Compilers

Introduction

Dr. Akar Taher

Outlines

- Introduction to Compilers
- Why Compilers
- Program Language Classification
- History of Compilers
- Language Processors
- Lexical analysis (Scanning)
- Syntax Analysis (Parsing)
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generation

Why compilers?

- Increase productivity
  - ✓ Low level programming languages are harder to write , less portable , error-prone area, harder to maintain.
- Reverse engineering eg. (Executble applications .exe)
  - ✓ runtime (0100101100)→assembly code→high level language.

Programming Language Classification

- Low-level language
  - Assembly language
  - Machine language
- High-level language
  - C, C++, java, Pascal, Prolog, Scheme
- Natural language
  - English, Kurdish.

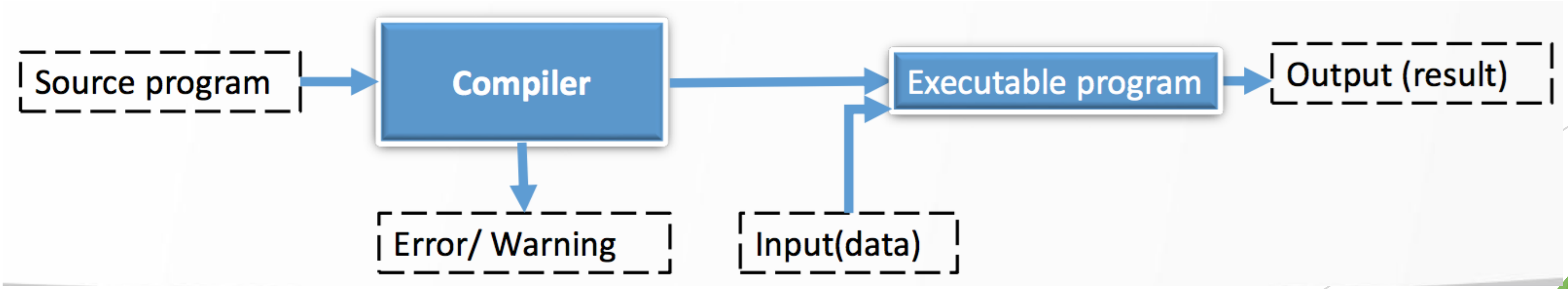
- There is also classification by programming language generations.

History of Compilers

- Grace Hopper, in 1952, A-0 System.
- Alick Glennie in 1952, Autocode.
- John W. Backus in 1953, Speedcoding
  - More productive , but 10-20 slower in execution.
  - took 300 bytes of memory (30% memory)
- Fortran, in 1954-1957 first complete compiler.
- In 1960 Cobol, lisp.
- In 1970 Pascal , C.
- In 1980 OOP Ada , smalltalk , C++.
- In 1990 Java , script , Perl.
- In 2000 language specifications.

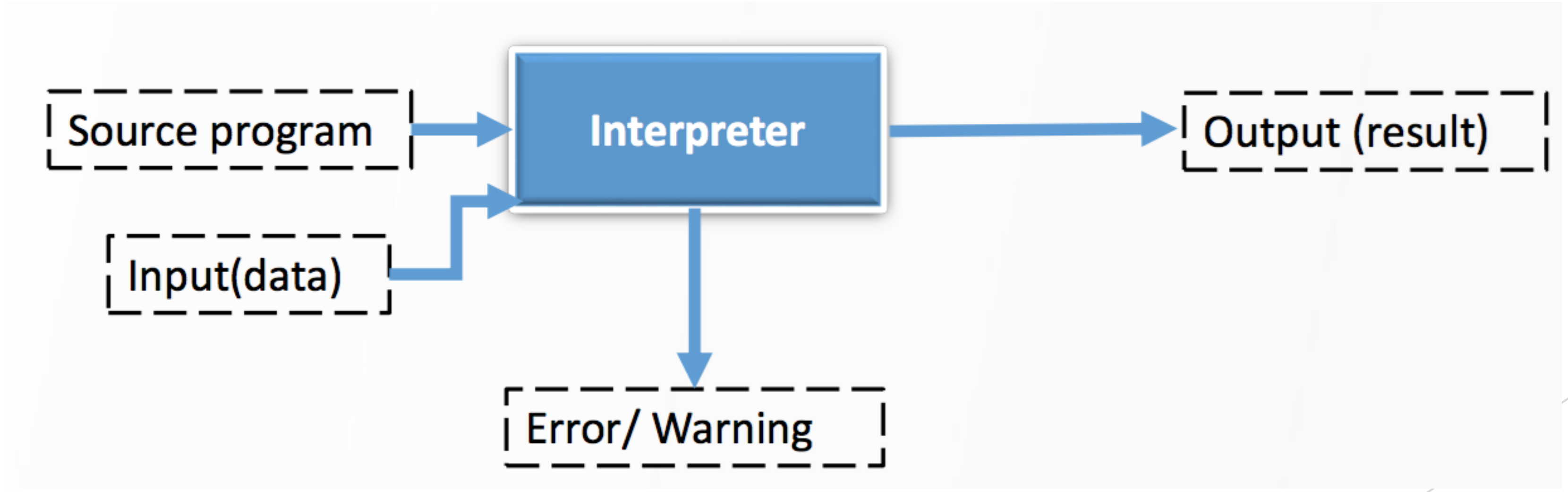
Language Processors (Compilers Definition)

- A compiler translates the code written in one language (HLL) to some other language (LLL) without changing the meaning of the program.
- It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.
- Compiler design covers basic **translation mechanism** and error **detection & recovery**.
- Examples: Fortran , Ada, C, C++ , C# , Cobol.

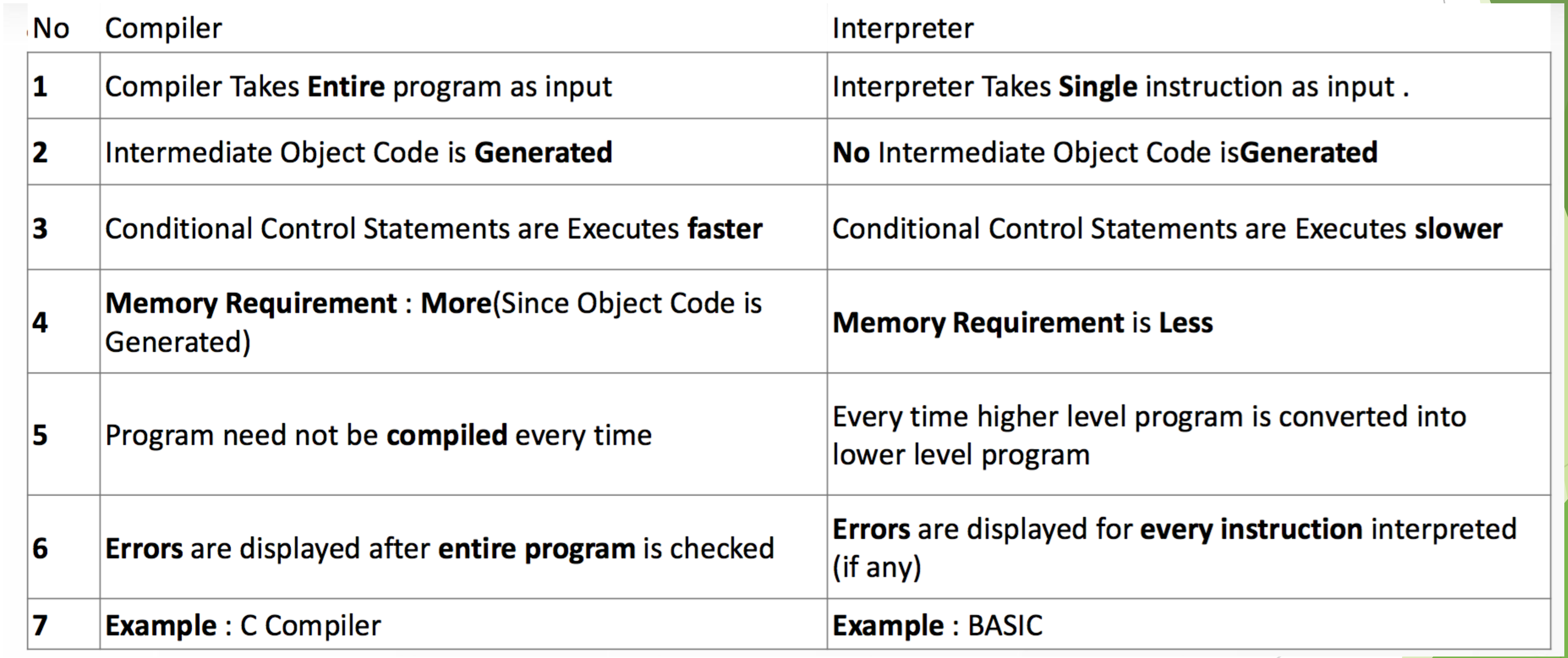


Language Processors (Interpreters Definition)

- Interpreter is a type of language processor that directly executes the operations specified in the source program on inputs supplied by the user.
- Example: Python , Perl , Basic , Ruby, AWK.

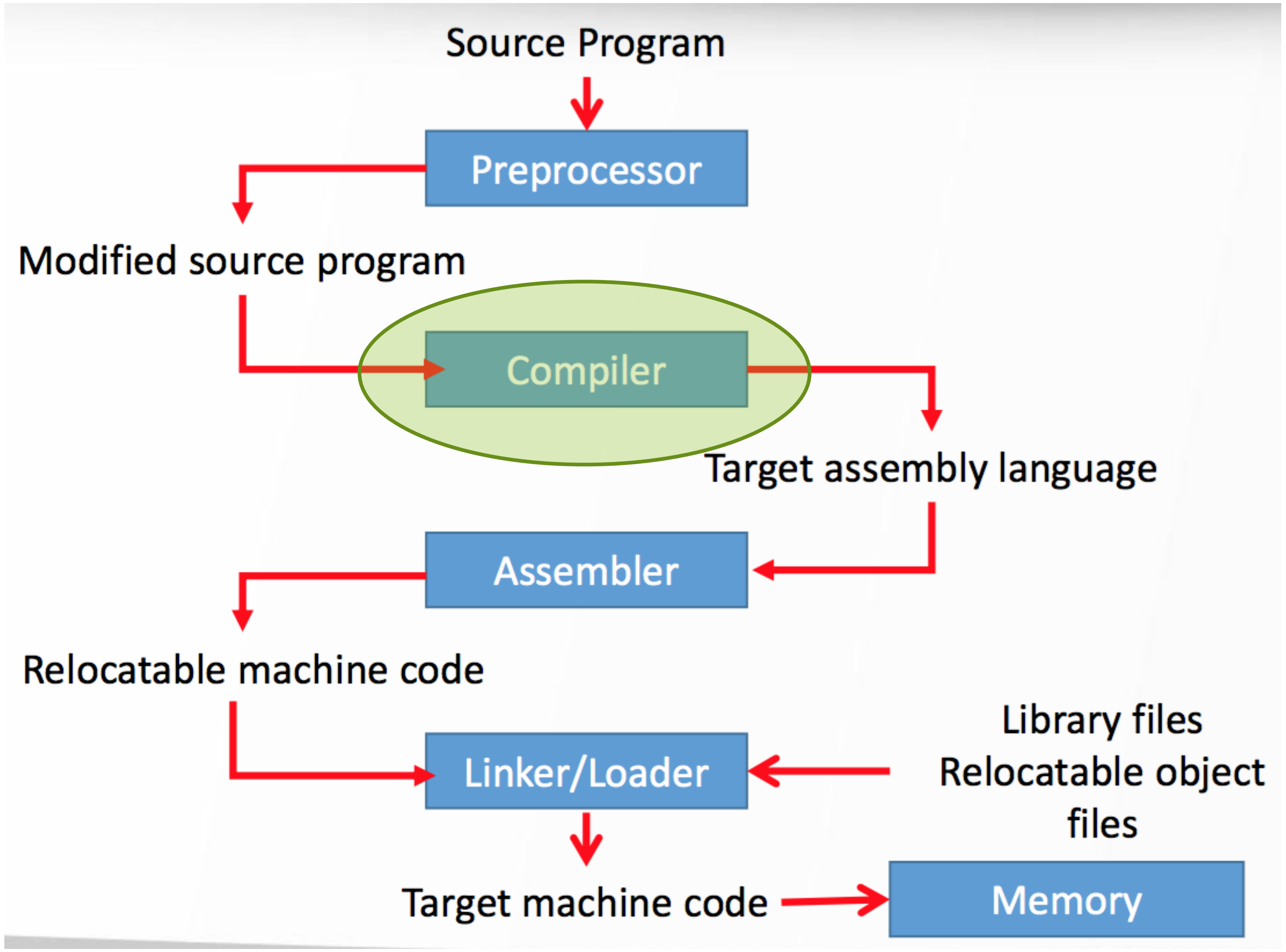


Language Processor (Compilers VS. Interpreters)



No	Compiler	Interpreter
1	Compiler Takes **Entire** program as input	Interpreter Takes **Single** instruction as input .
2	Intermediate Object Code is **Generated**	**No** Intermediate Object Code is**Generated**
3	Conditional Control Statements are Executes **faster**	Conditional Control Statements are Executes **slower**
4	**Memory Requirement** : **More**(Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need not be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example** : C Compiler	**Example** : BASIC

Language Processor (Phases)



Language Processor (Phases)

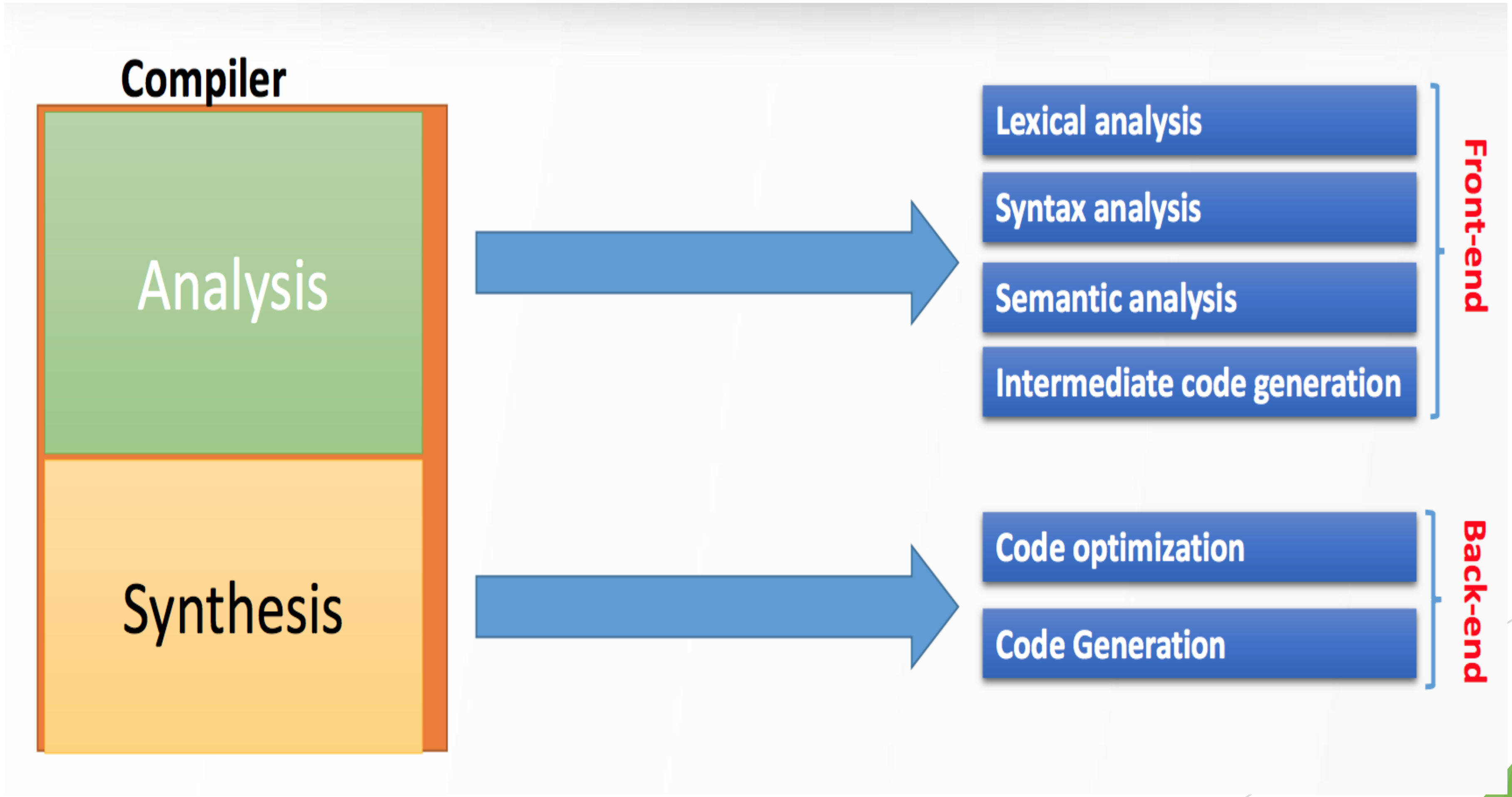
Let us first understand how a program, using for example C compiler, is executed on a host machine.

- User writes a program in C language (high-level language).
- The C compiler, compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).
- A loader loads all of them into memory and then the program is executed.

Language Processor (Phases cont’d)

- **Preprocessor:** is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.
- **Assembler:** it translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.
- **Linker:** is a computer program that links and merges various object files together in order to make an executable file.
- Loader is a part of operating system and is responsible for loading executable files into memory and execute them.

Language Processing (Compiler Phases)



Analysis VS Synthesis

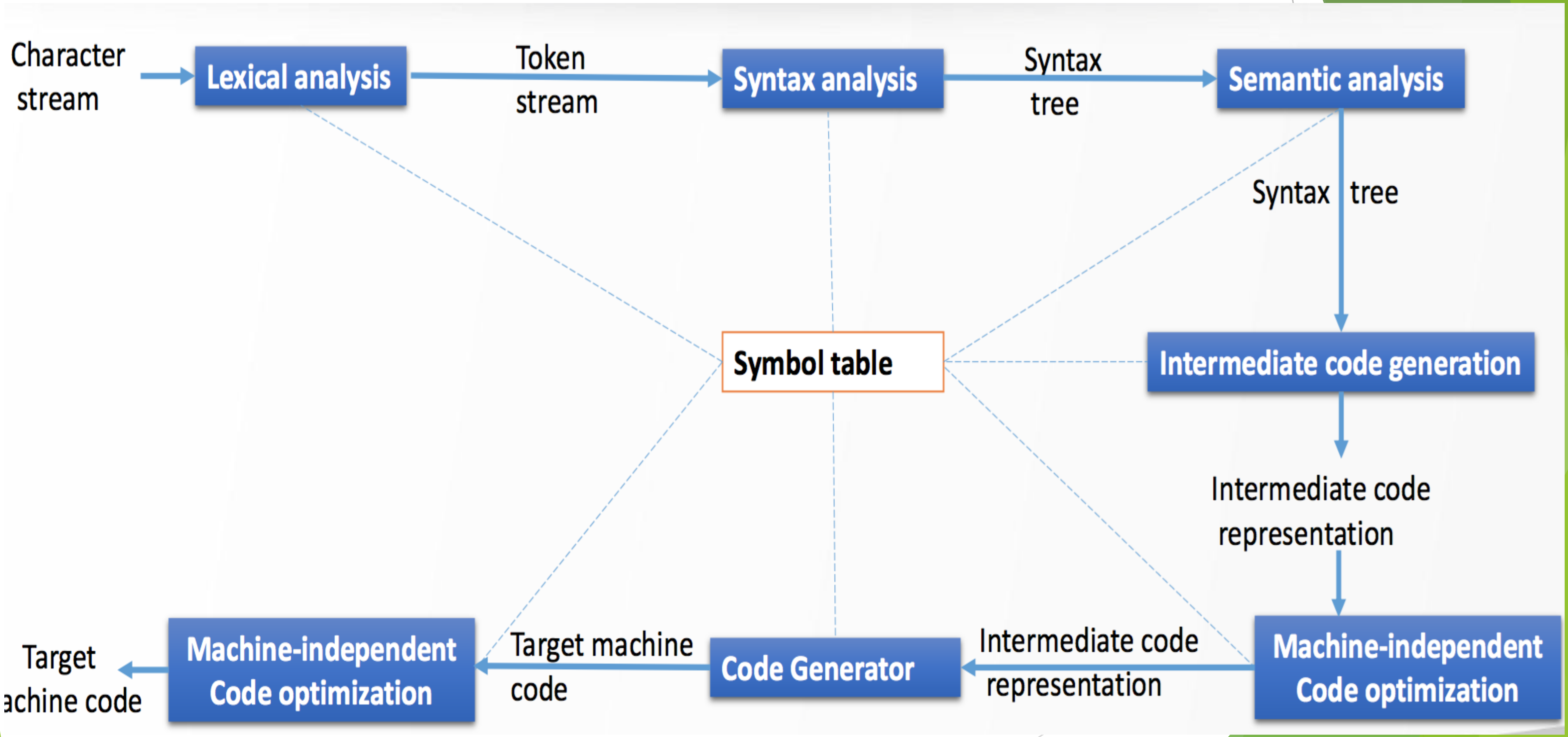
Analysis part

- Breaks up the source program into constituent pieces and create an intermediate representation of the source program. It is often called the front end of the compiler.
- The analysis part can be divided along the following phases: Lexical analysis , syntax analysis , semantic analysis and intermediate code generation (optional)

Synthesis part

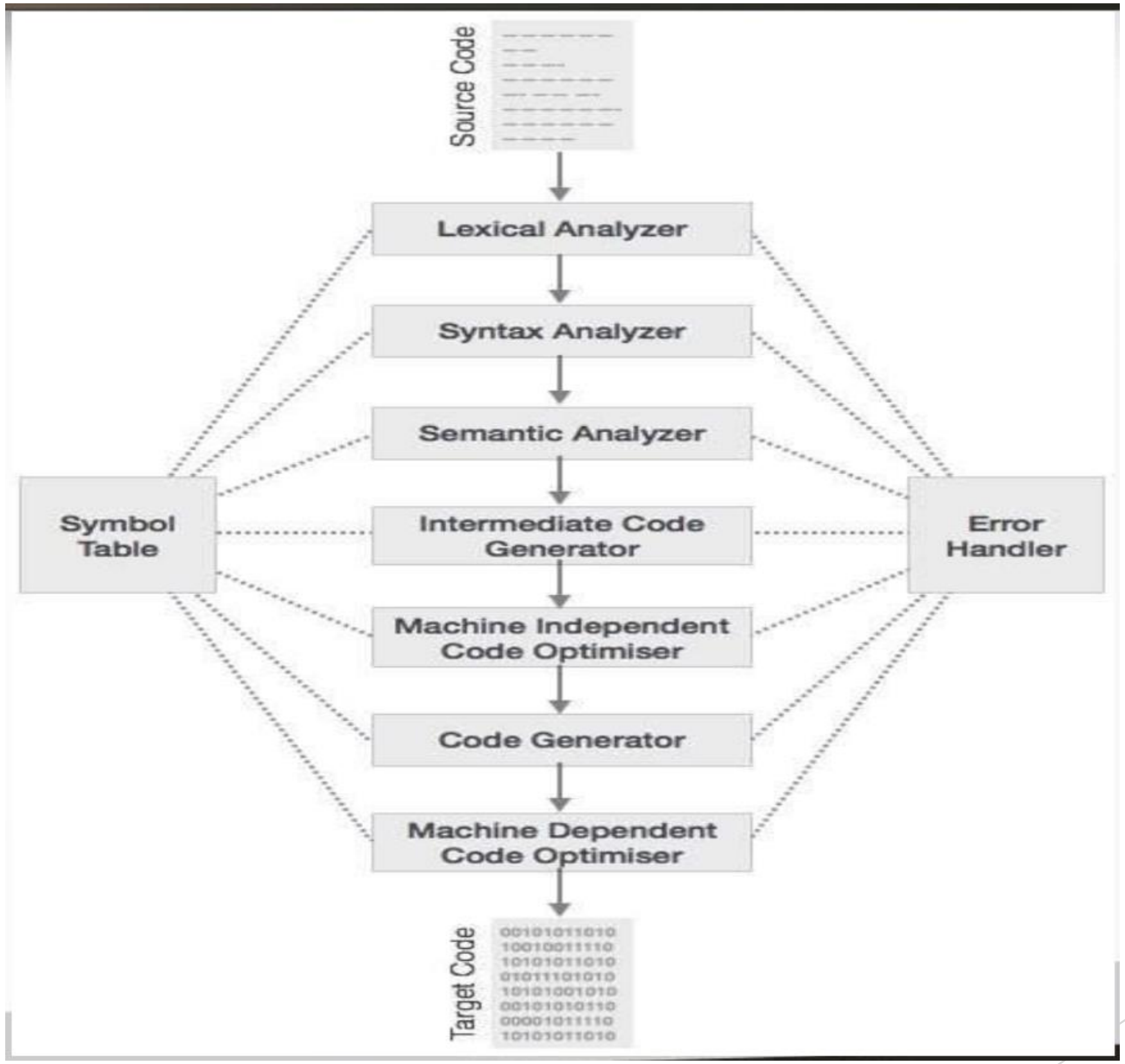
- Construct the desired target program from the intermediate representation and the information of the symbol table. It is often called the back end.
- The Synthesis part can be divided along the following phases: Intermediate code generator, code optimizer, code generator

Compiler Phases Diagram



Character stream
Lexical analysis
Token stream
Syntax analysis
Syntax tree
Semantic analysis
Syntax tree
Symbol table
Intermediate code generation
Intermediate code representation
Machine-independent Code optimization
Intermediate code representation
Code Generator
Target machine code
Machine-independent Code optimization
Target achine code

Compiler Phases Diagram (Cont’d)



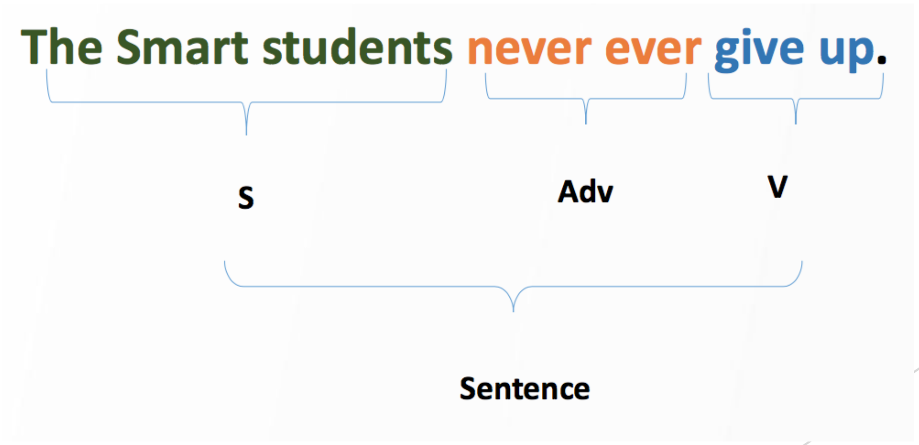
Lexical Analysis (Scanning)

- Lexical analysis is the process of converting a sequence of characters (such as a computer program or web page) into a sequence of tokens (strings with an identified "meaning").
- Lexical divides program text into “words” or “tokens”.
- How do we understand English ?
  Iamasmartstudent. I am a smart student.
- Break up sentence and recognize words.

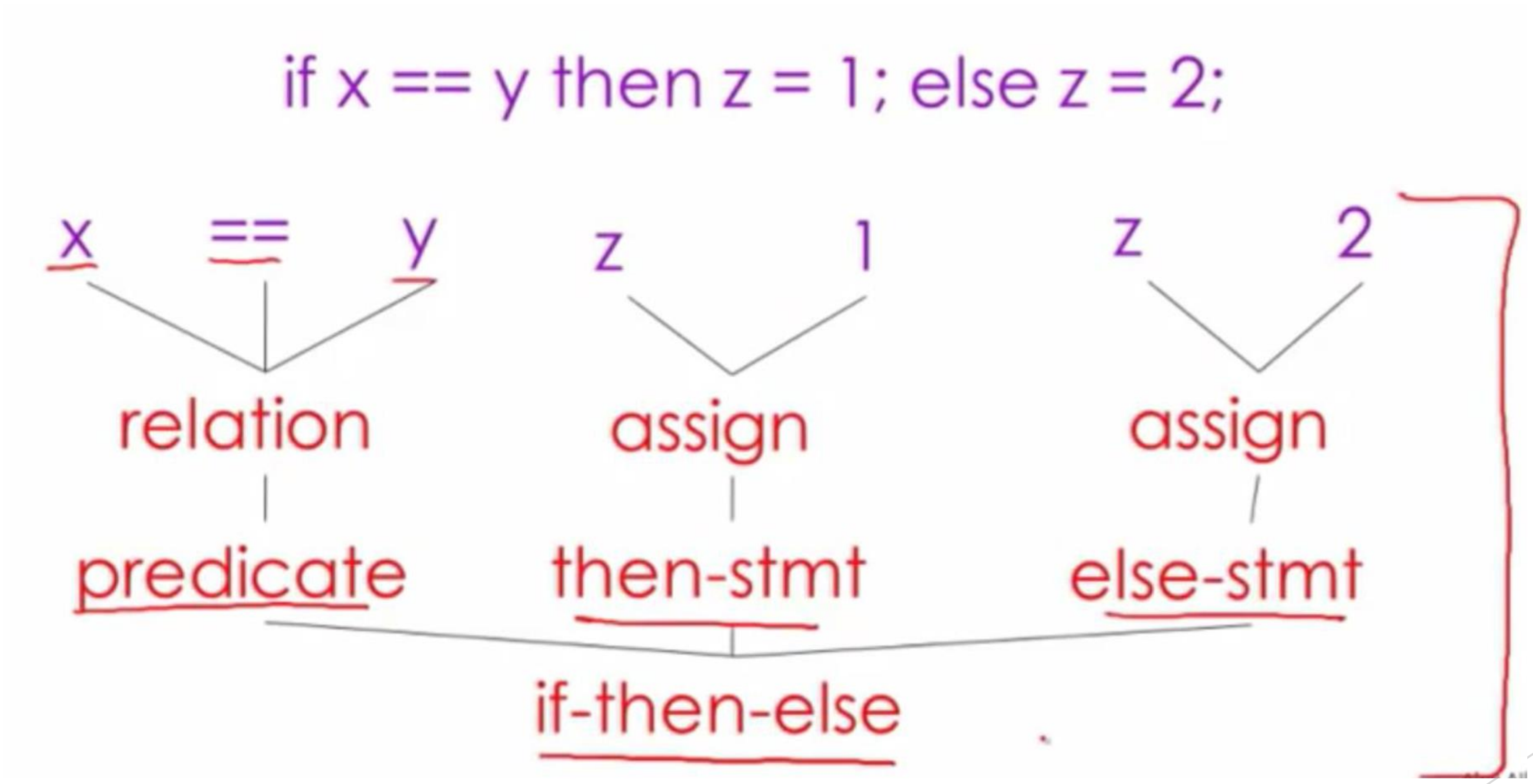
If X==Y Then Z=1; Else Z=2;

Syntax Analysis (Parsing)

- A Parser reads a stream of tokens from scanner, and determines if the syntax of the program is correct according to the context-free grammar (CFG) of the source language.
- Then, Tokens are grouped into grammatical phrases represented by a Parse Tree or an abstract syntax tree, which gives a hierarchical structure to the source program.
- How do we understand English?

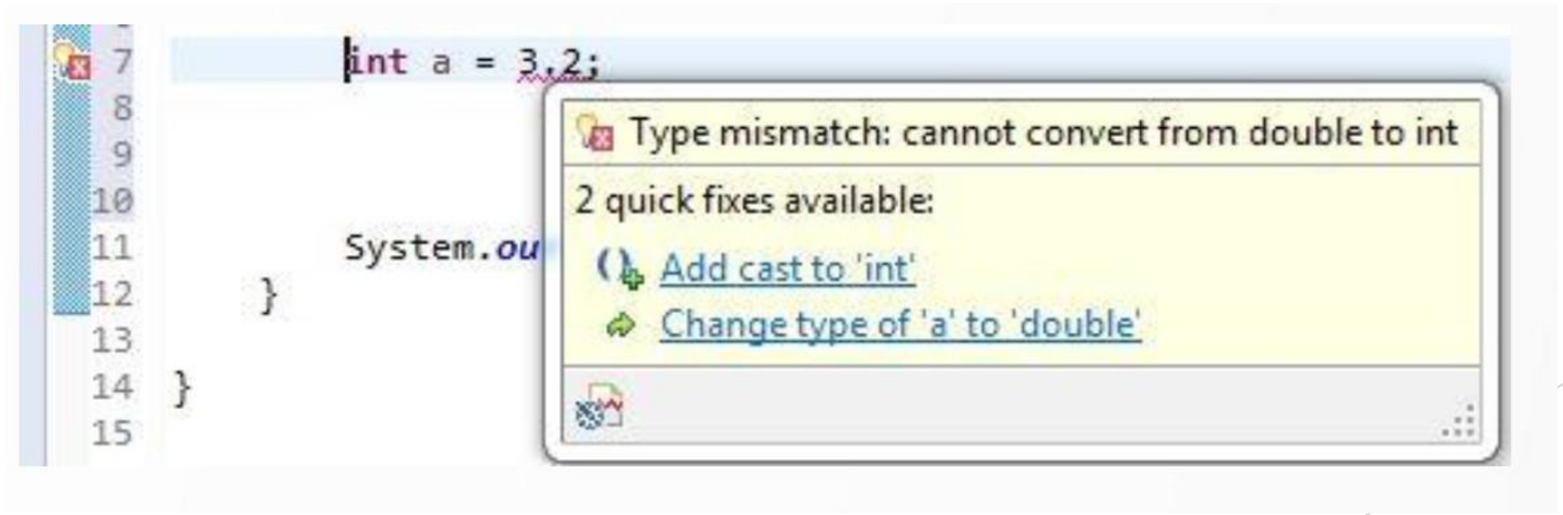


Syntax Analysis (Parsing) (Cont'd)



Semantic Analysis

- The Semantic Analysis phase checks the (meaning of) source program for semantic errors (Type Checking) and gathers type information for the successive phases.
- Semantic analysis is the heart of compiler. Also, type checking is the important part in this phase. Check language requirements like proper declarations.
  Semantic analysis catches inconsistencies for instance mismatching datatypes.



```
        int a = 3.2;


        System.ou
    }

}
```

Type mismatch: cannot convert from double to int

2 quick fixes available:

- Add cast to 'int'
- Change type of 'a' to 'double'

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

- This phase attempts to improve the intermediate code, which is produced. So that faster-running machine code can be achieved in the term of time and space.
- The optimized code MUST be CORRECT
- Run Faster (time)
- Minimize power consumption(Mobile devices)
- Use less memory
- Shorter is better
- Consider network, database access.

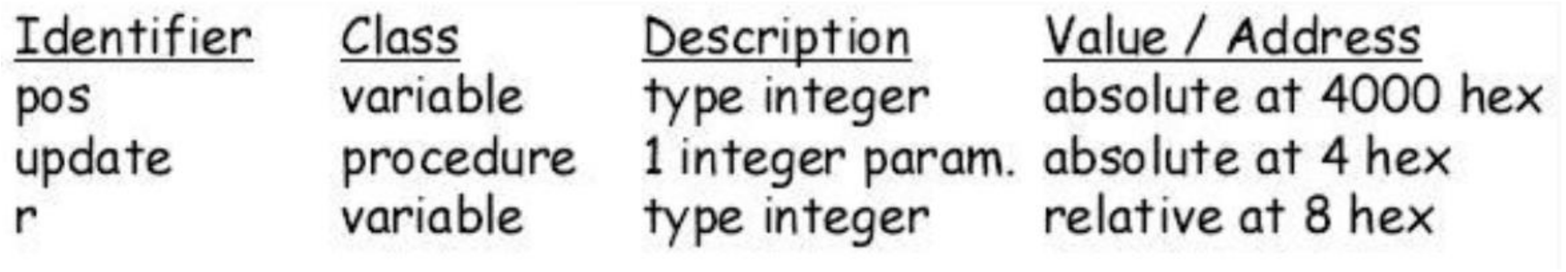
$X = Y * 0$ is the same as $X = 0$

Code Generation

- The code generator takes as input an intermediate code 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.

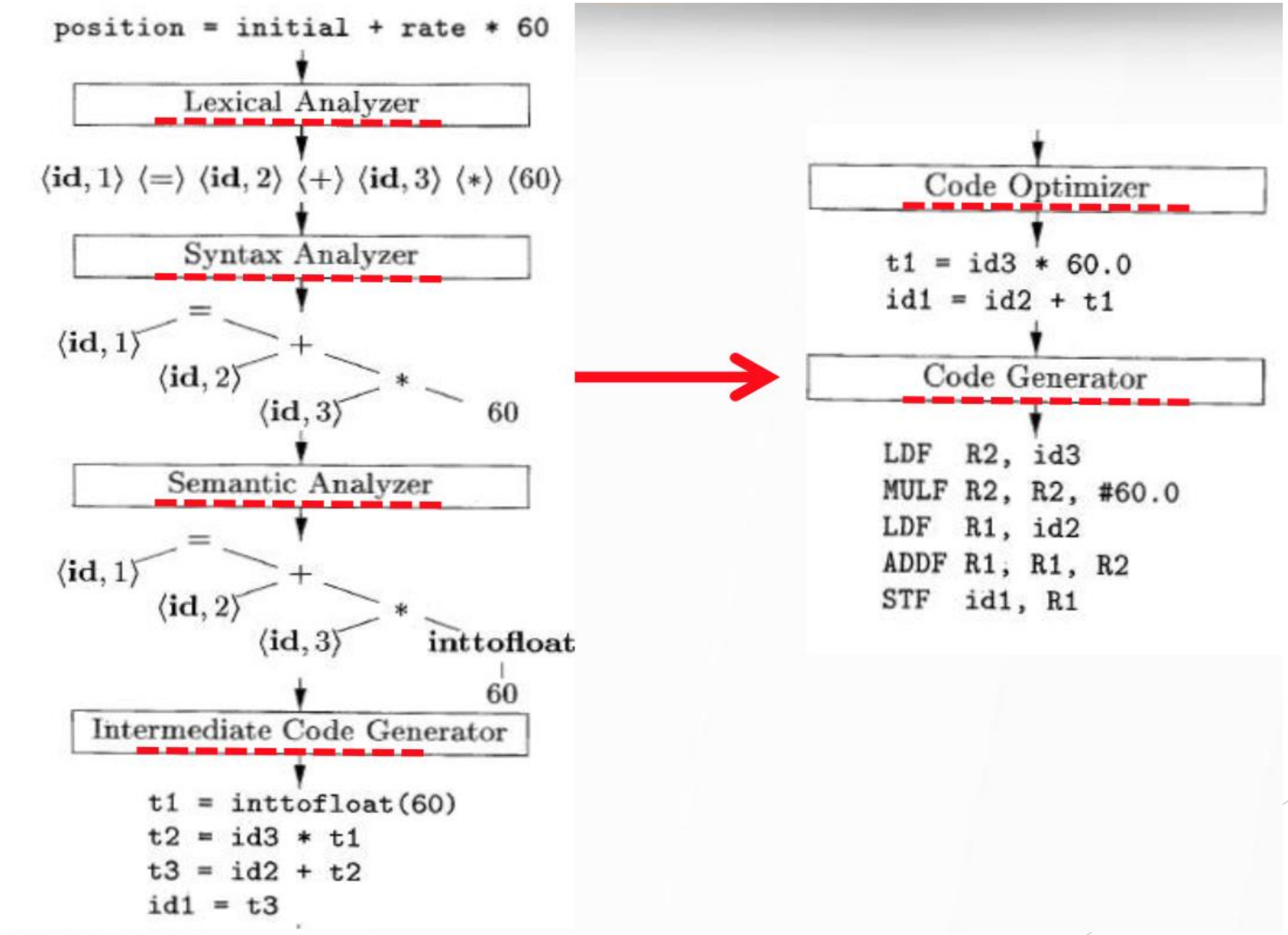
Symbol Table

- A symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.
- Symbol table should allow find the record for each name quickly and to store and retrieve data from that record quickly.
- Attributes may provide information about storage allocation, type, scope, number and type of arguments, method of passing, the type returned.



Identifier	Class	Description	Value / Address
pos	variable	type integer	absolute at 4000 hex
update	procedure	1 integer param.	absolute at 4 hex
r	variable	type integer	relative at 8 hex

All Phases Example



```
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
```

Compiler Phases (Term Definition)

- **Lexical Analysis:** The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as: `<token-name, attribute-value>`
- **Syntax Analysis:** The next phase is called the syntax analysis or **parsing**. 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.

Compiler Phases (Term Definition Cont’d)

- **Semantic Analysis:** Semantic analysis 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.

Compiler Phases (Term Definition Cont’d)

- **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.

Compiler Phases (Term Definition Cont’d)

- **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.

Thank You
Any Questions?