東南大學
SOUTHEAST UNIVERSITY

Compiler Principle
——Introduction

Zhizheng Zhang
Southeast University

1. What is a Compiler?

A Program written in some Programming Languages

Translation

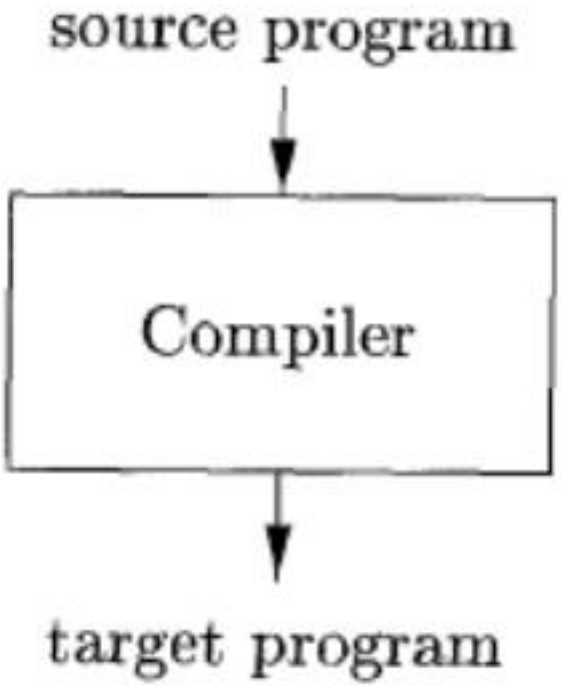
A sequence of machine instructions (executable program)

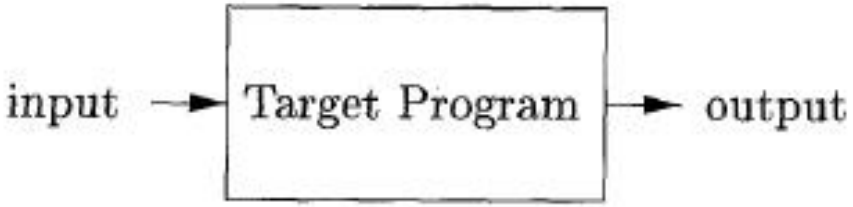
COMPILER

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

2. Programming Language Processors

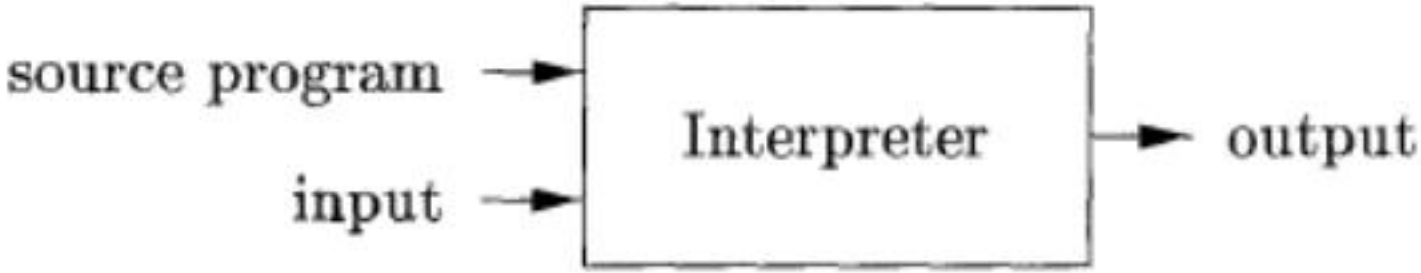
2.1 Compiler





If the **target program** is an executable machine-language program

2.2 Interpreter



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

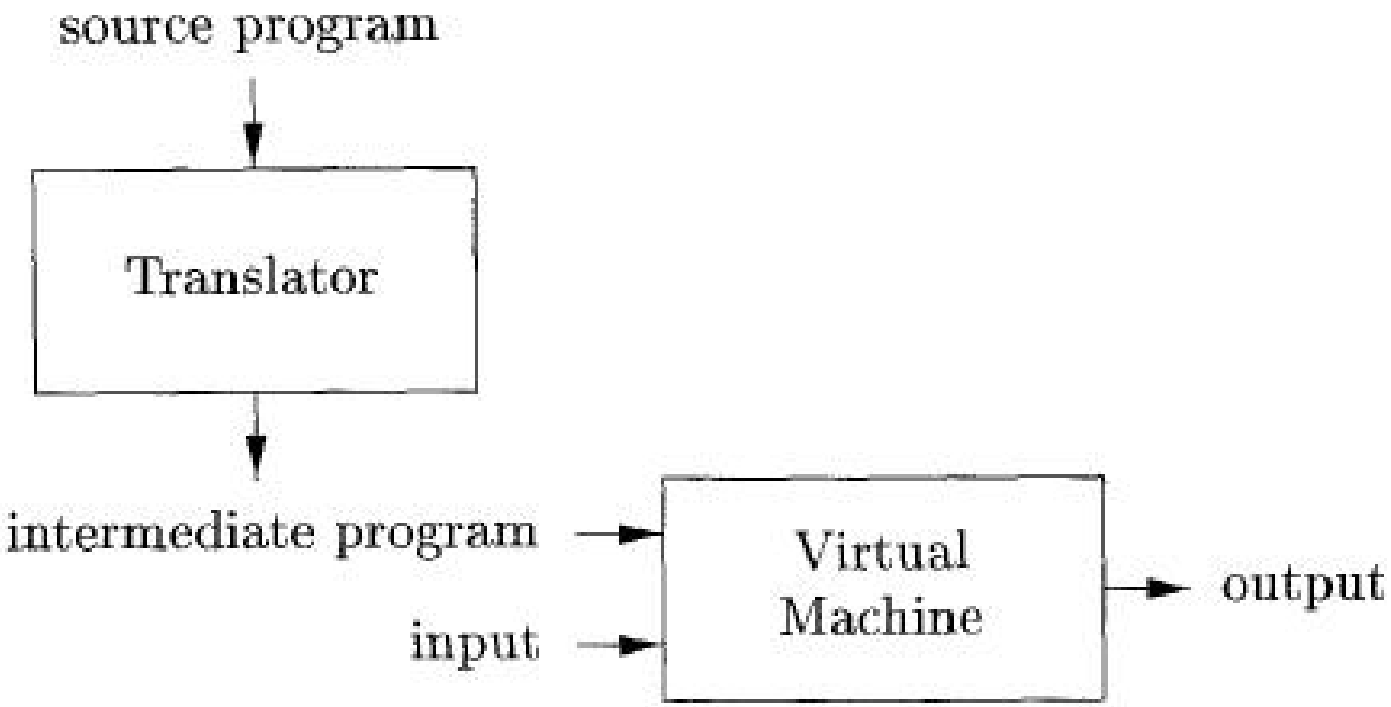
Example. HTML language, Basic Language, Prolog etc,.

The core of a browser software such as IE, CHROME, is an interpreter of HTML.

- 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.

2.3 Hybrid Processor

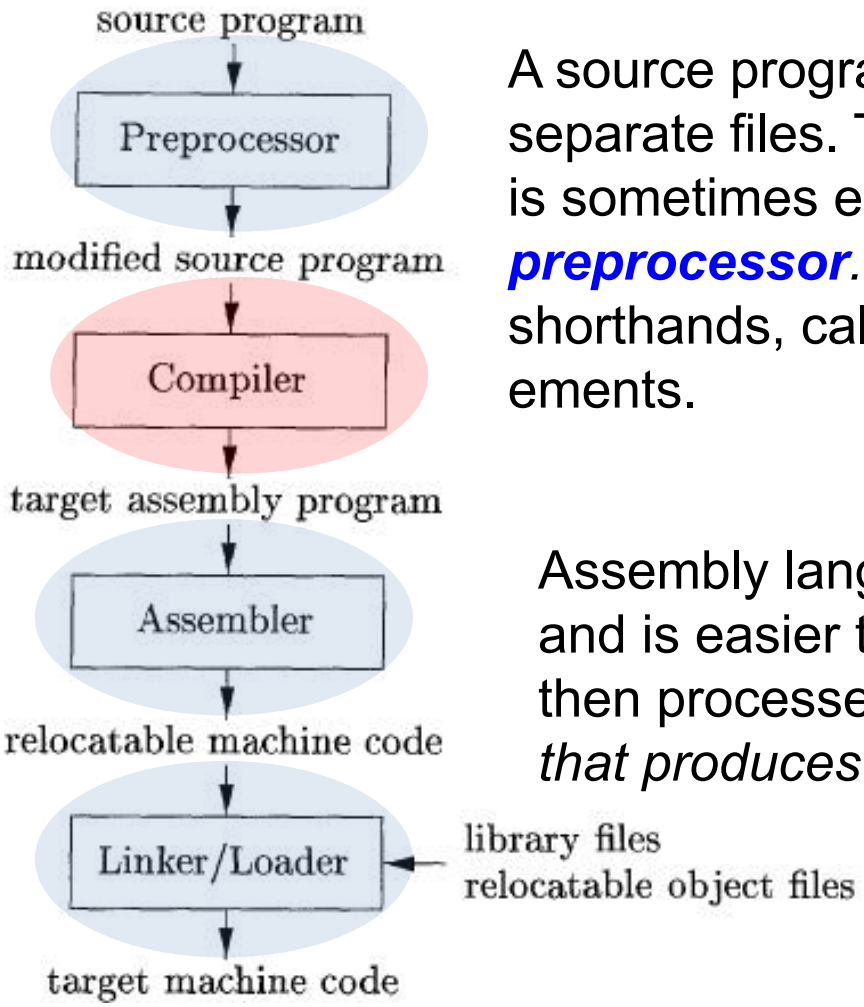


Example. Java language processors combine compilation and interpretation.

A Java source program may first be compiled into an intermediate form called *bytecodes*. *The bytecodes are then interpreted by a* virtual machine.

A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

2.4 A Typical Procedure



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 stat ements.

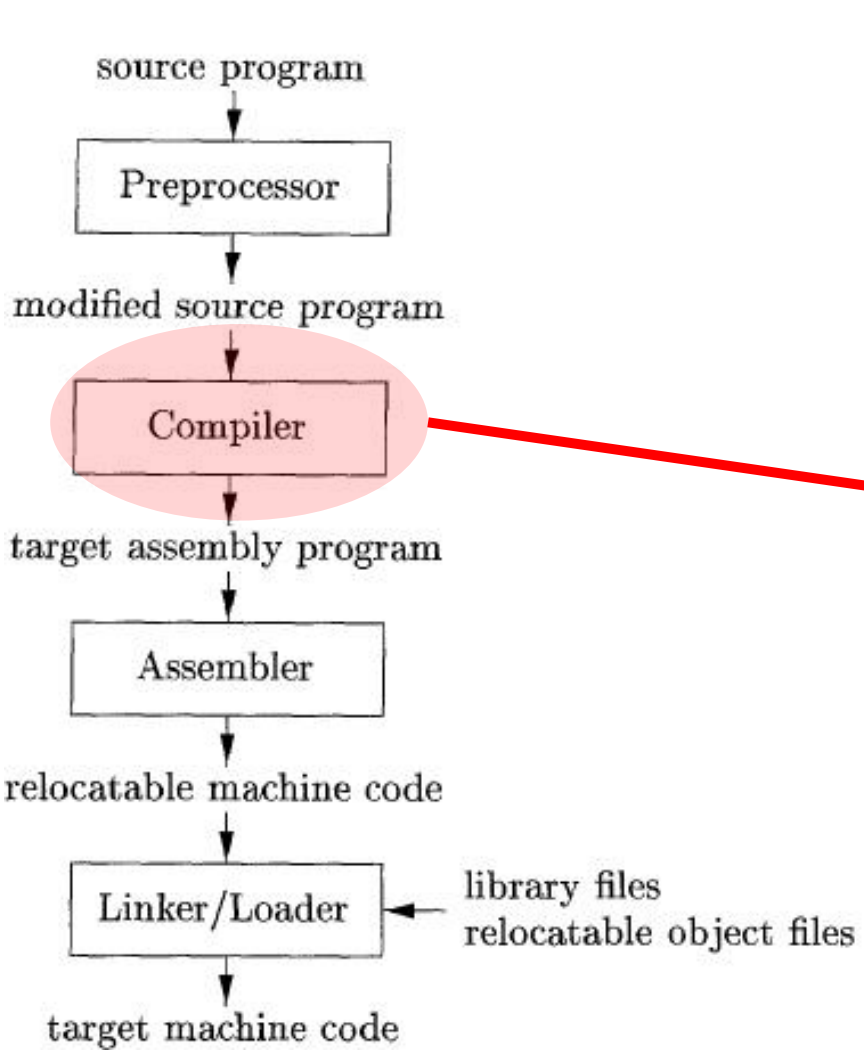
Assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an ***assembler*** *that produces* relocatable machine code as its output.

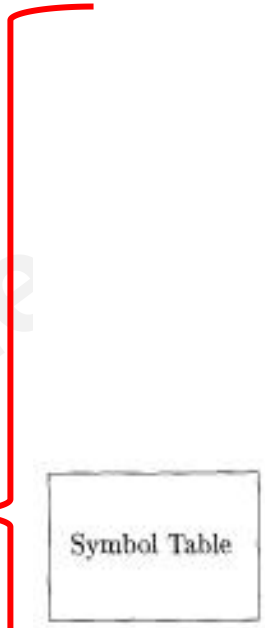
Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The ***linker resolves*** external memory addresses, where the code in one file may refer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

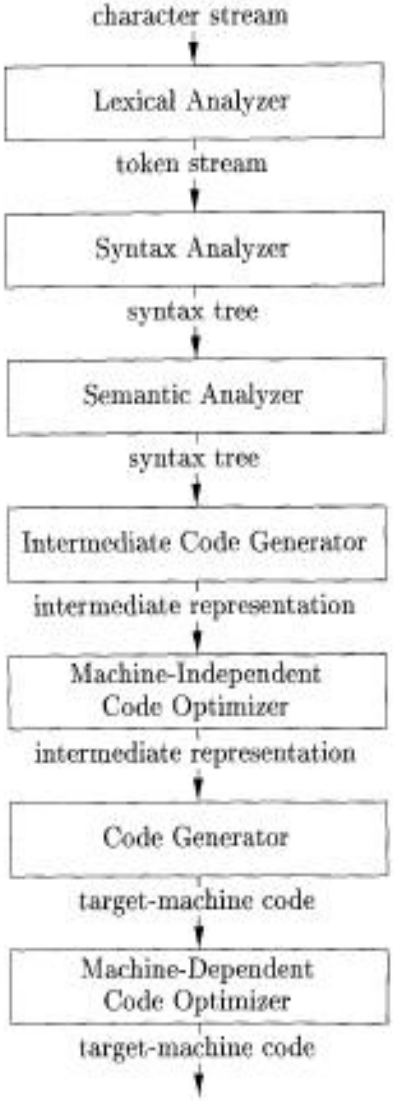
Assignment

- **Exercises for Section 1.1**

3. The Structure of a Compiler



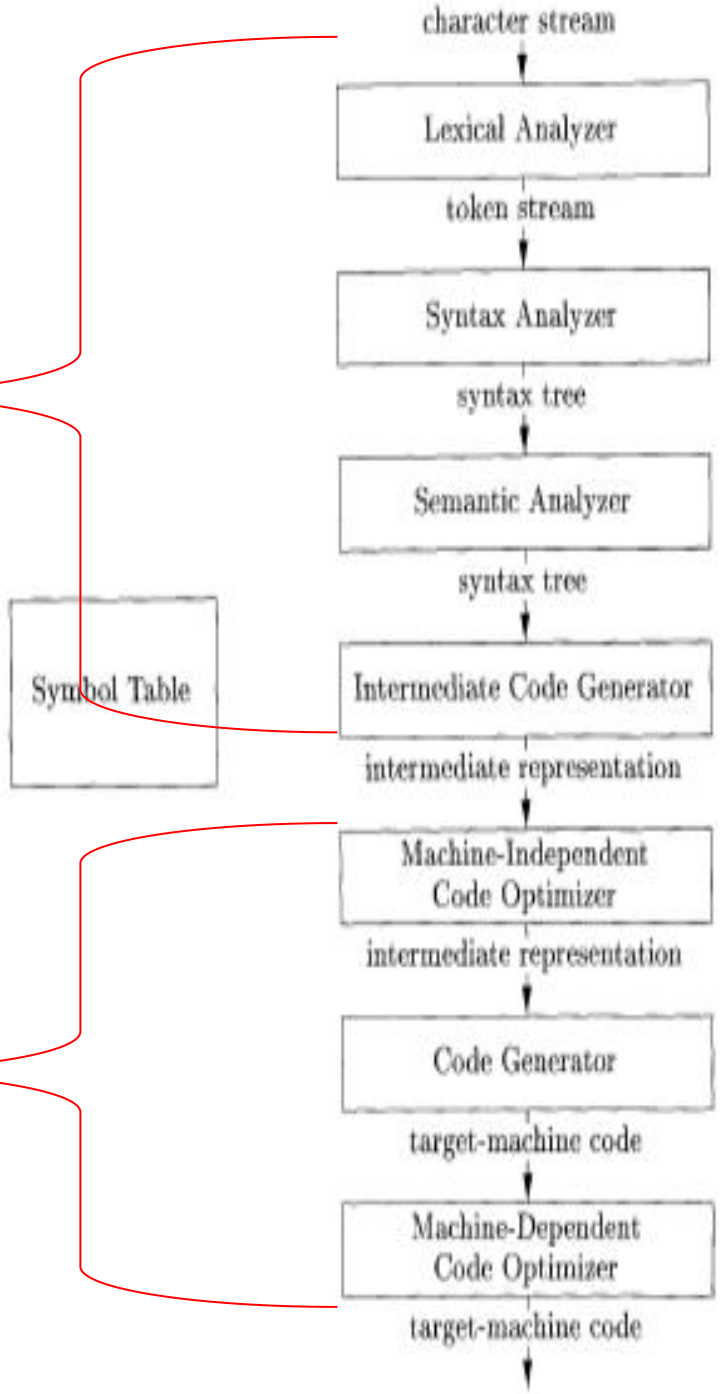




3.1 Two Part

The **analysis part** breaks up the source program into constituent pieces and imposes a grammatical structure on them, collects information about the source program and stores it in a data structure called a **symbol table.**

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



3.2 Eight Modules

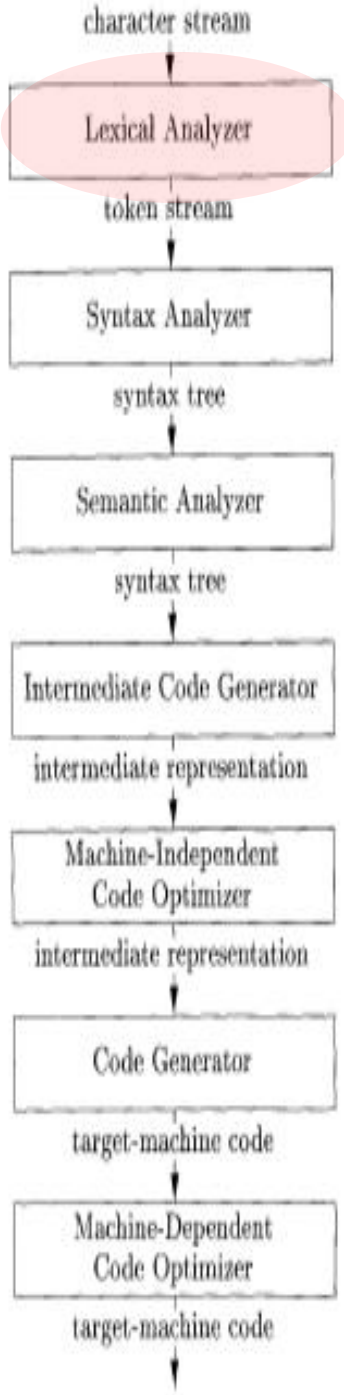
- Lexical Analysis
- Syntax Analysis
- Semantics Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generation
- Symbol Table Management

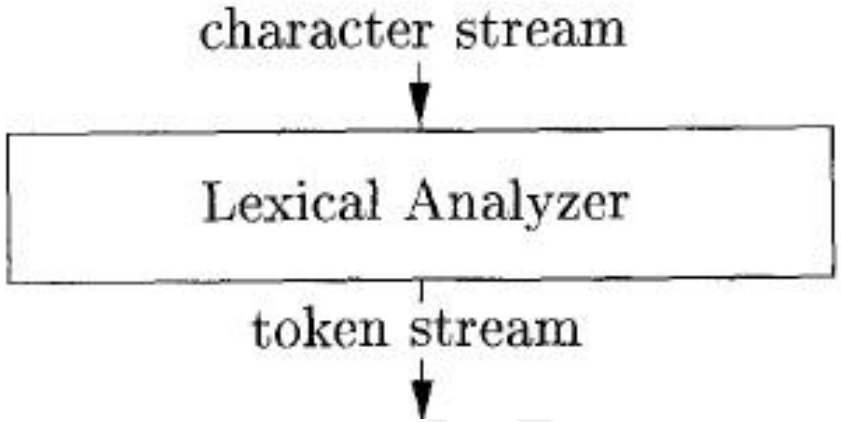
3.2.1 Lexical Analysis

Lexical Analyzer /Scanner

- Reads the stream of characters making up the source program,

- groups the characters into meaningful sequences called *lexemes*.





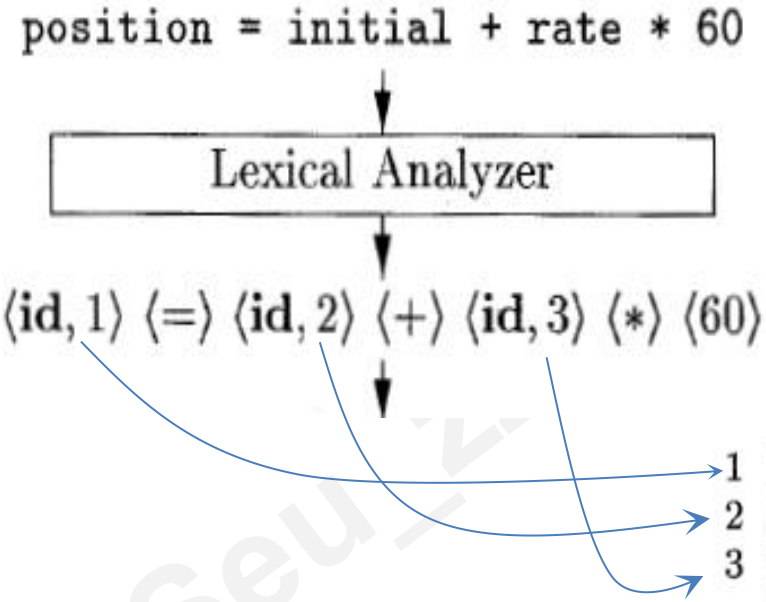
A Token is of the form:

<token-name, attribute-value>

where *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.

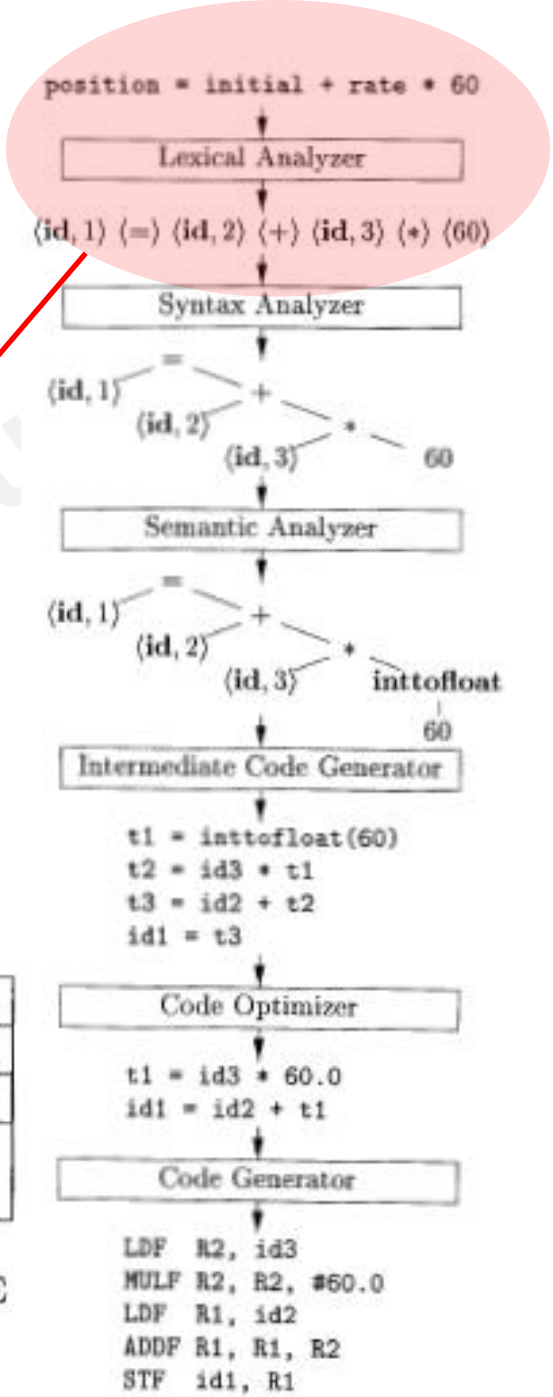
Example.

position = initial + rate * 60



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

SYMBOL TABLE

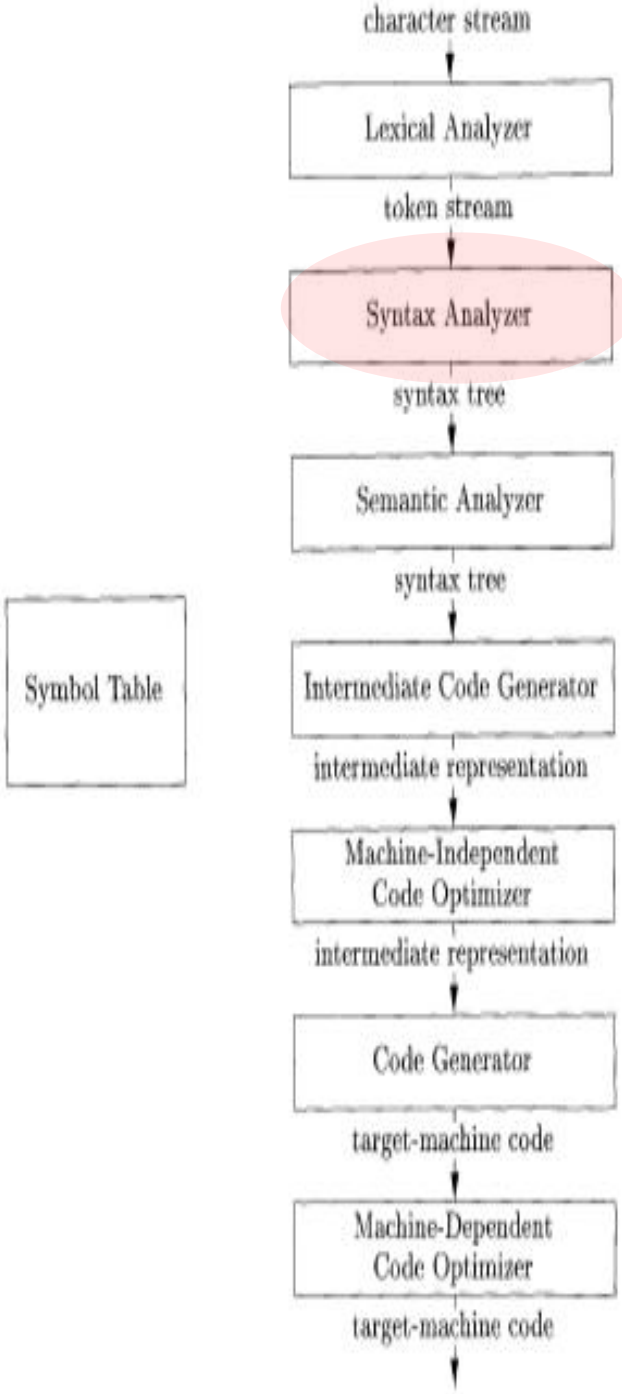


3.2.2 Syntax Analysis

Syntax Analyzer /Parser

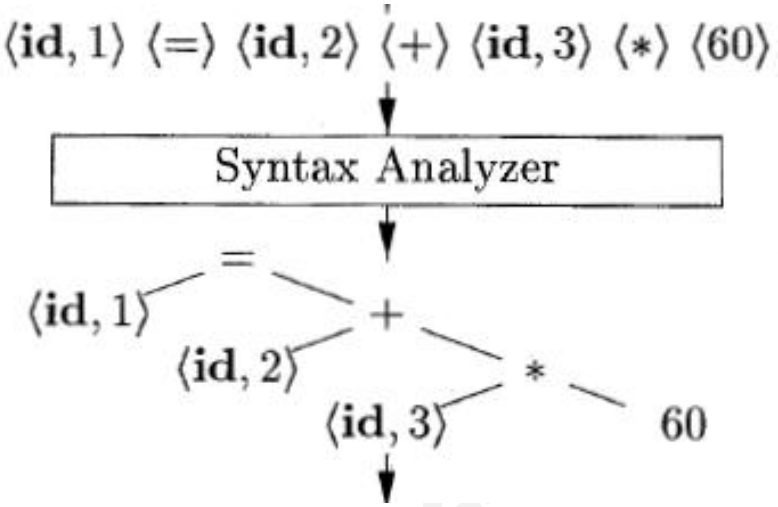
- Reads the stream of tokens generated by the scanner,

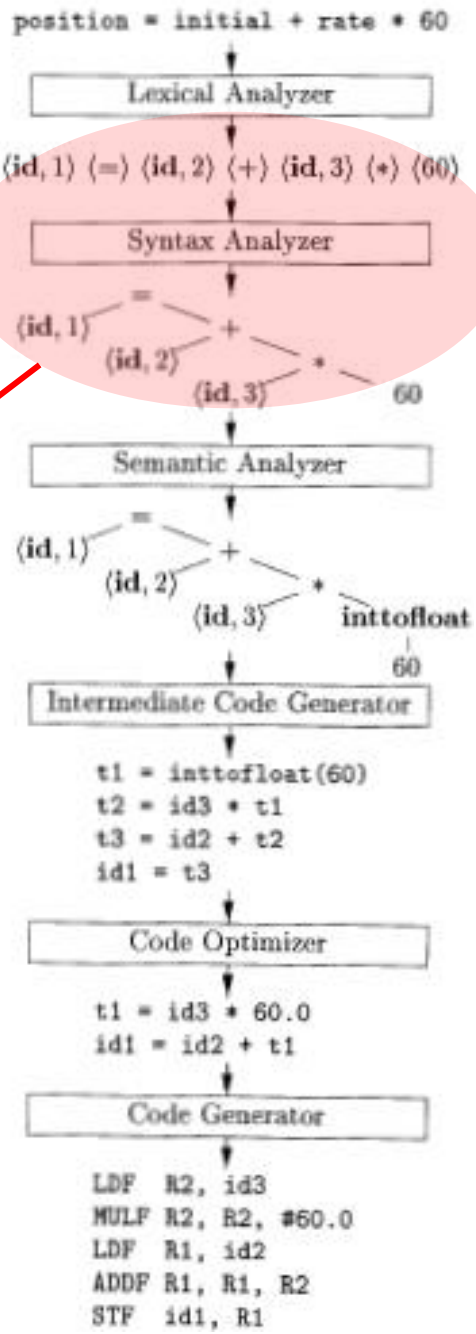
- groups the tokens into *sentential forms* represented as a *syntax tree* by the *construction laws* of the programming language.



Example.

position = initial + rate * 60

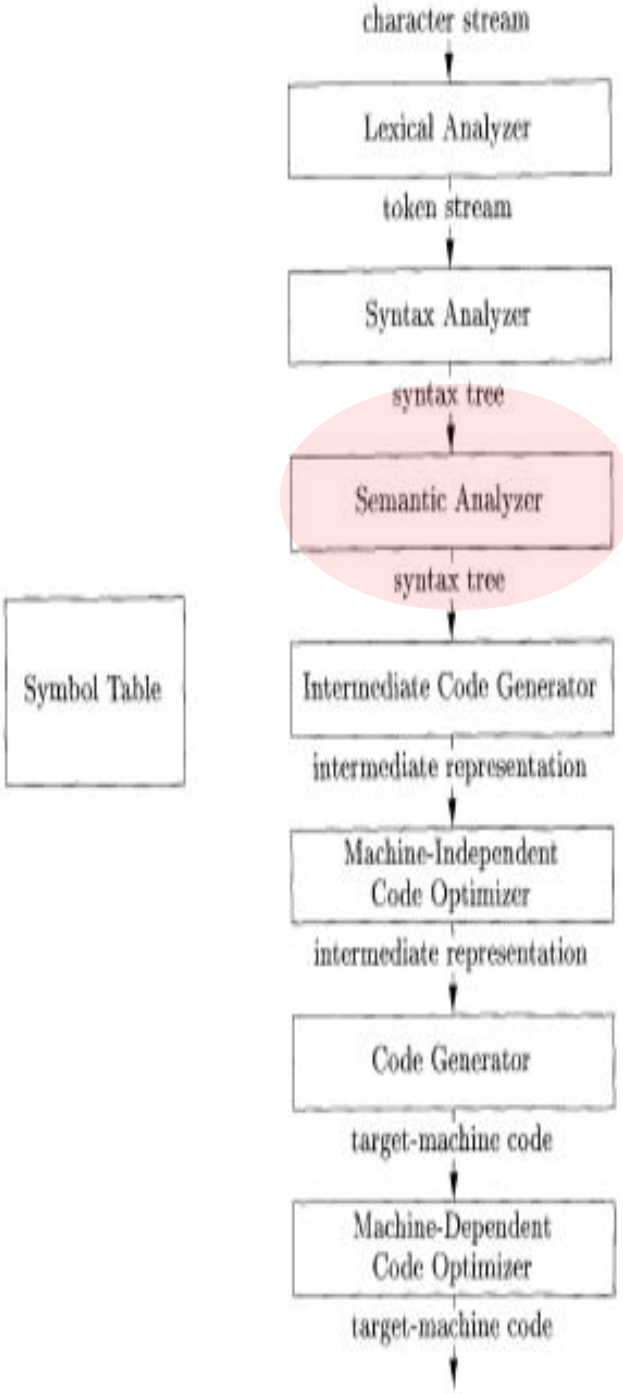




3.2.3 Semantics Analysis

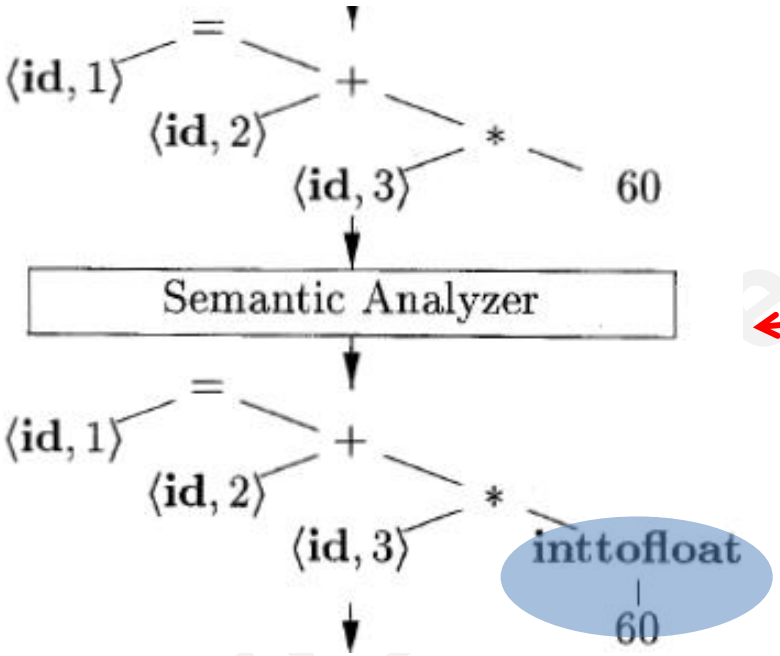
- Uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table.

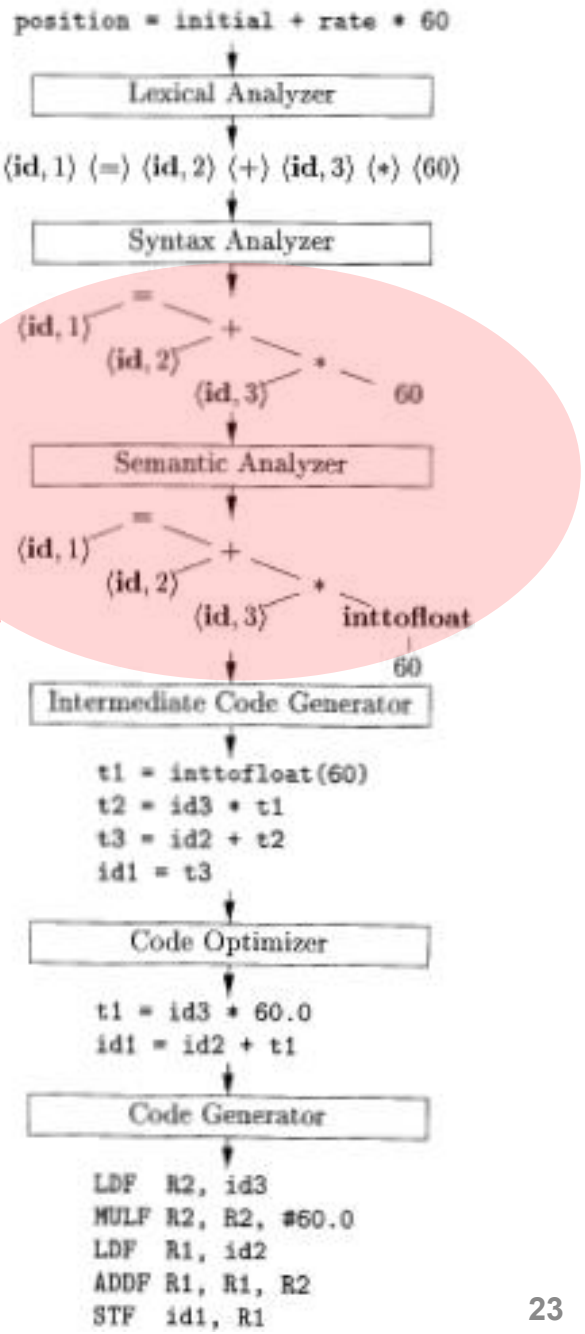
- Do *type checking*.



Example.

position = initial + rate * 60



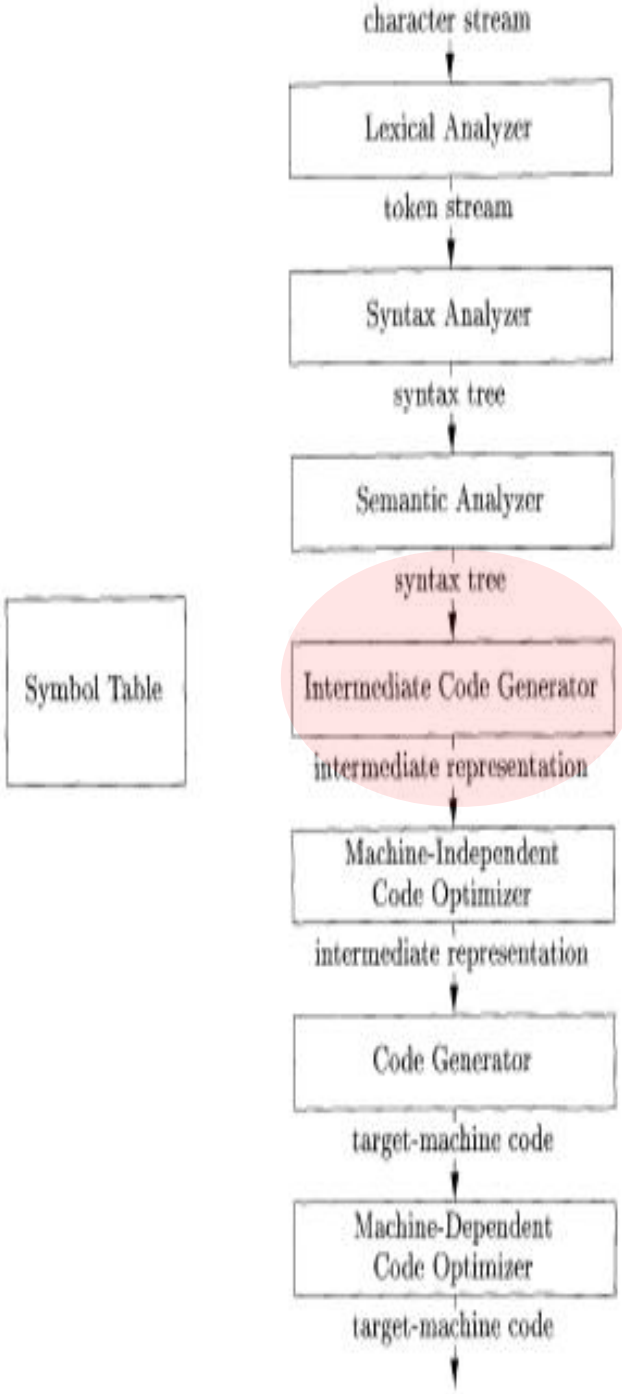


3.2.4 Intermediate Code

- Uses the syntax tree and produces codes of the form that **is easy to produce and is easy to translate into the target machine**.

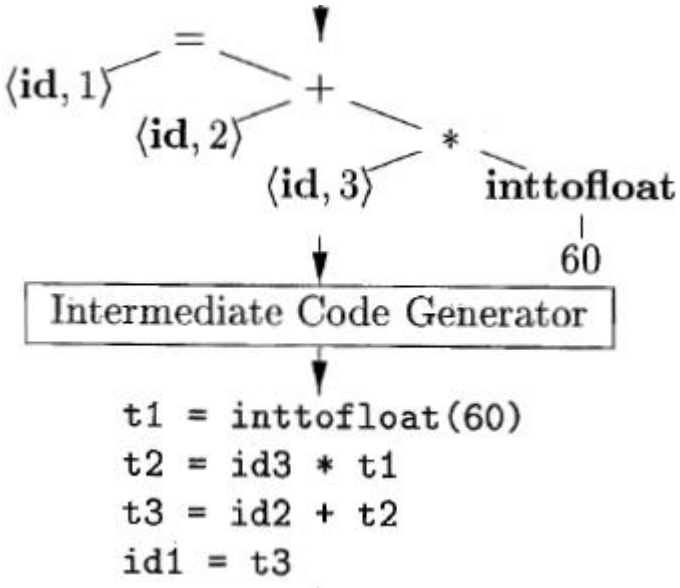
Why?

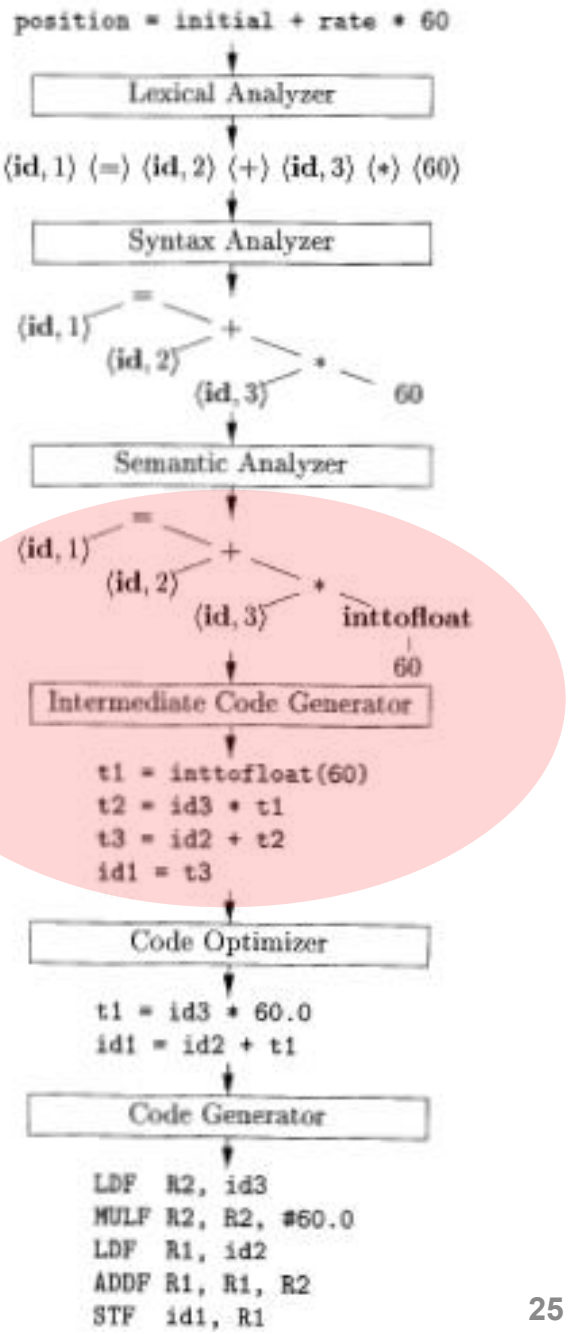
Code optimization.



Example.

position = initial + rate * 60

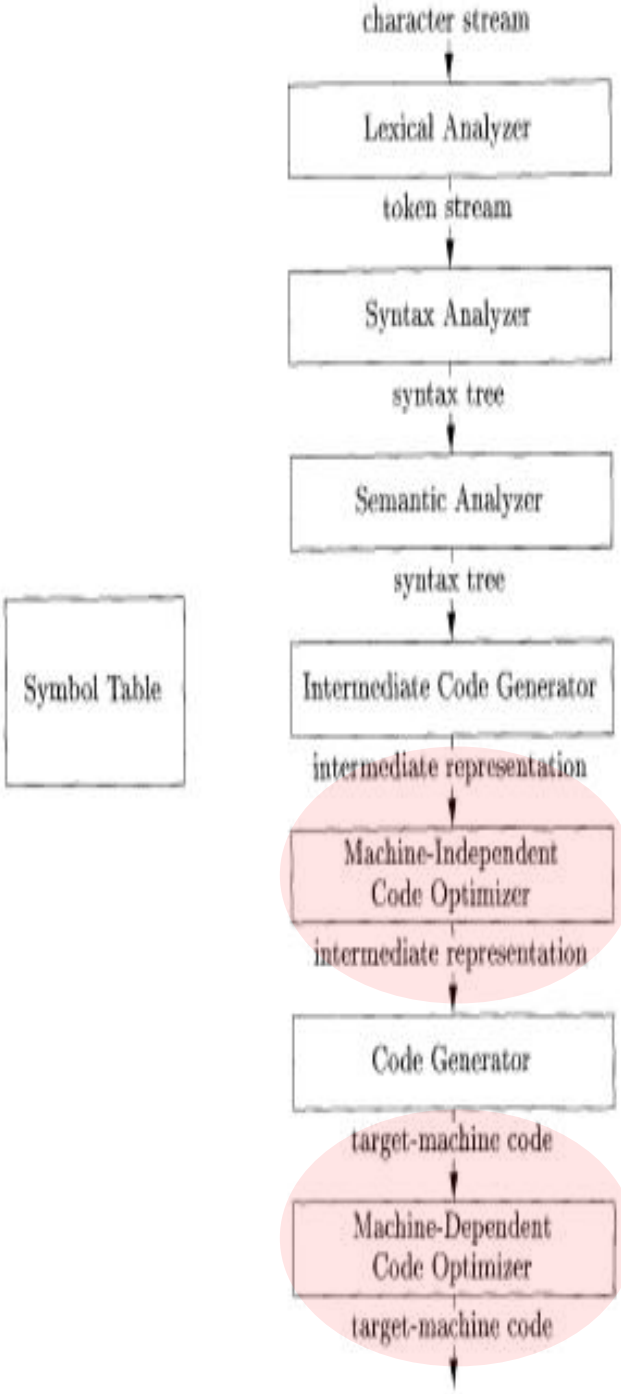




3.2.5 Code Optimization

- Machine-independent code optimization.

- Machine-dependent code optimization.



Example.

position = initial + rate * 60

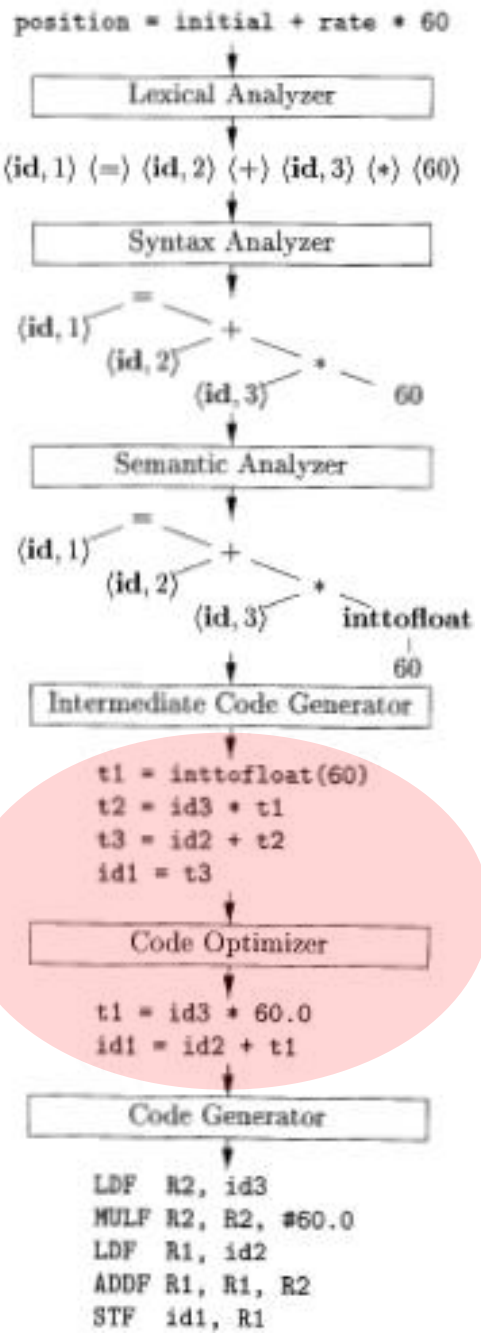
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

↓

Code Optimizer

↓

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

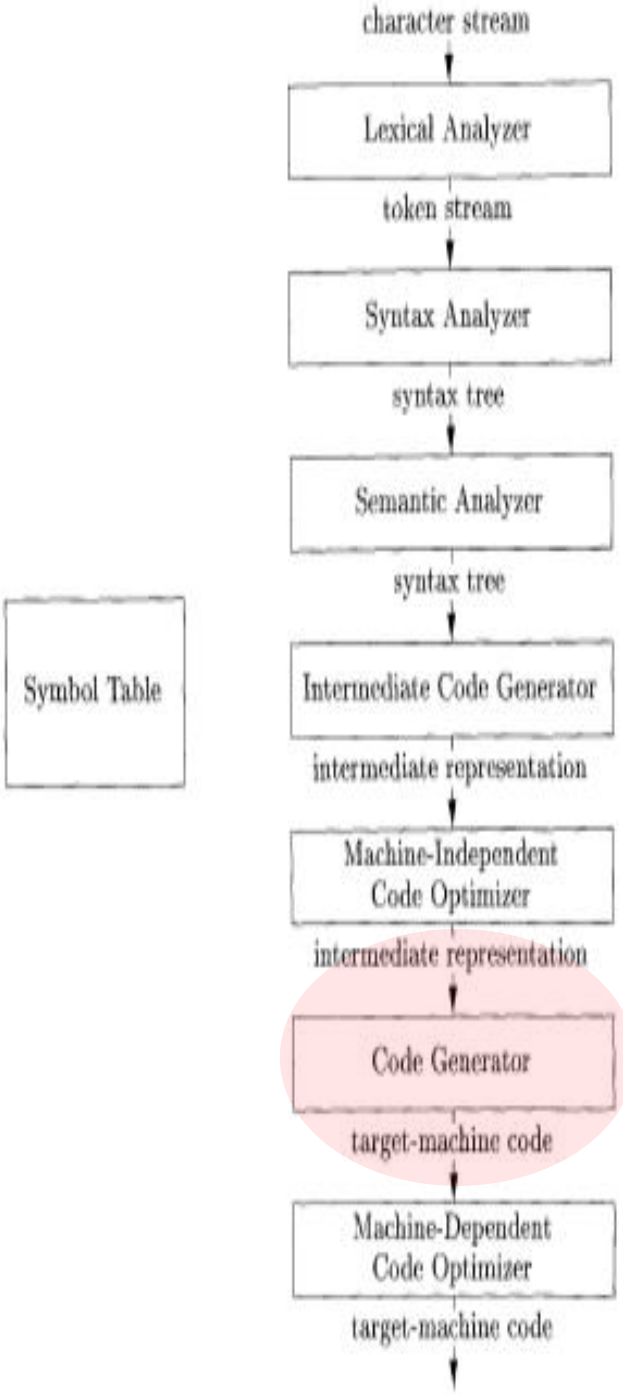


3.2.6 Code Generation

•The code generator takes as input an intermediate representation 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 is the judicious assignment of registers to hold variables



Example.

position = initial + rate * 60

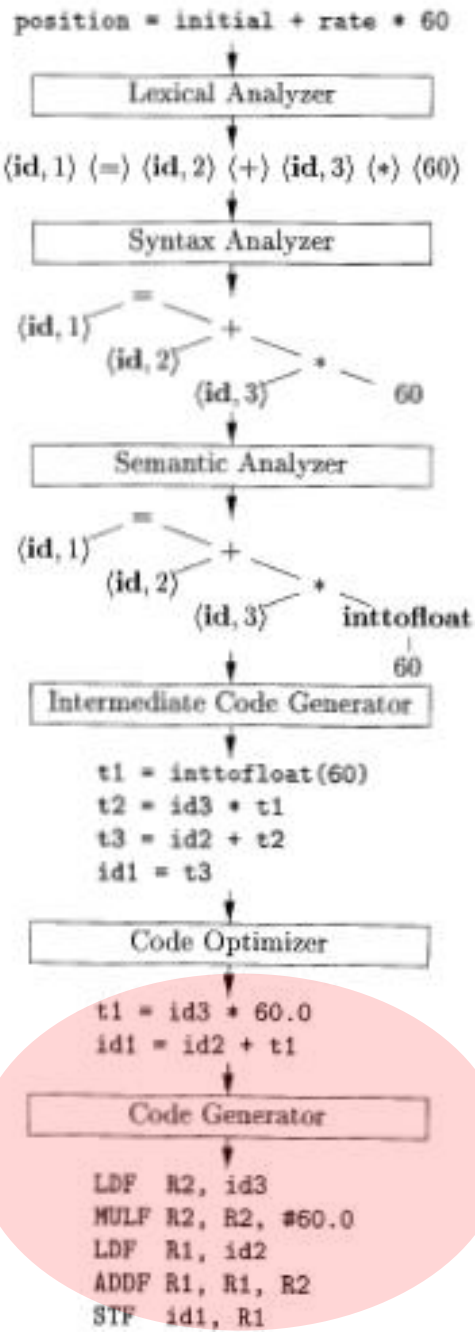
```
t1 = id3 * 60.0
id1 = id2 + t1
```

↓

Code Generator

↓

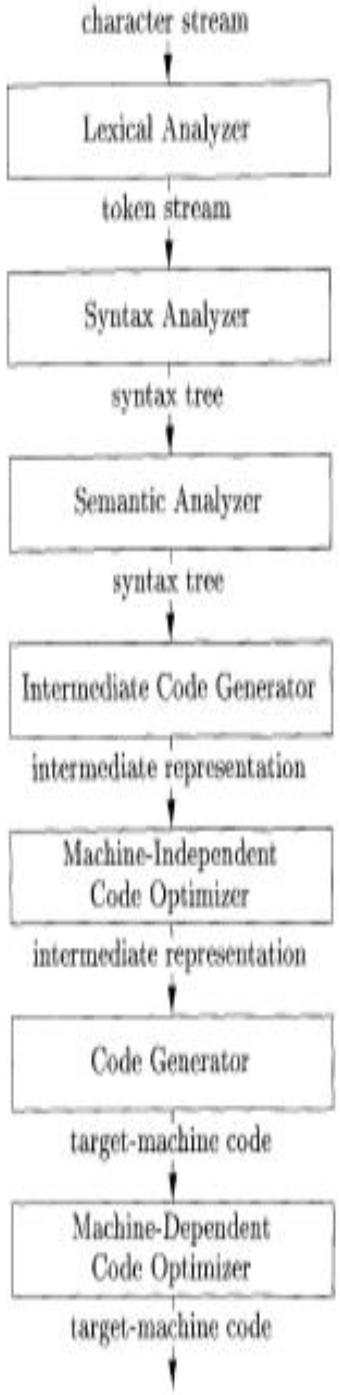
```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```



3.2.7 Symbol Table Management

The symbol table is a data structure containing a record for each *variable name*, with fields for the attributes of the name.

These attributes may provide the storage allocated for a name, its type, its scope, 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



Example.

position = initial + rate * 60

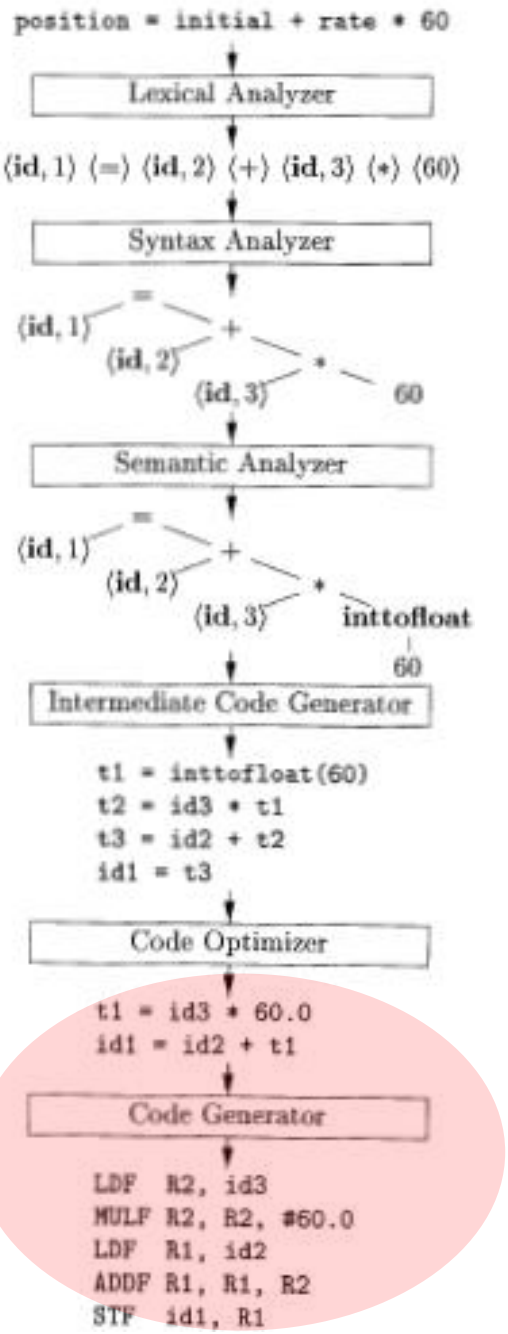
```
t1 = id3 * 60.0
id1 = id2 + t1
```

↓

Code Generator

↓

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



4. Compilers-Construction Tools

1. *Parser generators*.
2. *Scanner generators*.
3. *Syntax-directed translation engines*.
4. *Code-generator generators*.
5. *Data-flow analysis engines*.
6. *Compiler-construction toolk2ts*.

5. The Evolution of Languages

1st-generation: machine languages,

2nd-generation: the assembly languages

3nd-generation: the higher-level languages like Fortran, Cobol, Lisp, C, C++, C#, and Java.

4th-generation: designed for specific applications like NOMAD for report generation, SQL for database queries, and Postscript for text formatting.

5th-generation : Prolog and OPS5.

More Terms:

- Imperative language / procedural language
- Declarative language
- von Neumman Language
- Object-oriented language
- Scripting language

6. Why Do We Learn Compilers?

1. **Compilers employ many important multidisciplinary science topics.**
   - Computational Linguistics
   - Theoretical Computer Science
   - Artificial Intelligence

2. **There are a great many of applications of compiler technology.**

Application 1. Implementation of High-Level Programming

Application 2. Optimization for Computer Architecture

7. Advantages of Knowing the Compiler Principle

1. Promoting coding ability/skills.

2. Having a good case of software engineering.

3. Get a lot of basic ideas and technology in computer science and other fields.