Compiler Construction

CS-471

By Laeeq Khan Niazi

Dua

اللَّٰهُمَّ إِنِّي أَسْأَلُكَ عِلْمًا نَافِعًا وَرِزْقًا طَيِّبًا وَعَمَلاً مُتَقَبَّلاً

O Allah, I ask You for beneficial knowledge, goodly provision and acceptable deeds

CLOs

- Understand and Apply Mathematical Formalisms like Regular Expressions, Grammars, FAs & PDAs.
- Understand the working principles of various phases of a compiler and how they are integrated to produce a working.
- Compare Various Implementations Techniques of Phases of Compiler, in particular Lexical Analyser, Parser
- Be able to work in a team (preferably alone) to design, develop, test, and deliver analysis phases of a compiler

Books

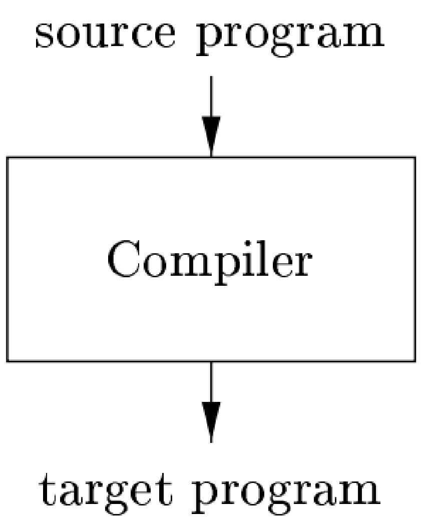
- Compiler – Principal, Techniques and Tools by Aho, Sethi and Ullman
- Engineering a Compiler – Keith D. Cooper & Linda Torczon 2nd Edition

Agenda

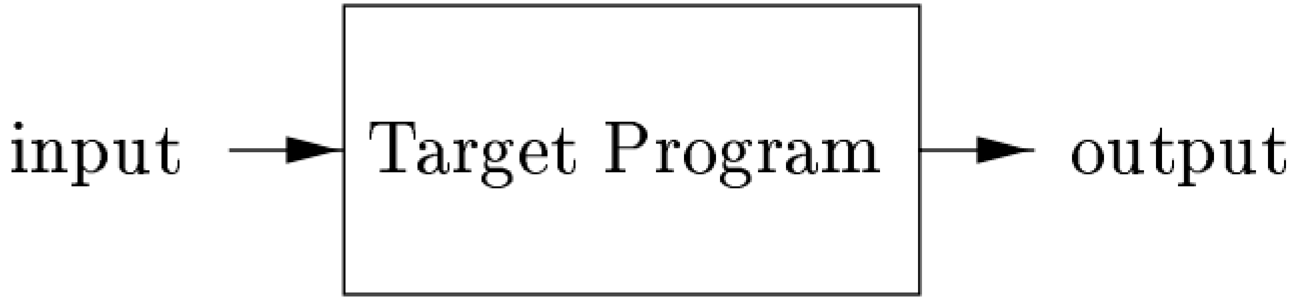
- Language Processors
- The Structure of a Compiler
- The Evolution of Programming Languages
- The Science of Building a Compiler
- Applications of Compiler Technology
- Programming Language Basics

Compiler

- Simply stated, 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

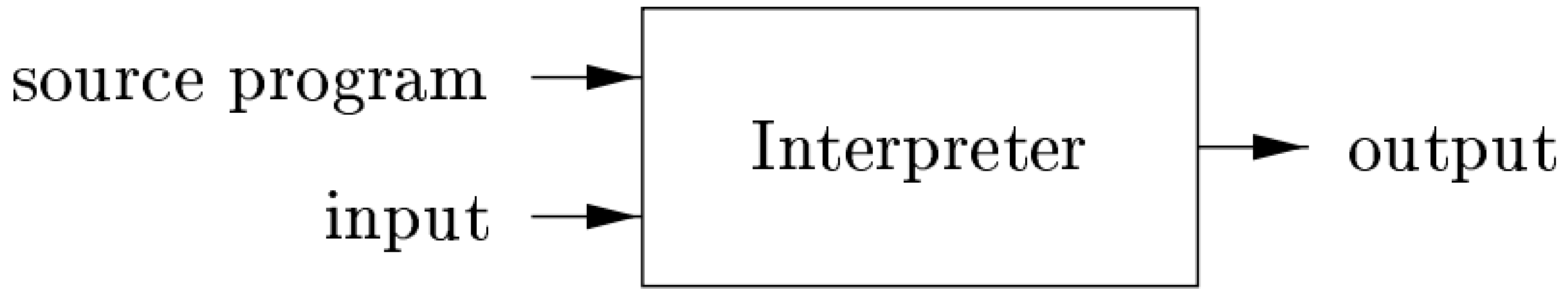


- If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs



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 specied in the source program on inputs supplied by the user

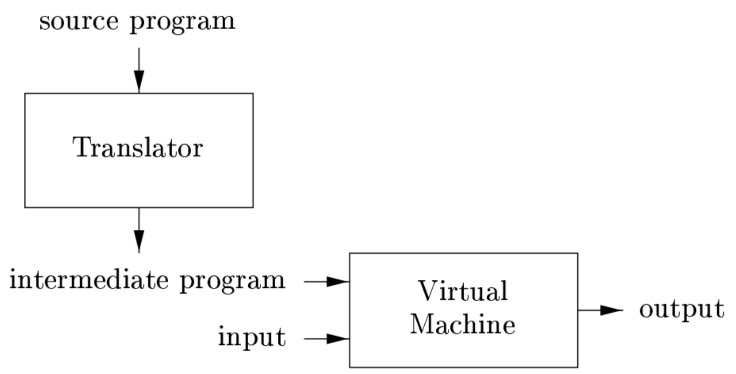


Difference

- 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

How Java compilation happens

- Java language processors combine compilation and interpretation.
- A Java source program first compiled into an intermediate form called bytecodes.
- The bytecodes are then interpreted by a virtual machine(JVM)
- A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine



JIT Compilation in Java

- In order to achieve faster processing of inputs to outputs, some Java compilers, called just-in-time compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.
- Just-In-Time (JIT) compilation is feature of Java Virtual Machine (JVM) and it is mostly used on the simple or frequently used methods in the code.

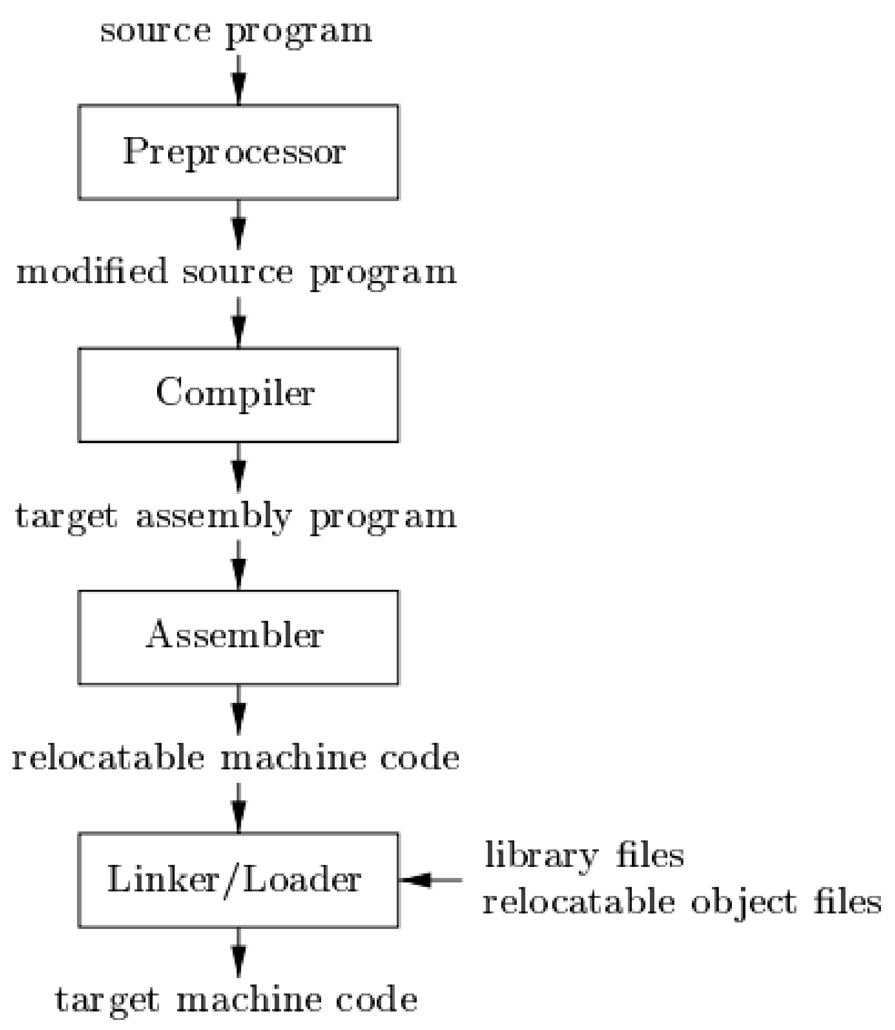
Source code to executable

- In addition to a compiler, several other programs may be required to create an executable target program.
- A source program may be divided into modules stored in separate files.
- A preprocessor collects and processes the source code before compilation, expanding macros and handling directives to prepare the code for the compiler.
- The modified source program is then fed to a compiler and it generate the assembly language most of the time.
- The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output

Source code to executable

- 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.

Source code to executable



The Structure of a Compiler

- Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program.
- If we open up this box a little, we see that there are two parts to this mapping: **analysis and synthesis**.

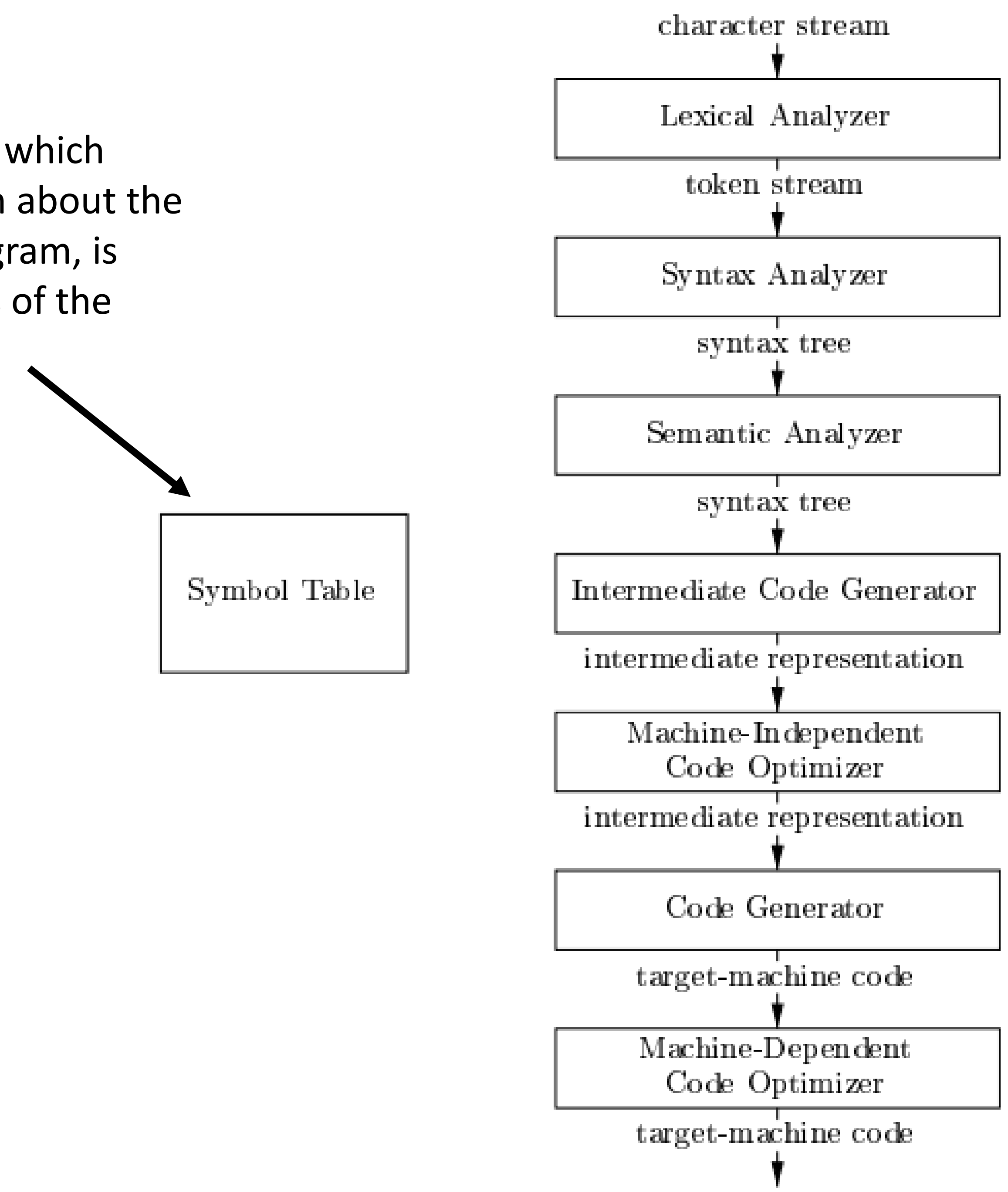
Analysis Part

- The analysis phase breaks down the source program into parts, organizes them grammatically, and creates an intermediate representation.
- It also gathers information about the program and stores it in a symbol table

Synthesis Part

- The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.
- The analysis partis often called the front end of the compiler; the synthesis part is the back end

The symbol table, which stores information about the entire source program, is used by all phases of the compiler



Lexical Analysis

- The first phase of a compiler is called **lexical analysis** or **scanning**.
- 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>** this token is passes on to the subsequent phase, syntax analysis.

Lexical Analysis (Cont.)

- 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.

Understanding with Example

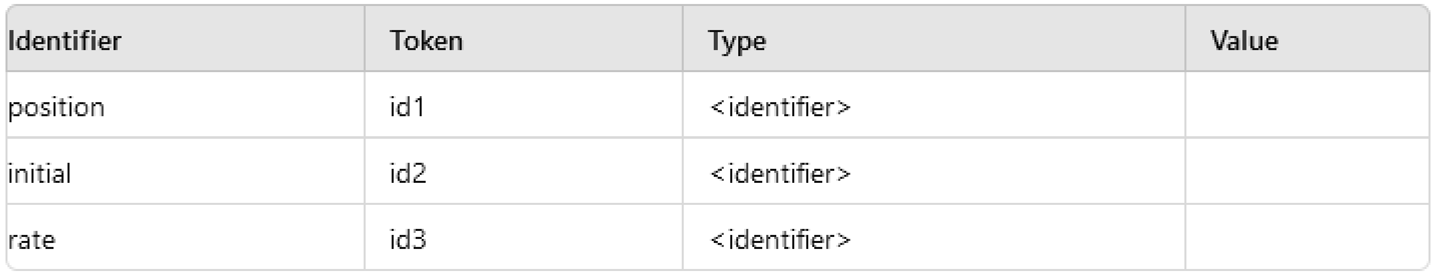
- For example, suppose a source program contains the assignment statement
  - position = initial + rate * 60

In the expression **position = initial + rate * 60**, the source code is broken into parts called lexemes, which are converted into tokens for the syntax analyzer.

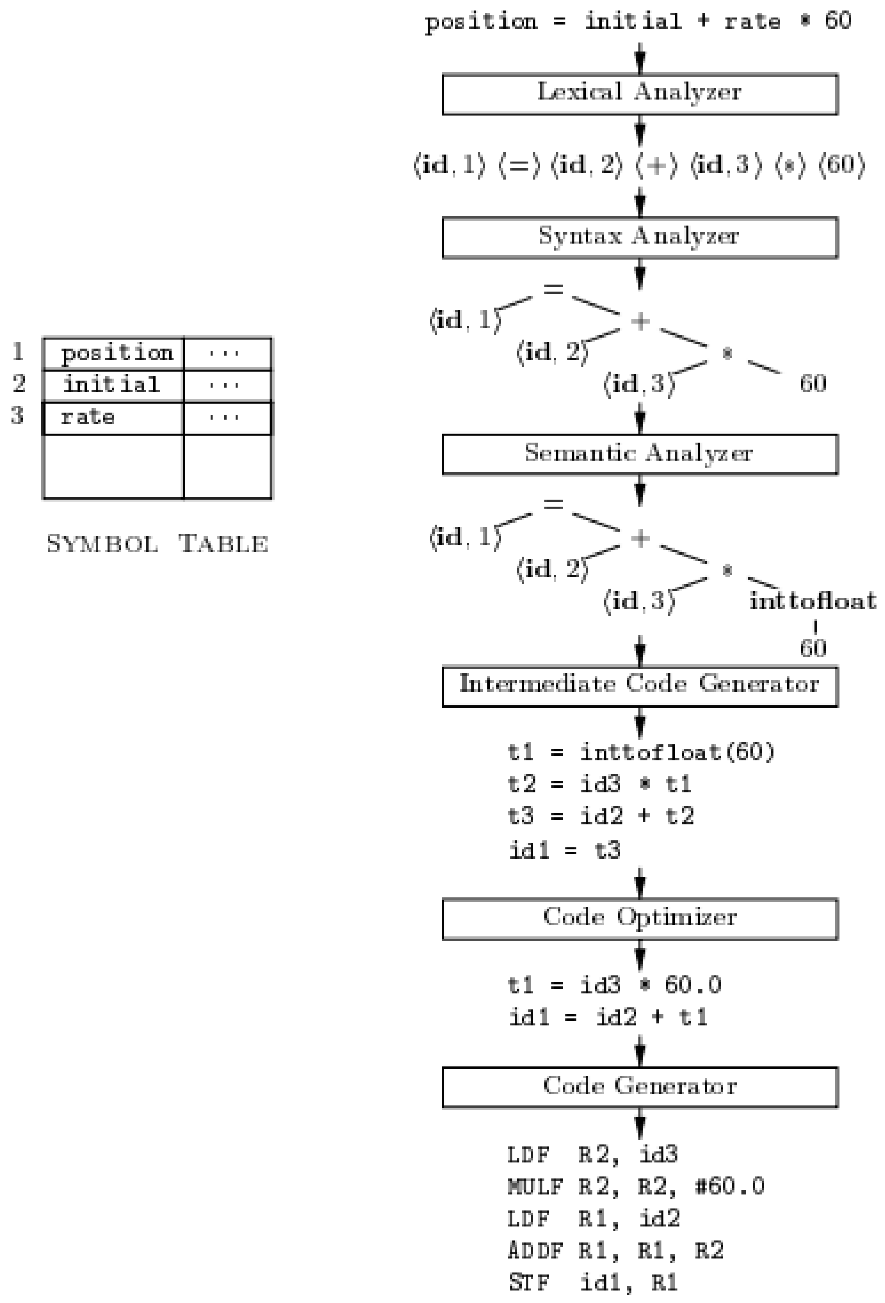
- For example, position, initial, and rate are mapped to tokens like <id, 1>, <id, 2>, and <id, 3>, with each number pointing to the symbol table entry that holds details about the identifier.
- The symbols =, +, and * are mapped directly to tokens like <=>, <+>, and <*>, and 60 is mapped to <60>.
- The lexical analyzer ignores spaces between the lexemes.

<id,1> <=> <id,2> <+> <id,3> <*> <60>

The symbol table will somehow look like this



Identifier	Token	Type	Value
position	id1	<identifier>	
initial	id2	<identifier>	
rate	id3	<identifier>	



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

SYMBOL TABLE

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

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

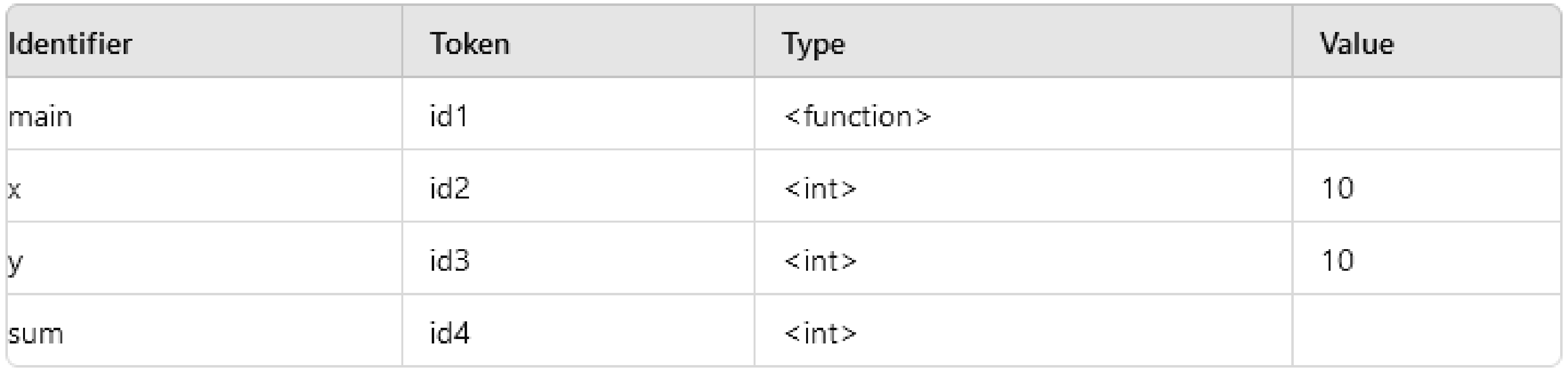
```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Class Task

- Here is code make its symbol table and also write the best possible tokens for this code

```
int main(){
    int x = 10;
    int y = 10;
    int sum = x + y;
    cout<<sum


}
```

Symbol Table



Identifier	Token	Type	Value
main	id1	<function>	
x	id2	<int>	10
y	id3	<int>	10
sum	id4	<int>	

Tokens

Token1→ int: <int>
Token2→ main: <id,1>
Token3→ (: <(>
Token4→): <)>
Token5→{:<{>
Token6→ x:<id,2>
Token7→ y:<id,3>
Token8→ sum<id,4>
Token9→ 10:<10>
Token10→ =: <=>
.
.
.

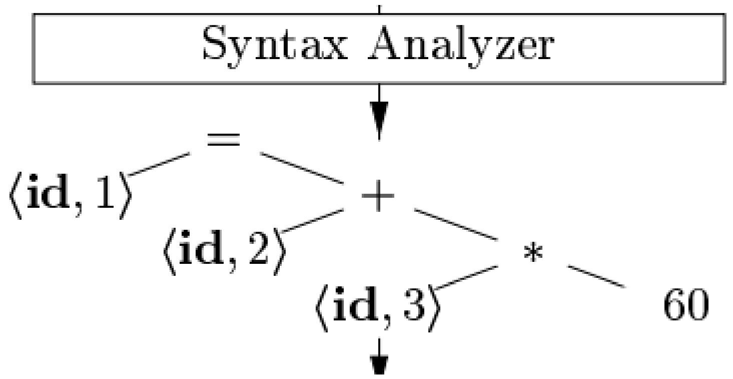
```
int main(){
        int x = 10;
        int y = 10;
        int sum = x + y;
        cout<<sum

}
```

Syntax Analysis

- The second phase of the compiler is 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.

position = initial + rate * 60

<id,1> <=> <id,2> <+> <id,3> <*> <60>



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.
- In programming languages, sometimes you need to perform operations with different types of data. For example, you might want to add an integer to a floating-point number. To make this possible, the language can automatically convert one type to another. This is also done by the compiler in semantic analysis phase.

Class Task

- In this expression how the compiler perform conversion?

position = initial + rate * 60

Intermediate Code Generation

- In the 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.

Code Optimization

- Code optimization improves the efficiency of intermediate code by refining it to reduce resource usage and execution time, while preserving the original program's functionality. This phase involves techniques like eliminating redundant operations and simplifying expressions to enhance performance and reduce the final machine code size.
- Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

Code Generation

The code generator takes as input an intermediate representation of the source program 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

Symbol-Table Management

- 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.
- The data structure should be designed to allow the compiler to and the record for each name quickly and to store or retrieve data from that record quickly

The Evolution of Programming Languages

- The first electronic computers appeared in the 1940's and were programmed in machine language by sequences of 0's and 1's that explicitly told the computer what operations to execute and in what order.
- The operations themselves were very low level: move data from one location to another, add the contents of two registers, compare two values, and so on.

The Move to Higher-level Languages

- The first step towards more people-friendly programming languages was the development of mnemonic assembly languages in the early 1950’s.
- Initially, the instructions in an assembly language were just mnemonic representations of machine instructions.
- Later, macro instructions were added to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions

The Move to Higher-level Languages

- A major step towards higher-level languages was made in the latter half of the 1950's with the development of Fortran for scientificcomputation.
- Cobol for business data processing, and Lisp for symbolic computation
- The philosophy behind these languages was to create higher-level notations with which programmers could more easily write numerical computations, business applications, and symbolic programs

Classification of Languages

- **First-Generation Languages (1GL) → Machine Language**
  - Directly understood by the computer's CPU.
  - Consists of binary code (0s and 1s).
  - Very low-level, hardware-specific instructions.
- **Second-Generation Languages (2GL) → Assembly Languages**
  - Mnemonic codes representing machine instructions.
  - Each assembly instruction corresponds to a specific machine code instruction.
  - Slightly higher-level than machine language, but still closely tied to hardware.

Classification of Languages

- **Third-Generation Languages (3GL) → High level languages**
  - More abstract and closer to human language.
  - Examples include Fortran (scientific computation), Cobol (business data processing), Lisp (symbolic computation), C, C++, C#, and Java.
  - Portable across different hardware platforms, requiring a compiler or interpreter to translate into machine code.

- **Fourth-Generation Languages (4GL) → Application Specific**
  - Designed for specific tasks or domains.
  - SQL: Database queries and management.
  - Postscript: Text and graphics formatting

Classification of Languages

- **Fifth-Generation Languages (5GL) → Logic- and Constraint-Based Languages**
  - Focus on solving problems using constraints and logical inference.
  - Often used in AI and expert systems, emphasizing "what" to solve rather than "how" to solve it.
  - For example, In Prolog, you define relationships and rules, and the system uses these to answer queries or solve problem