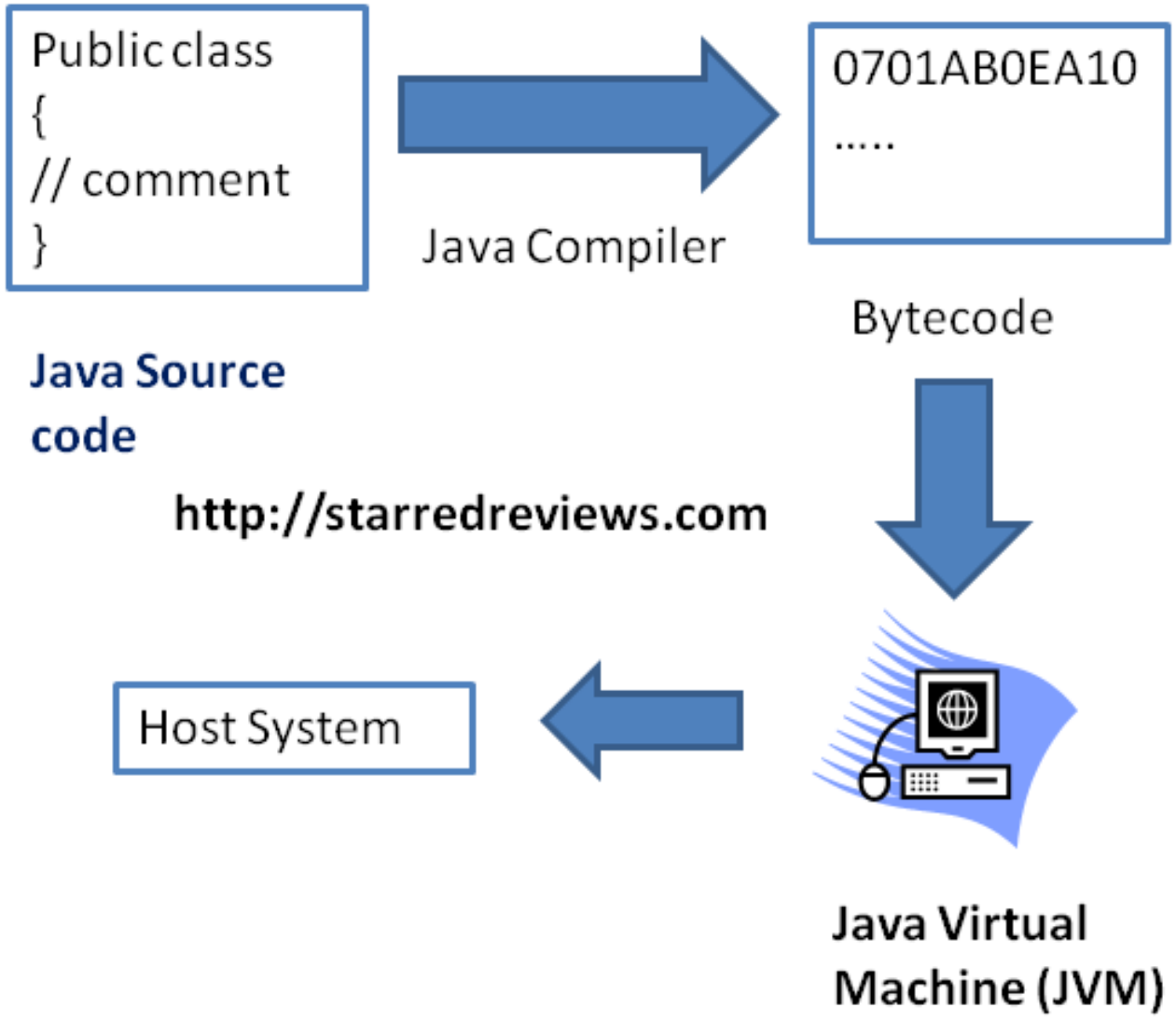
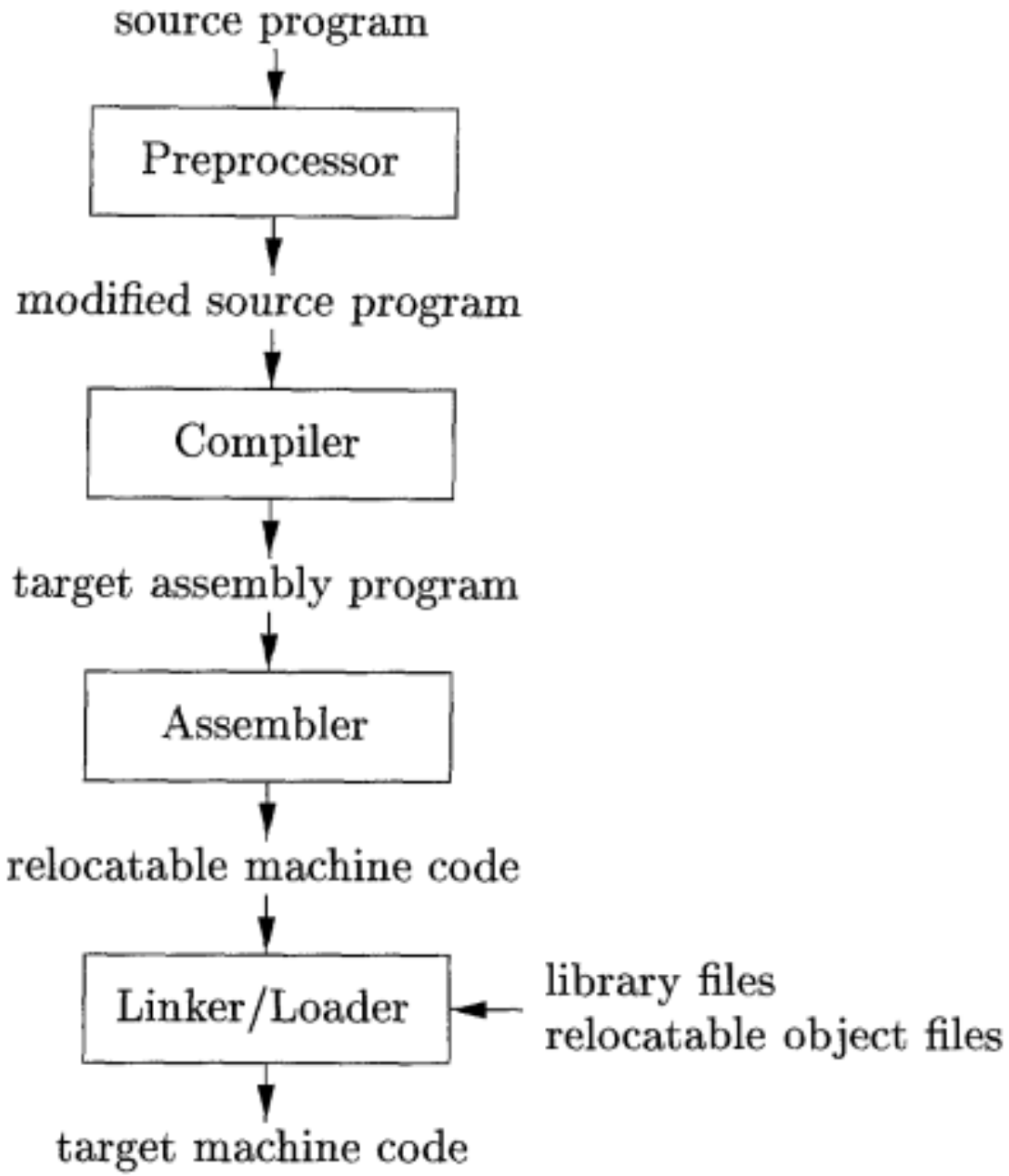


Image from http://starredreviews.com/wp-content/uploads/2011/04/JVM_Bytecode.png

Preprocessor

- In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1.5. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a preprocessor.
- The preprocessor may also expand shorthands, called macros, into source language statements.
- The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output.

- Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.
- The linker resolves external memory addresses, where the code in one file may refer to a location in another file.
- The loader then puts together all of the executable object files into memory for execution.



The Structure of a Compiler

- A compiler can be divided into two parts: analysis and synthesis.

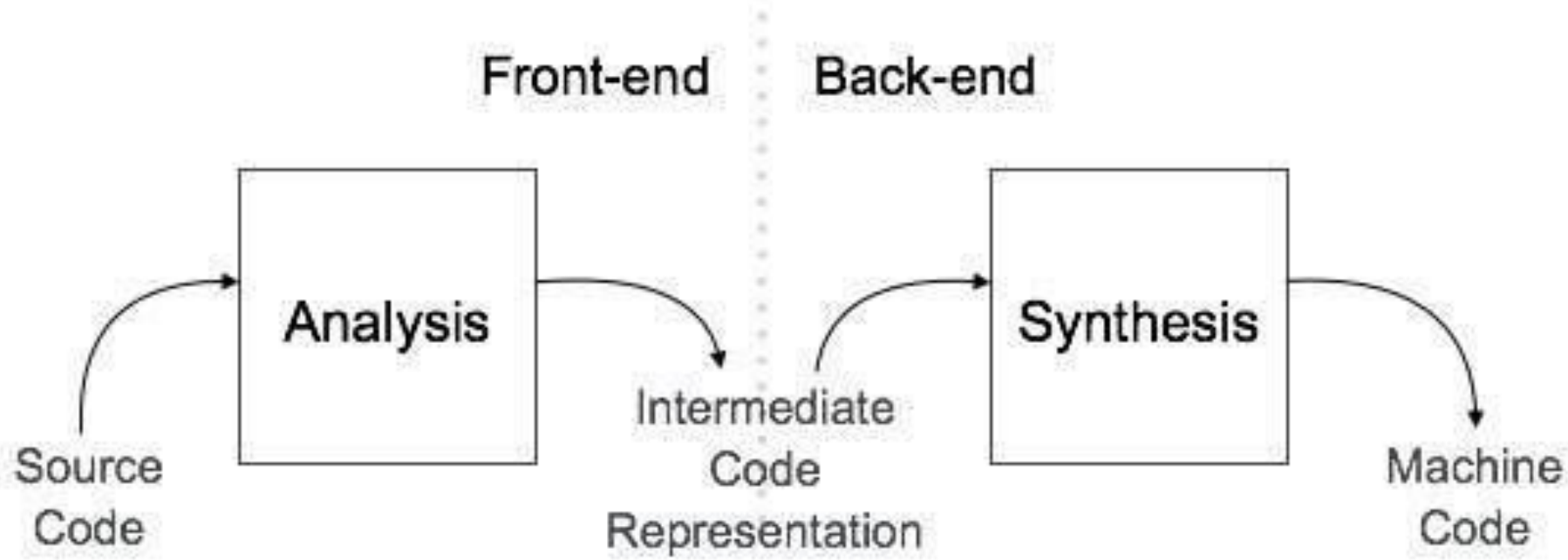


Image from https://www.tutorialspoint.com/compiler_design/images/compiler_analysis_synthesis.jpg

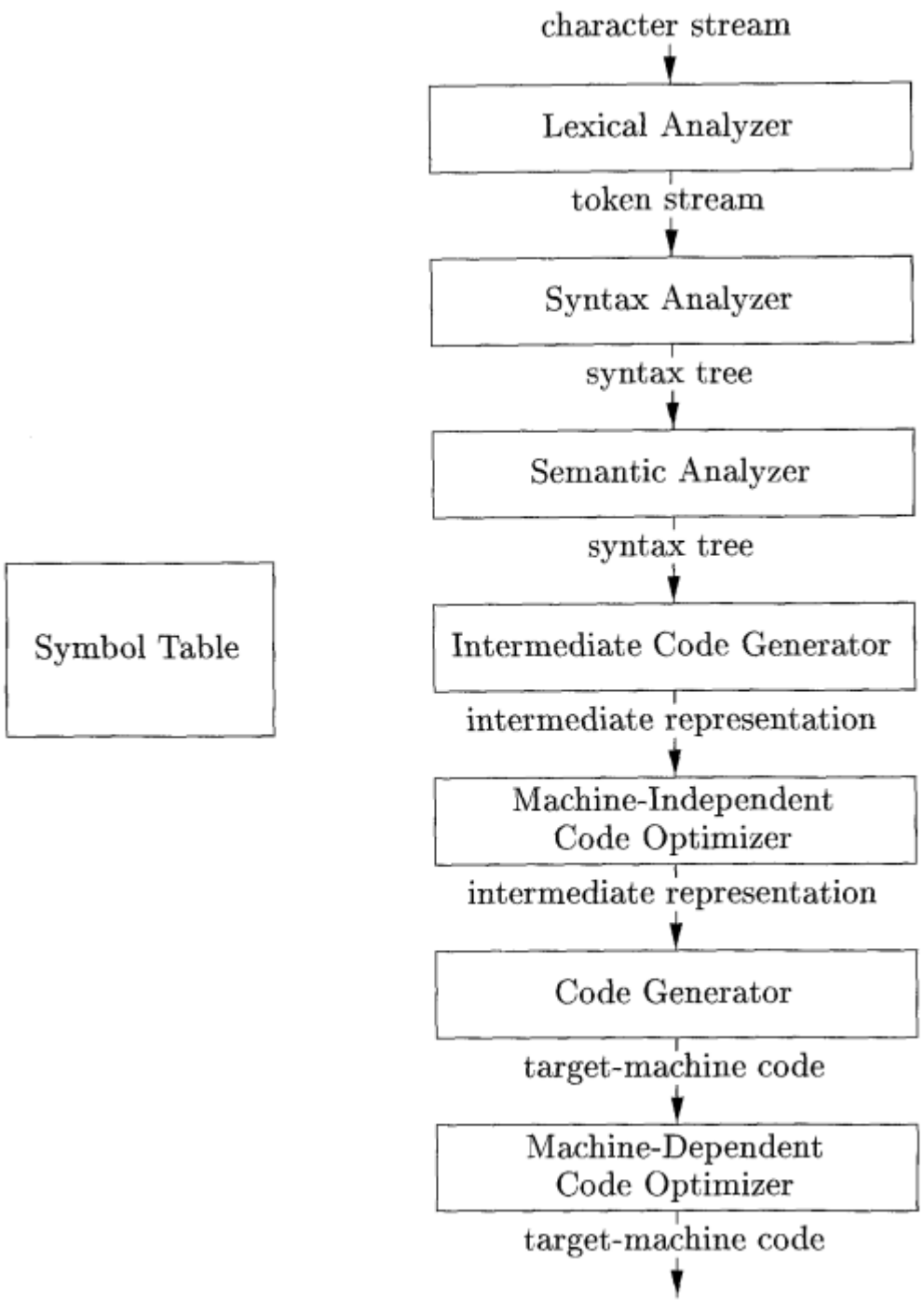
Analysis part

- The **analysis** part reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.
- The analysis part generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis part as input.

Synthesis part

- The **synthesis** part generates the target program with the help of intermediate source code representation and symbol table.

Phases of a compiler



Lexical Analysis (scanner)

- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.
- For each lexeme, the lexical analyzer produces as output a token of the form:

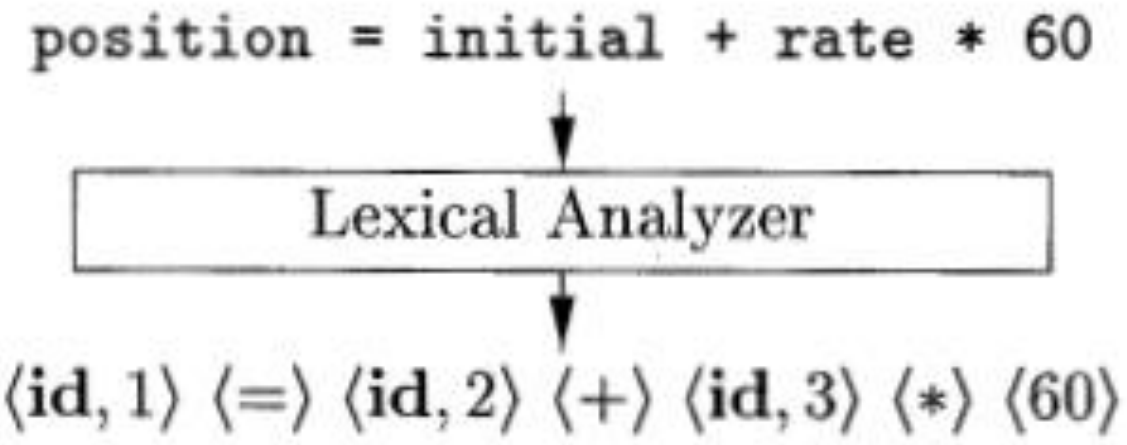
(token-name, attribute-value)

that it passes on to the subsequent phase, syntax analysis

position = initial + rate * 60

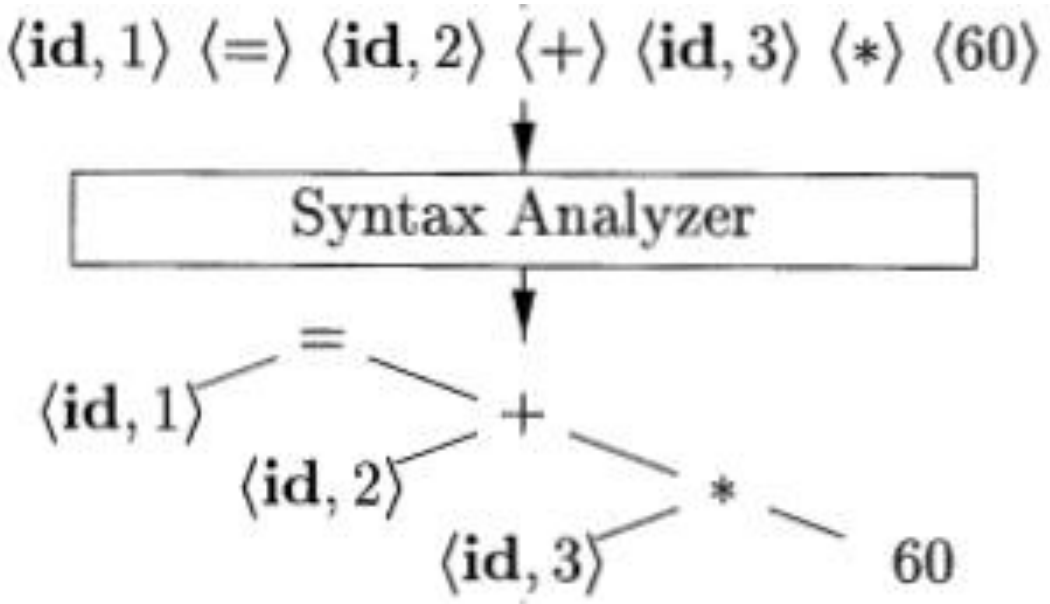
Example

- position is a lexeme that would be mapped into a token (id, I), 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
- The assignment symbol = is a lexeme that is mapped into the token (=)
- initial is a lexeme that is mapped into the token (id, 2)
- + is a lexeme that is mapped into the token (+)
- * is a lexeme that is mapped into the token (*)
- 60 is a lexeme that is mapped into the token (60)



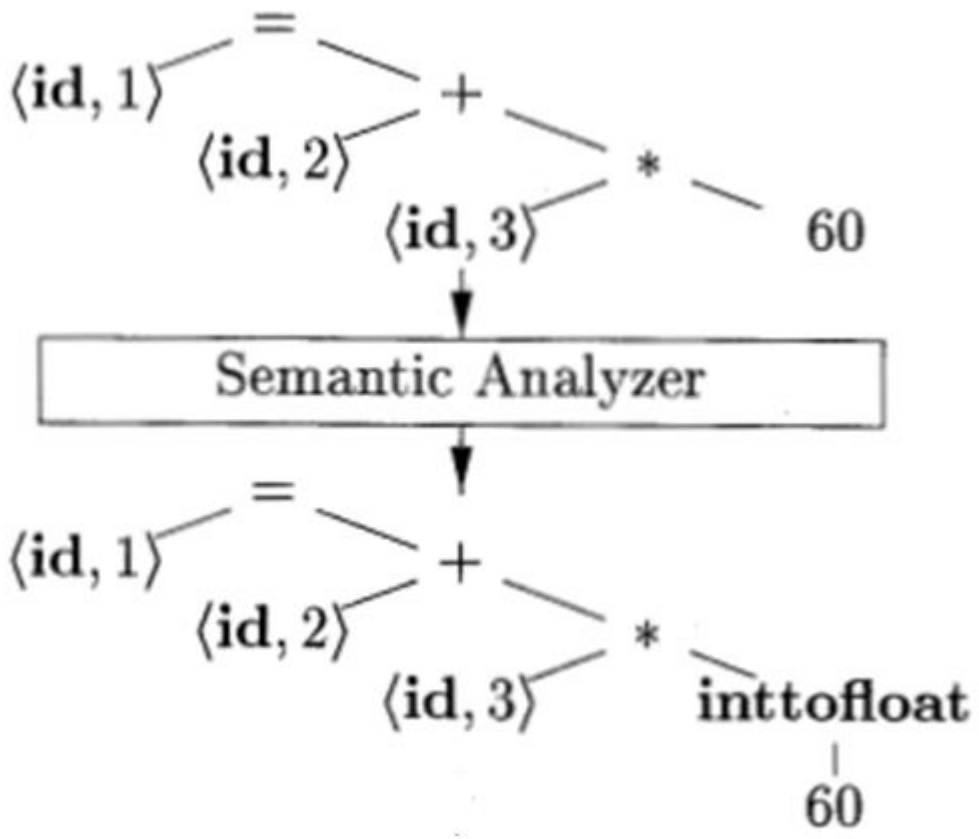
Syntax Analysis (parser)

- It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).
- In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.



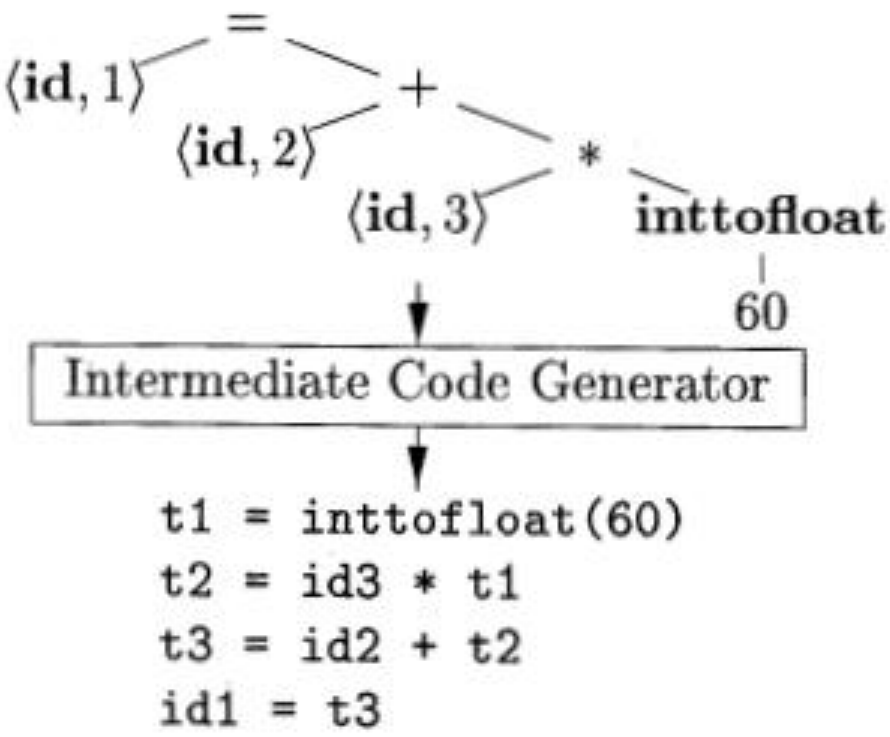
Semantic analysis

- Semantic analyzer checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer.
- Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.



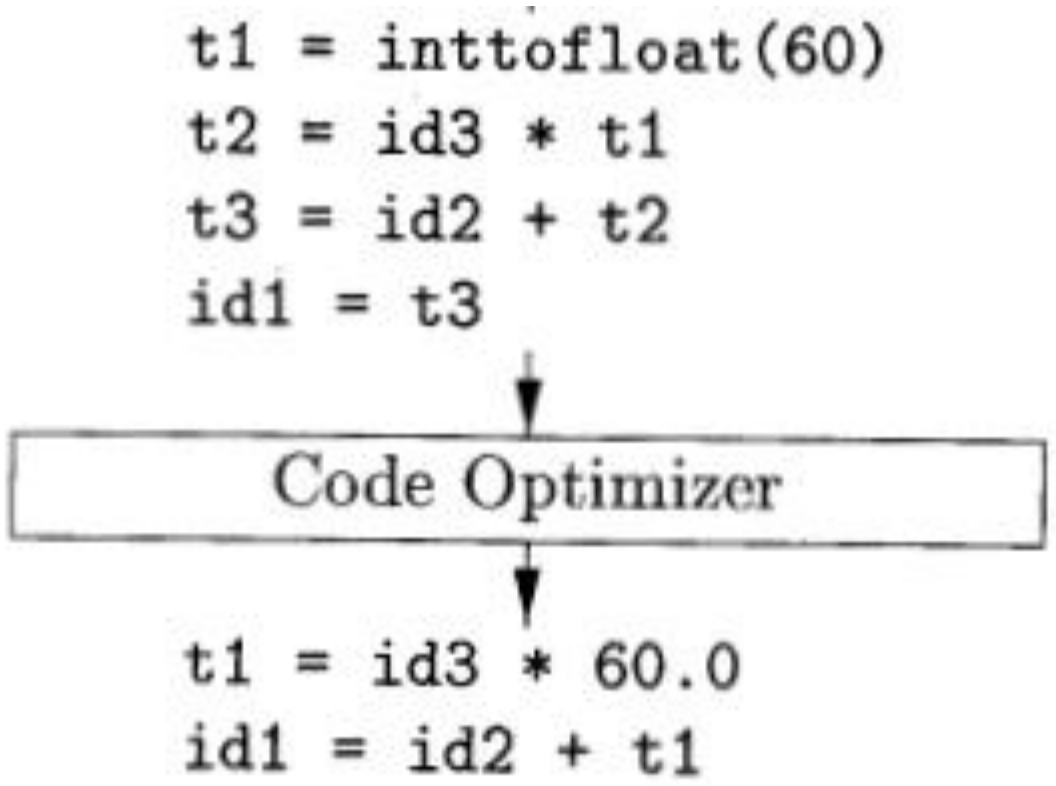
Intermediate Code Generation

- After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.



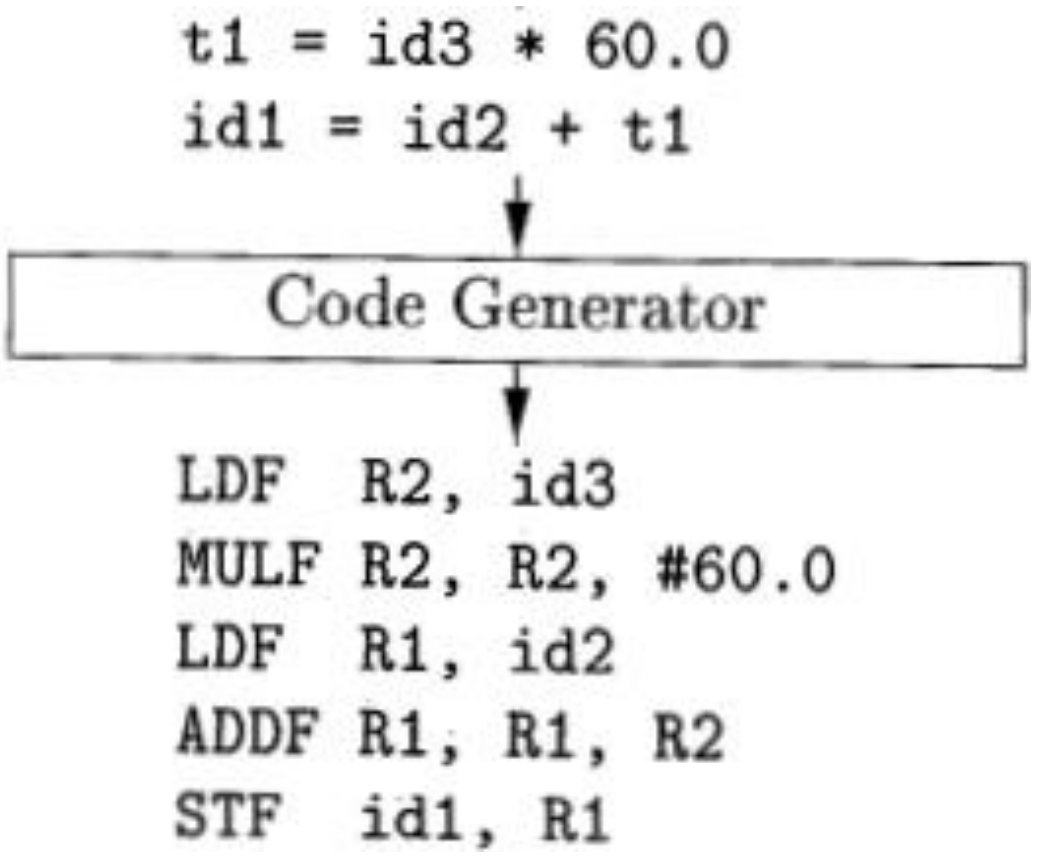
Code Optimization

- The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).
-



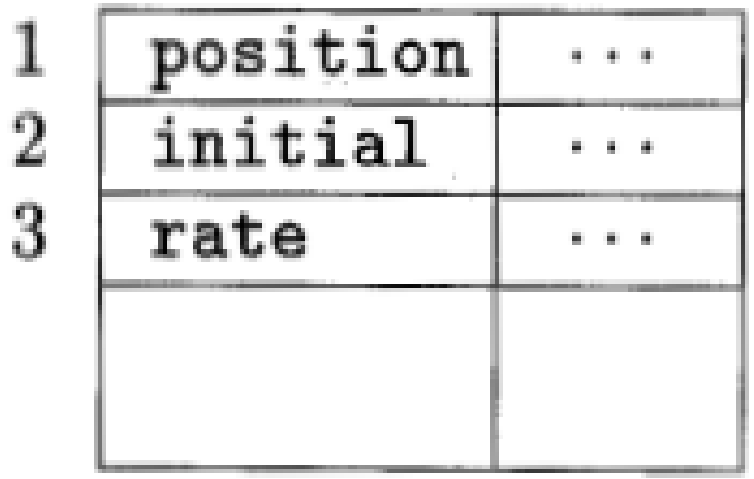
Code Generation

- In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.



Symbol Table

- It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.



SYMBOL TABLE

Refernces

- Compilers: Principles, Techniques, and Tools ,A. V. Aho, R. Sethi, J. D. Ullman; (c) 2010

- https://www.tutorialspoint.com/compiler_design

- http://starredreviews.com/wp-content/uploads/2011/04/JVM_Bytecode.png