Lecture 01

Compilers Overview

Hyosu Kim

School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://hcslab.cau.ac.kr

hskimhello@cau.ac.kr, hskim.hello@gmail.com

The move to higher-level languages

The evolution of programming languages

Machine languages

Machine instructions (= the patterns of 0's and 1's)

Assembly languages

ADD t1, t2

Higher-level languages

C
C++
Java
Python
…

Towards more people-friendly languages

Easier-to-develop!!

Using high-level languages is a free lunch?

No

More human-friendly = Less computer-friendly

(even non-executable by computers)

Q. How can a program written in some high-level language be executed by computers?

Language translation (additional process) is required

(from high-level languages to machine languages)

The major role of language processors

1. **Language translation**

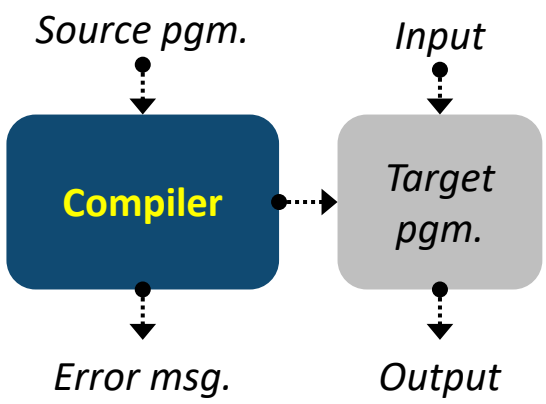
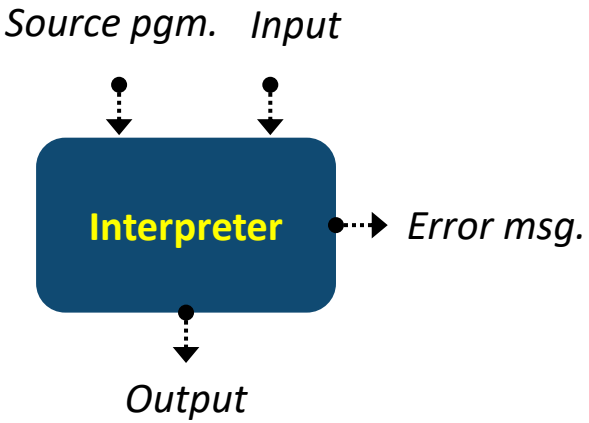
   translates **source** code (e.g., C, C++, java, python, …)

   into **semantically-equivalent target** code (e.g., assembly / machine languages)

2. **Error detection**

   detects and reports any errors in the source program during the translation process

Two representative strategies

	Compilation	Interpretation
What to translate	An entire source program	One statement of a source program
When to translate	Once before the program runs	Every time when the statement is executed
Translation result	A target program (equivalent to the source program)	Target code (equivalent to the statement)
Examples	C, C++	Javascript
		

Two representative strategies

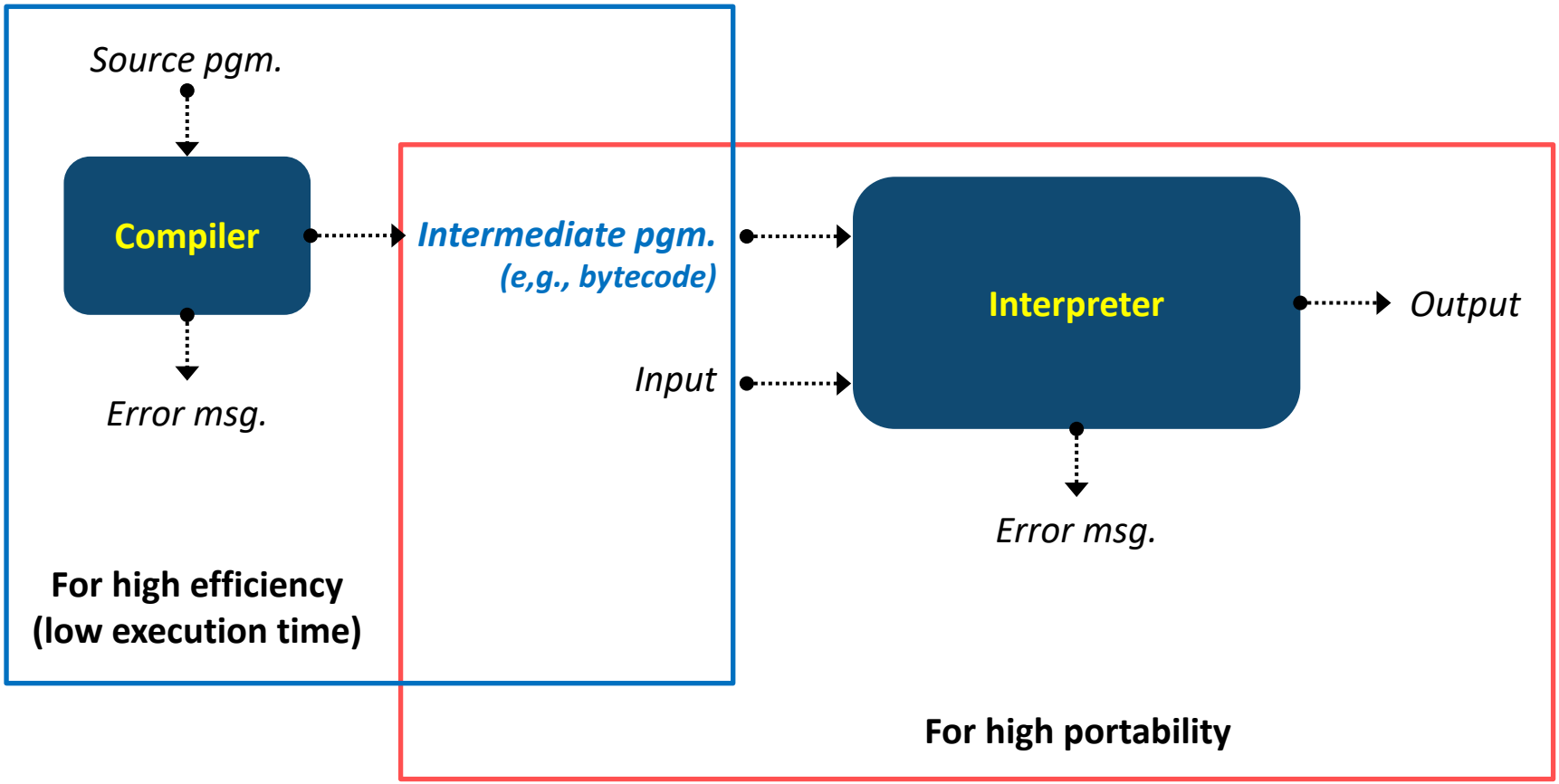
Pros / cons

	Compilation	Interpretation
Runtime performance (execution time)		
Portability / flexibility		
Debugging / development		

Two representative strategies

Variation: Hybrid compilers

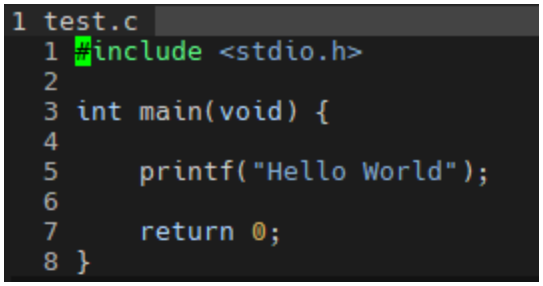
combine compilation and interpretation (e.g., Java, Python)



Common language-processing systems

e.g., in GCC (GNU C Compilers)

Source program ..... *test.c*

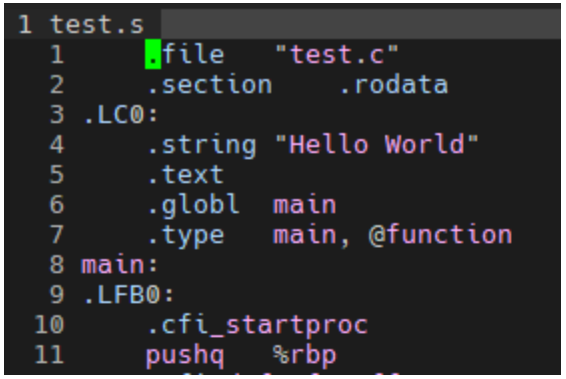


Preprocessor

Modified source program

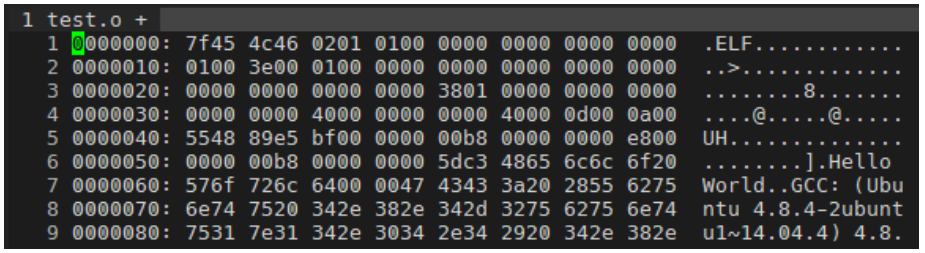
Compiler

Target assembly program ..... *test.s*



Assembler

Relocatable machine code ..... *test.o*

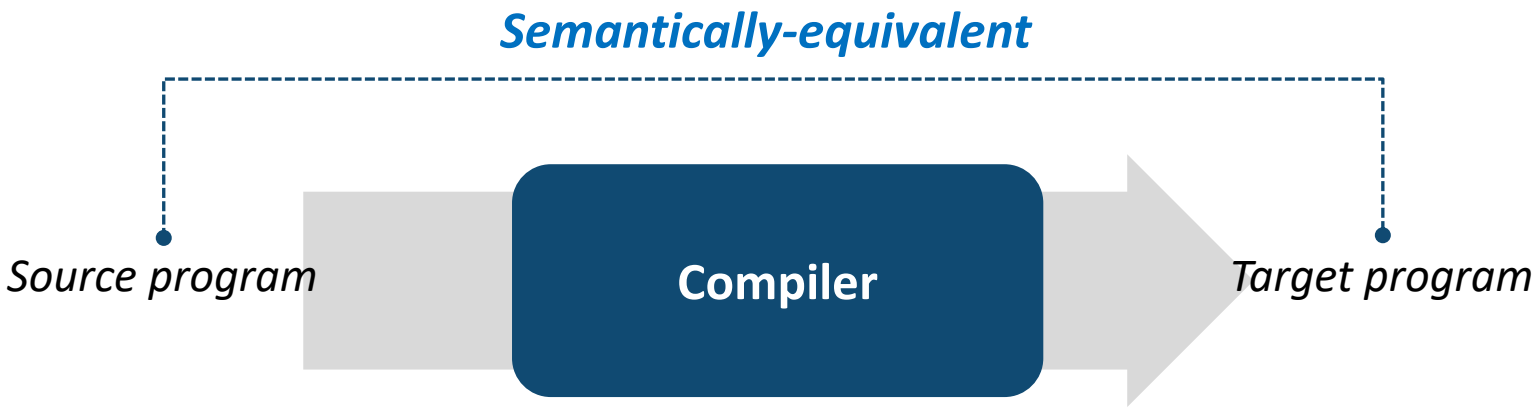


Linker ⟵ *Library files, relocatable object files*

Absolute machine code (executable binary file) ..... *test.out*

Requirements for designing good compilers

1. **Correctness (mandatory)**



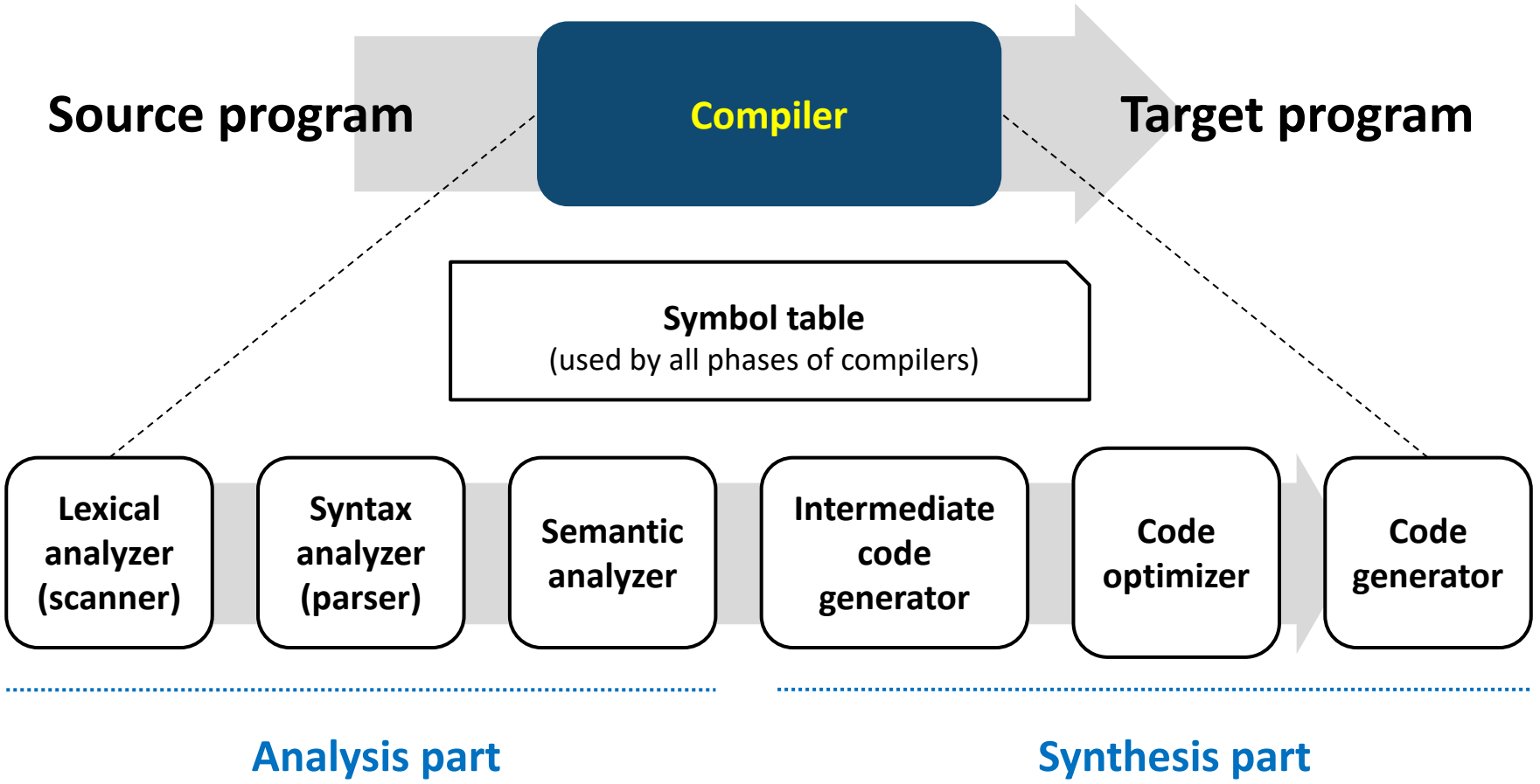
2. **Performance improvement (optional)**

3. **Reasonable compilation time (optional)**

Structure of modern compilers

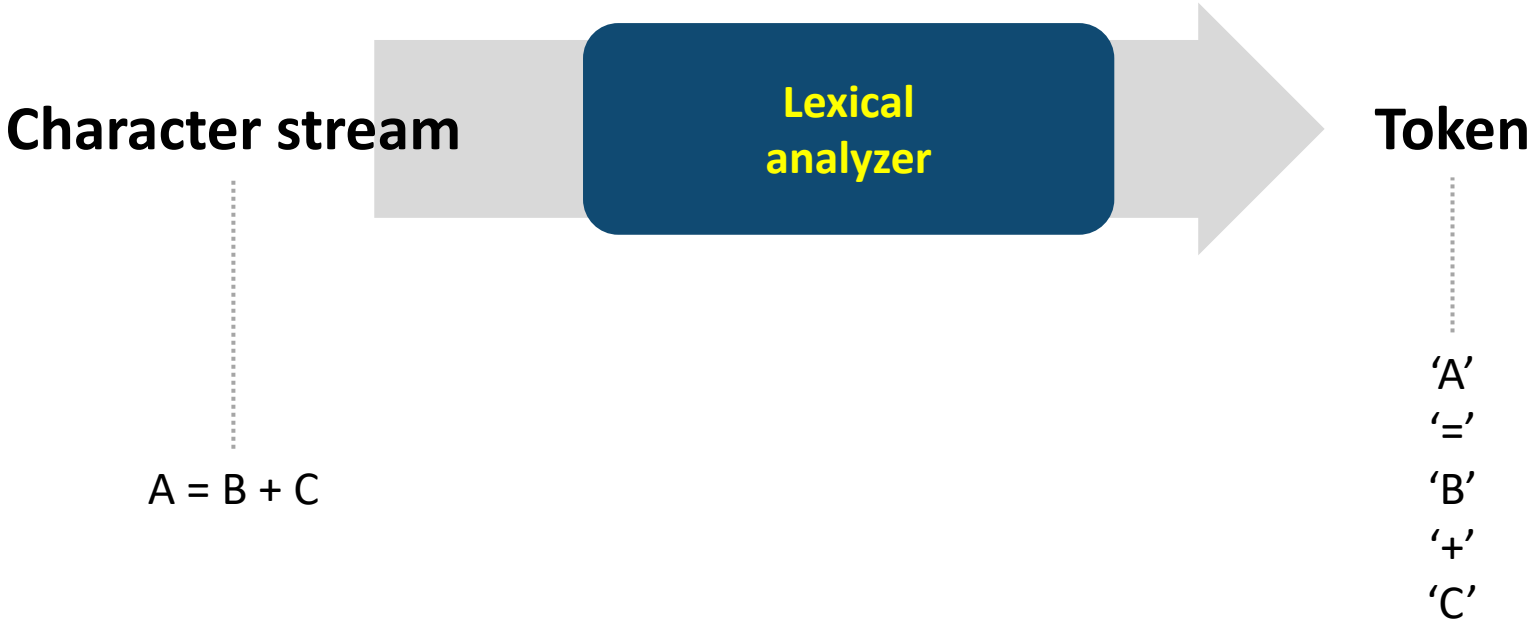
Modern compilers preserve the outlines of the FORTRAN I compiler

(the first compiler, in the late 1950s)



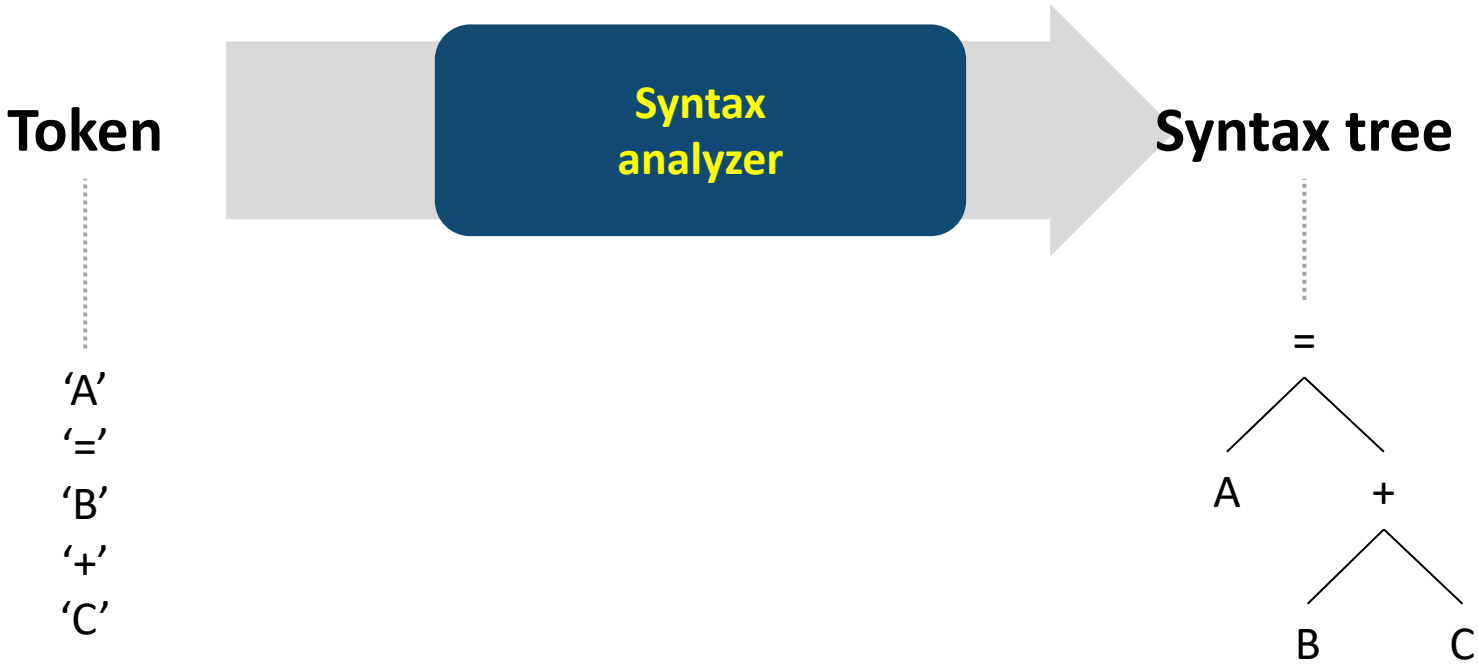
Lexical analyzer (scanner)

Divides the stream of characters into meaningful sequences and produces a set of tokens



Syntax analyzer (parser)

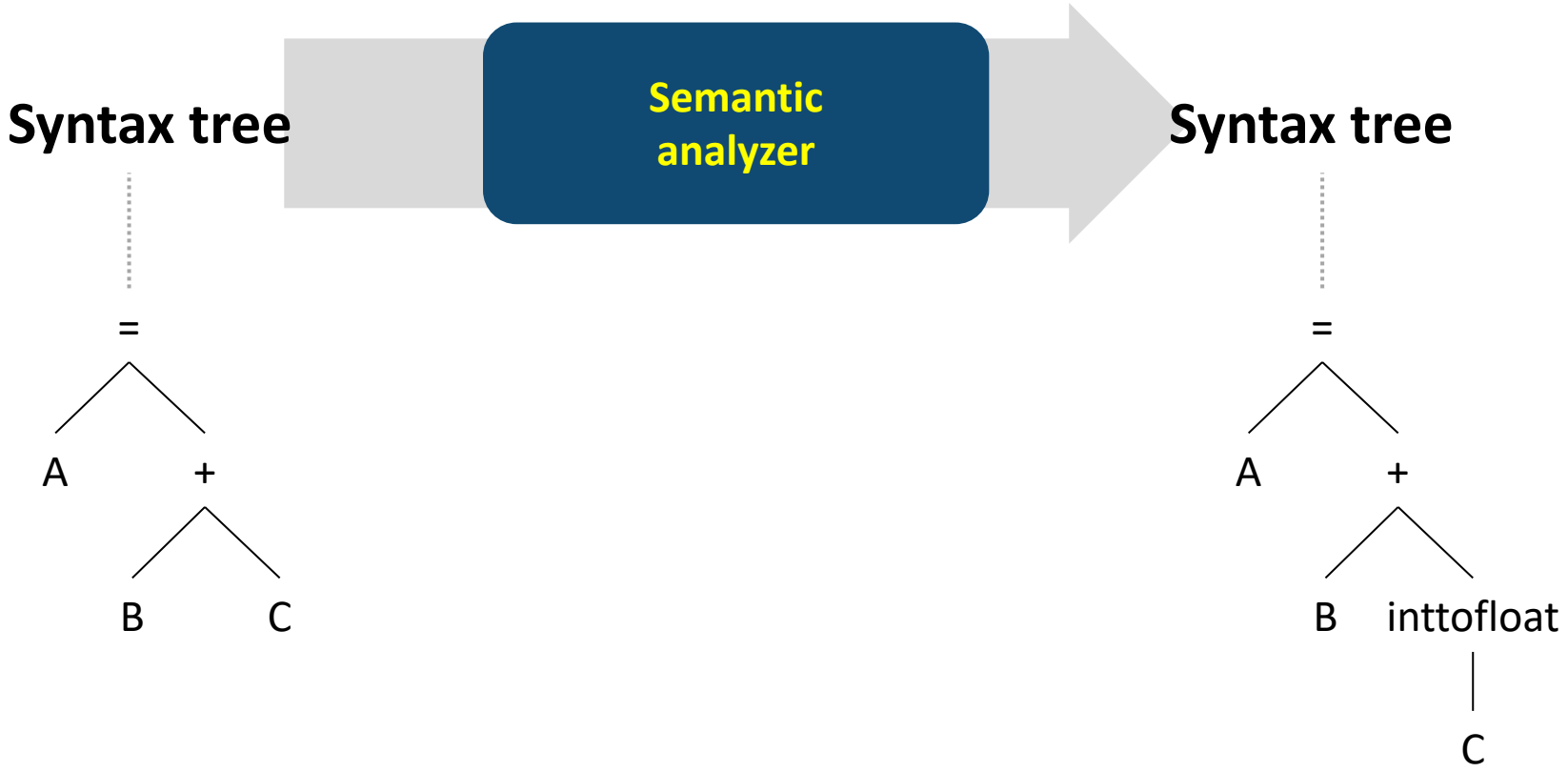
Creates a tree-like intermediate representation (e.g., syntax tree) that depicts the grammatical structure of the token stream



Semantic analyzer

Checks the source program for semantic consistency with the language definition

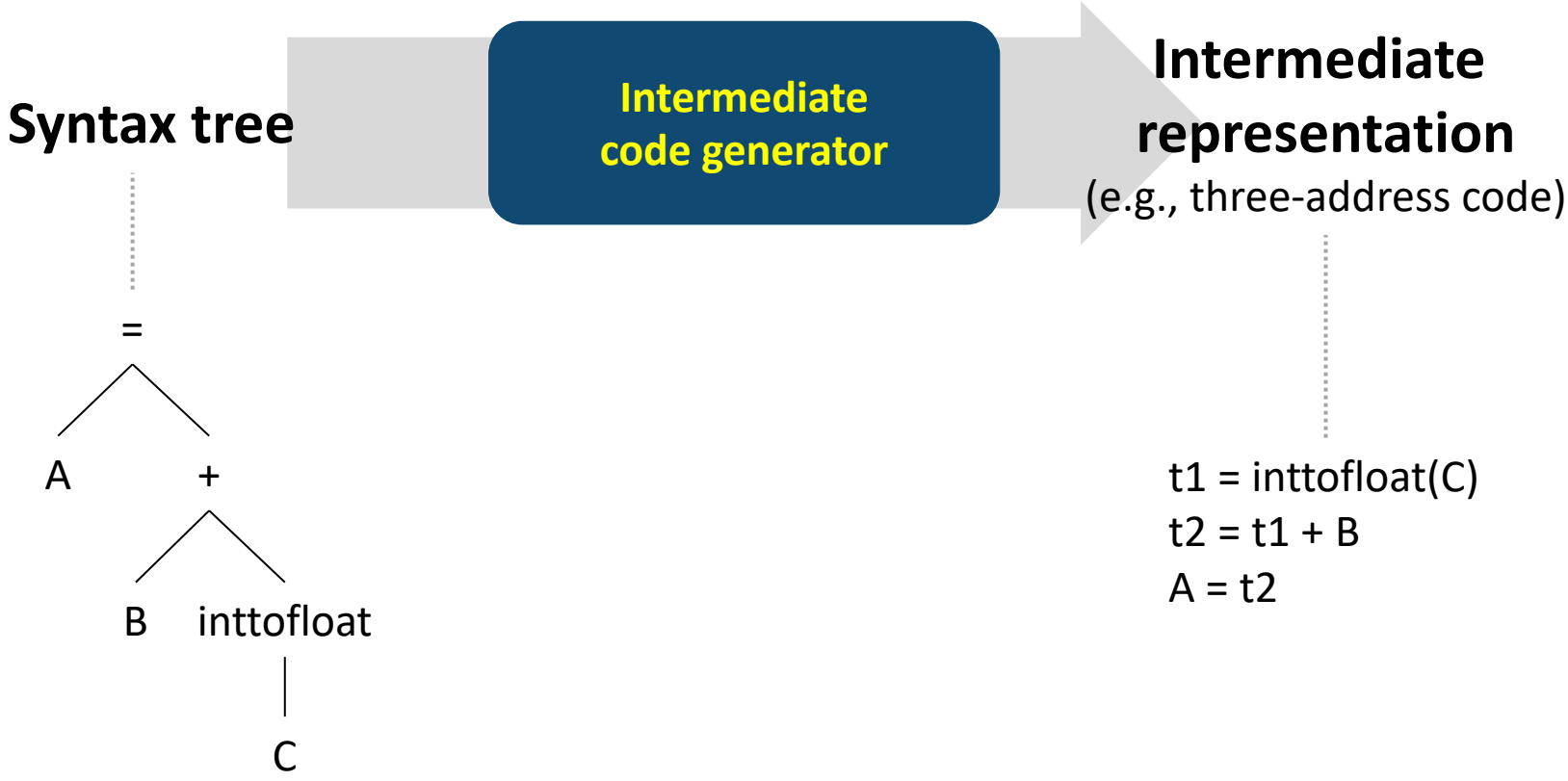
(e.g., type checking / conversion)



Intermediate code generator

Constructs intermediate representations

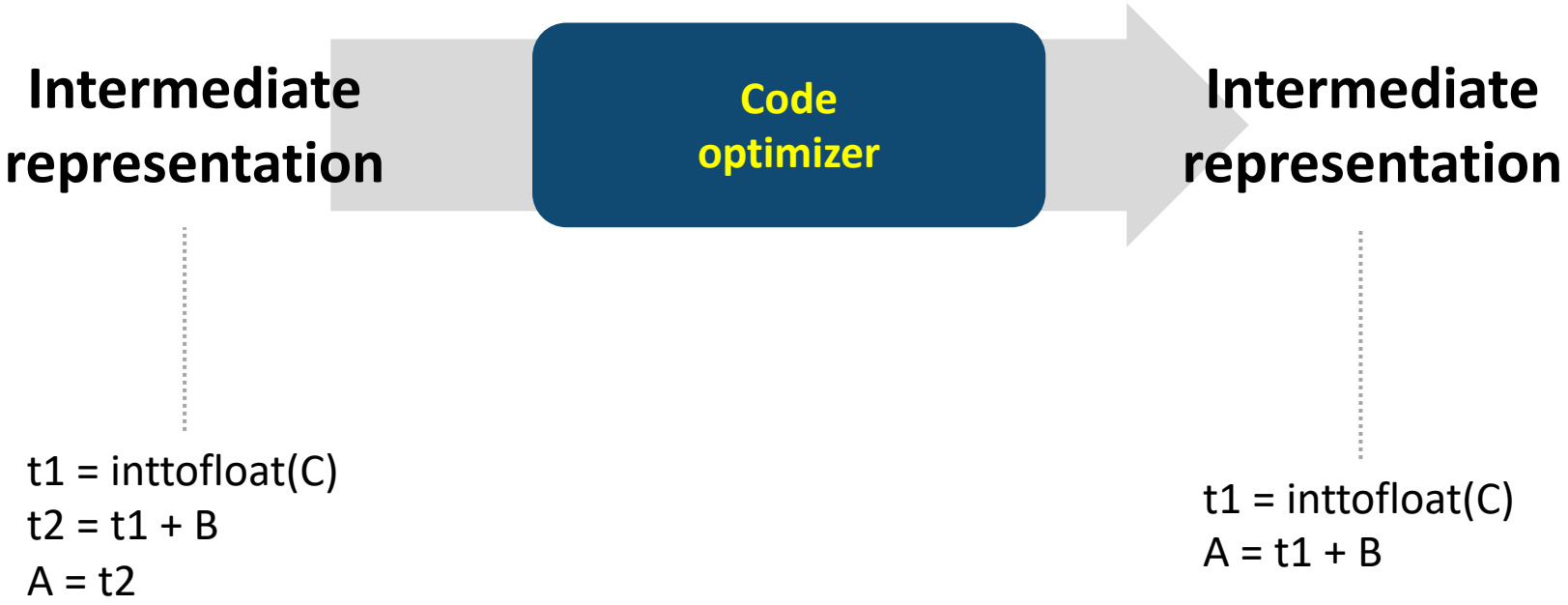
They should be easy to produce and easy to translate into a target machine code



Code optimizer (optional)

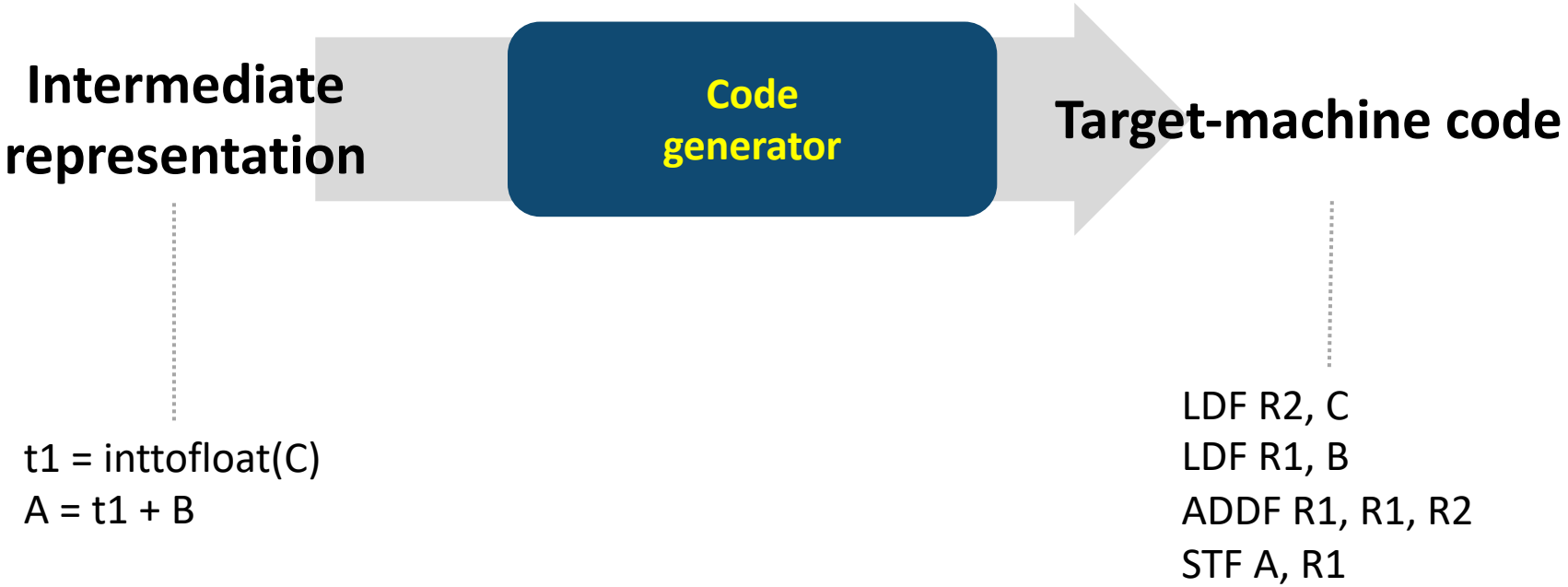
Attempts to improve the intermediate code so that better target code will result

e.g, Better code = faster or shorter code

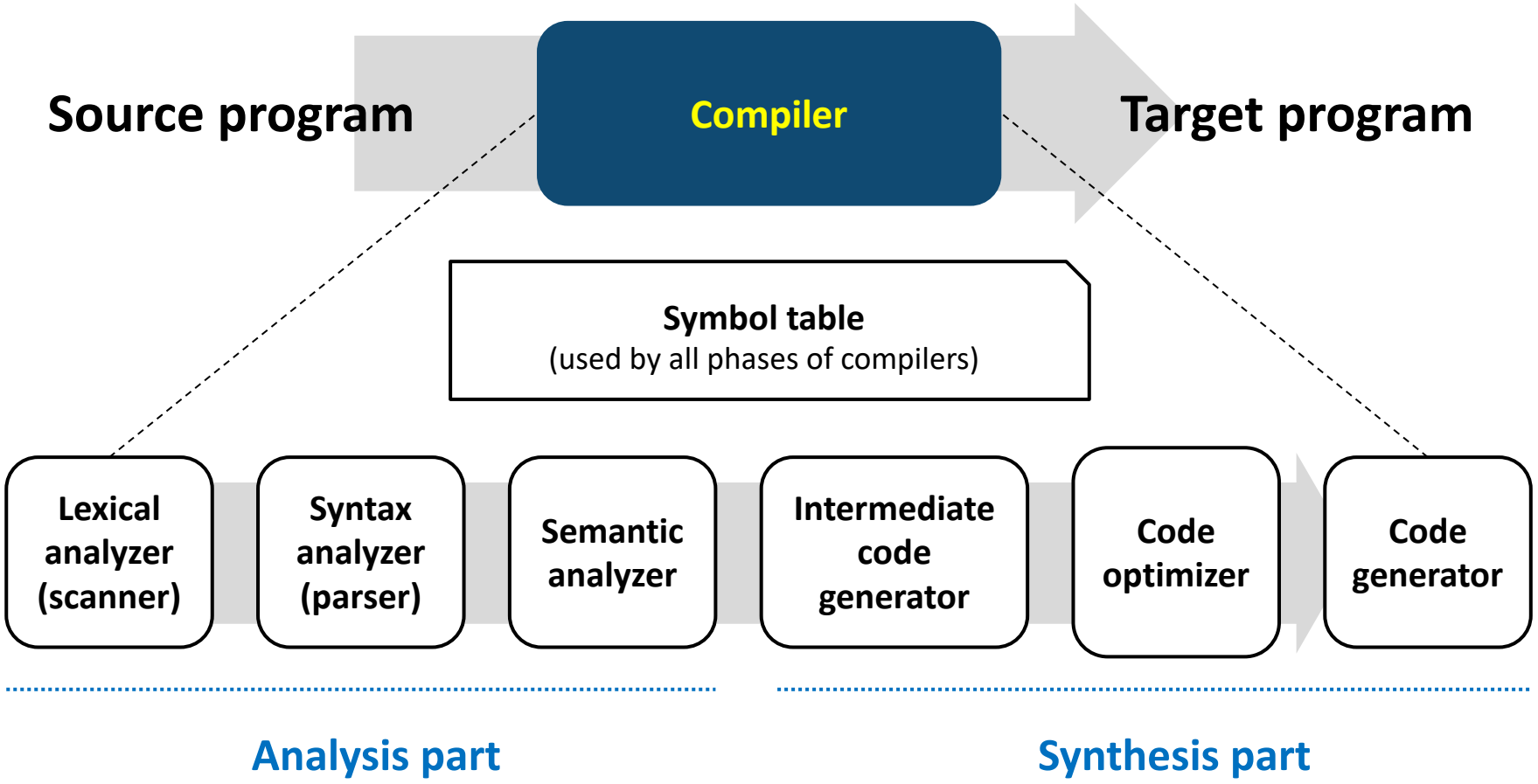


Code generator

Maps an intermediate representation of the source into the target language



Summary


Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part