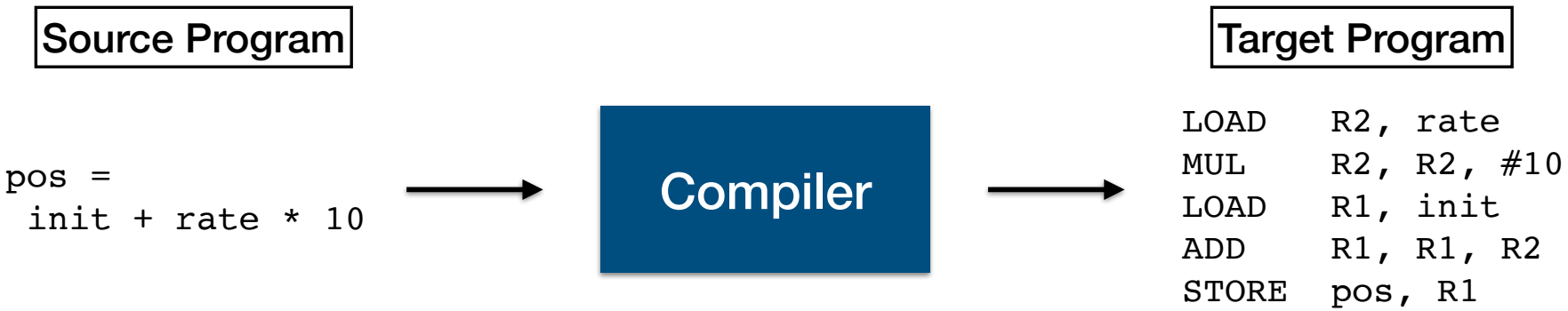
EC3204: Programming Languages and Compilers

Lecture 1 — Overview of Compilers

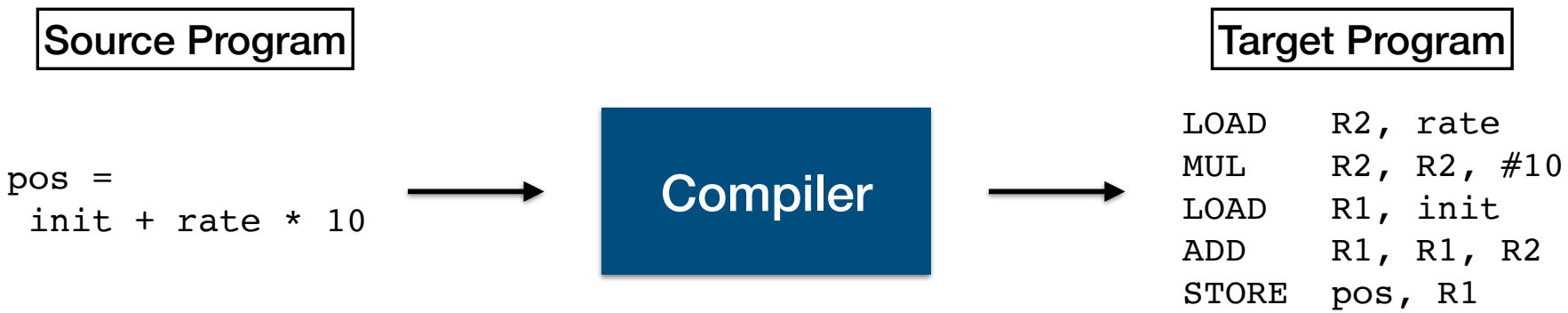
Sunbeom So
Fall 2024

Compiler: Programming Language Translator



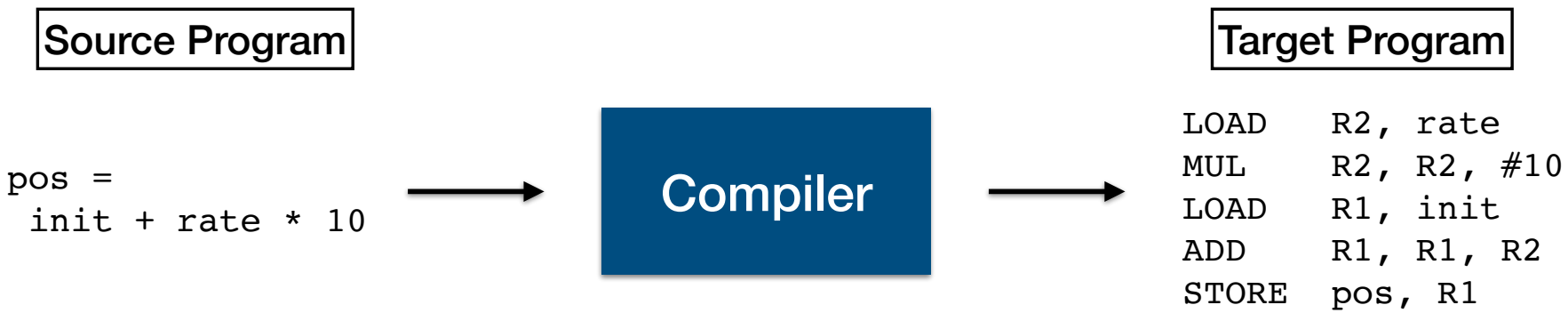
- Software systems that translate a program written in one language ("source language") into a program written in another language ("target language").

Compiler: Programming Language Translator



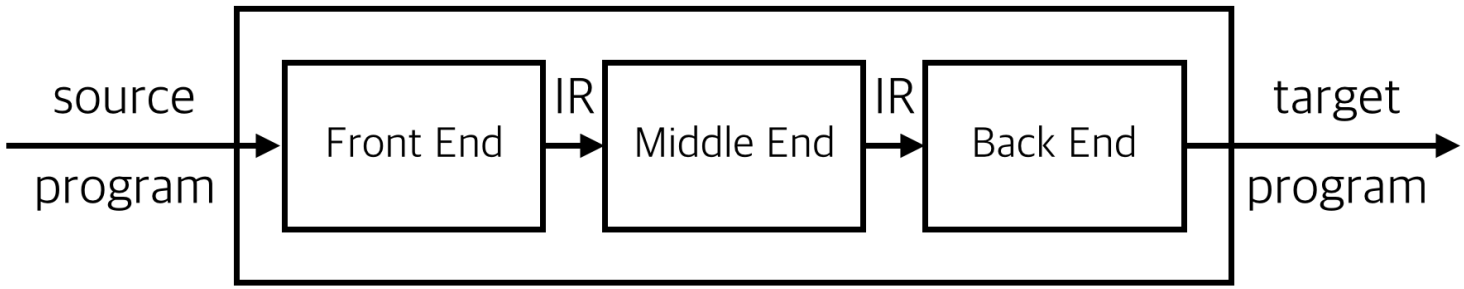
- Software systems that translate a program written in one language ("source language") into a program written in another language ("target language").
- Typically,
  - the source language is a high-level language (e.g., C, Java).
  - the target language is a low-level machine language (e.g., x86, MIPS).

Compiler: Programming Language Translator



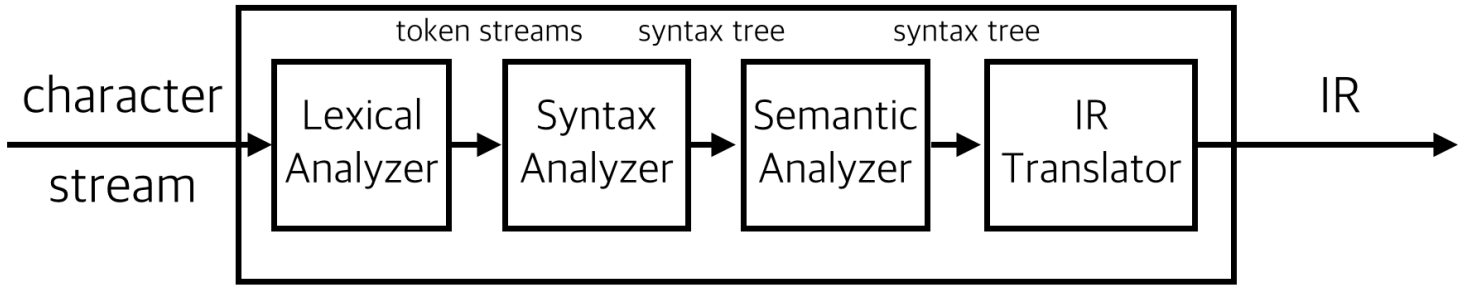
- Software systems that translate a program written in one language ("source language") into a program written in another language ("target language").
- Typically,
  - the source language is a high-level language (e.g., C, Java).
  - the target language is a low-level machine language (e.g., x86, MIPS).
- cf) Transpiler: source-to-source compiler (i.e., both source and target languages are high-level languages).

Structure of Modern Compilers



- **Front-end:** understands the source program and translates it to an intermediate representation (IR).
- **Middle-end:** takes a program in IR and optimizes it in terms of efficiency, energy consumption, and so on.
- **Back-end:** transforms the IR program into a machine-code.

Front-End



- **Lexical analyzer:** transforms the character stream (i.e., source code) into a token stream.
- **Syntax analyzer:** transforms the stream of tokens into a syntax tree.
- **Semantic analyzer:** checks if the source program has some semantic errors or not.
- **IR translator:** translates the syntax tree into an IR.

Lexical Analyzer (Lexer, Tokenizer, Scanner)

A lexer analyzes the lexical structure of the source program.

- **Input:** character stream

  `pos = init + rate * 10`

- **Interim result:** a sequence of **lexemes** ("meaningful" sequences of characters)

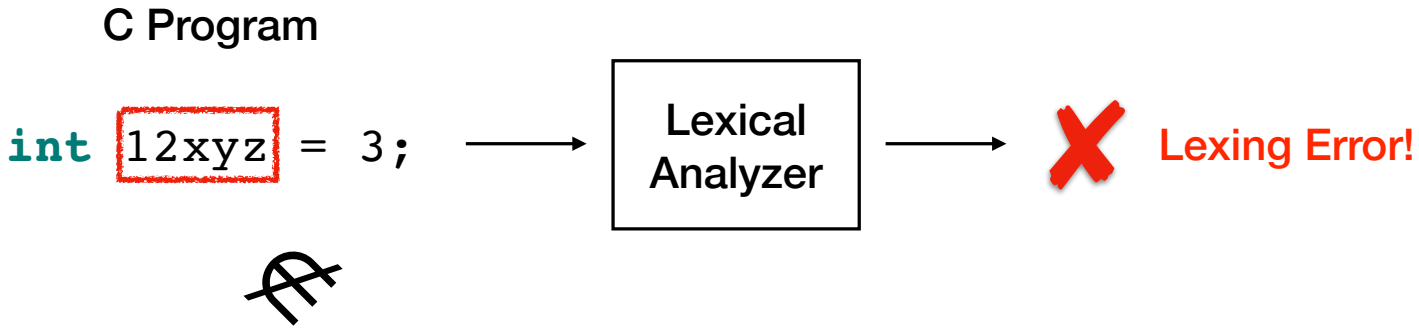
  [`pos`, `=`, `init`, `+`, `rate`, `*`, `10`]

- **Output:** a sequence of **tokens** (lexemes with abstract symbols)

  [(ID, `pos`), ASSIGN, (ID, `init`), PLUS, (ID, `rate`), MULT, (NUM,`10`)]

Lexical Analyzer (Lexer, Tokenizer, Scanner)

A lexer rejects programs with invalid tokens.



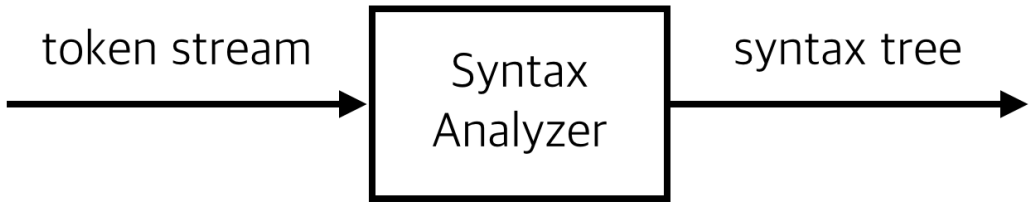
$$[A-Za-z\_]([A-Za-z\_]+[0-9])*$$

Identifiers should start with English alphabet or '_'.

Regex for C identifiers

Syntax Analyzer (Parser)

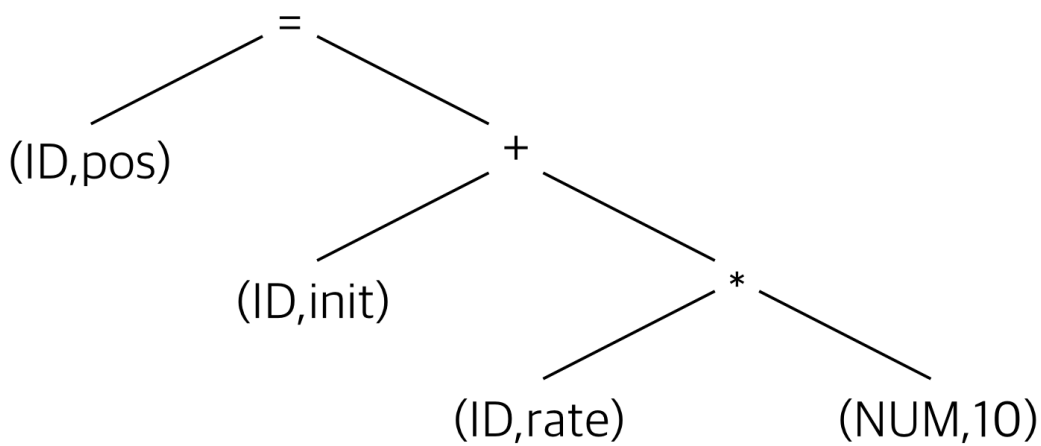
A parser recognizes the grammatical structure of the source program.



- **Input:** a sequence of tokens

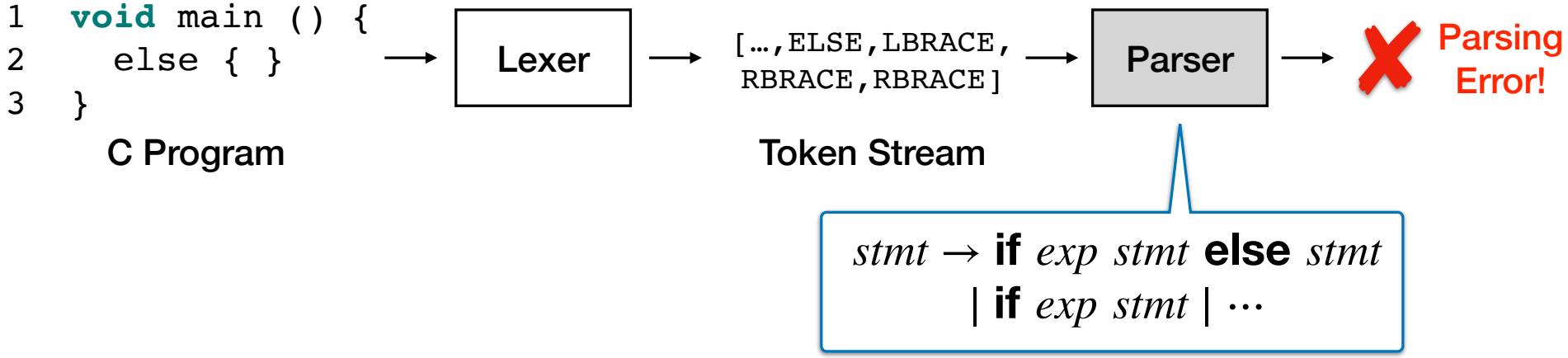
  [(ID, `pos`), ASSIGN, (ID, `init`), PLUS, (ID, `rate`), MULT, (NUM,`10`)]

- **Output:** syntax tree (grammatical structure of a source program)



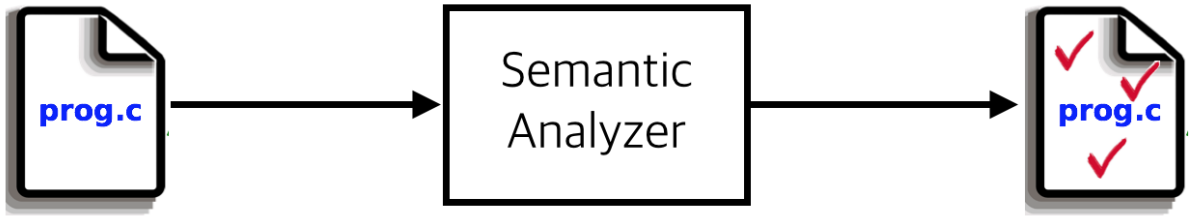
Syntax Analyzer (Parser)

A parser rejects syntactically wrong programs (i.e., programs that cannot be expressed by a context-free grammar).



Semantic Analyzer

A semantic analyzer detects semantic errors (e.g., type errors, buffer-overflow, null-dereference, divide-by-zero).



(Example) The following Java program is lexically and syntactically valid.

```
int x = 1;
String y = "hello";
int z = x + y;
```

However, it will be eventually rejected by the javac with the following error message: "`error: incompatible types: String cannot be converted to int`".

Source Technology of Semantic Analyzer

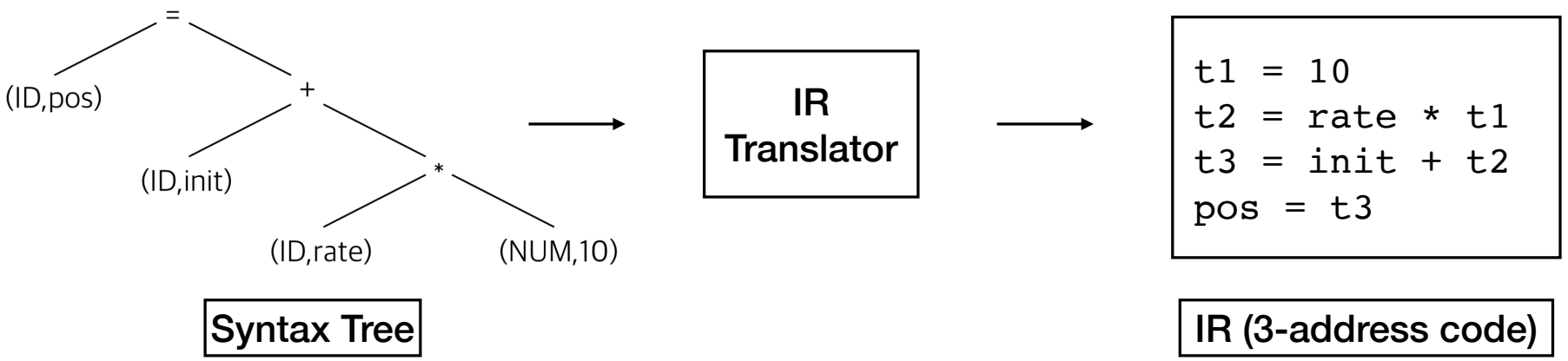
- **Static program analysis:** predict program behaviors "statically" and "automatically".
  - "static": analyze program texts (i.e., source code) without actually running the programs.
  - "automatic": SW that analyzes SW
- Application examples
  - Verification: e.g., does this program always satisfy its formal specification?
  - Bug-finding: e.g., does this program have integer overflow bugs?
  - Equivalance checking: e.g., are these two programs semantically equivalent?

IR Translator

An IR translator converts the syntax tree into an Intermediate Representation (IR):

- lower-level than the source language
- higher-level than the target language (machine language)

(Example of IR) "three-address code": a sequence of assembly-like instructions with at most three operands per instruction.

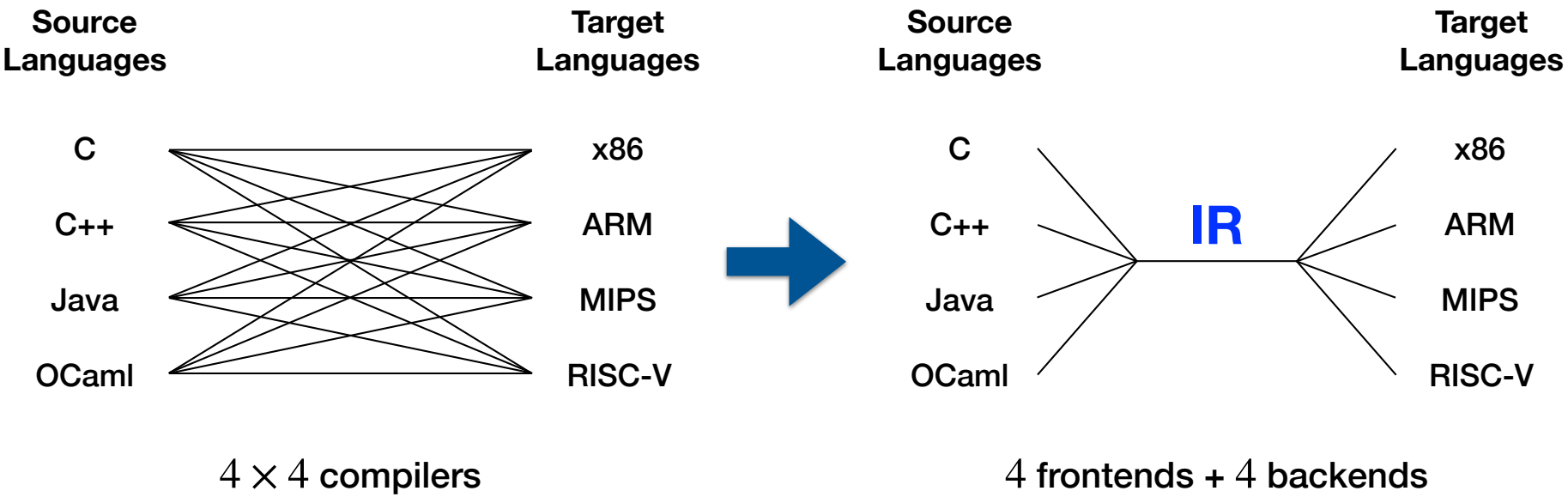


IR Translator

- Q: Why do we need IR? Why not directly translate the syntax tree into a target language?

IR Translator

- Q: Why do we need IR? Why not directly translate the syntax tree into a target language?
- A: To reduce the efforts for building compilers

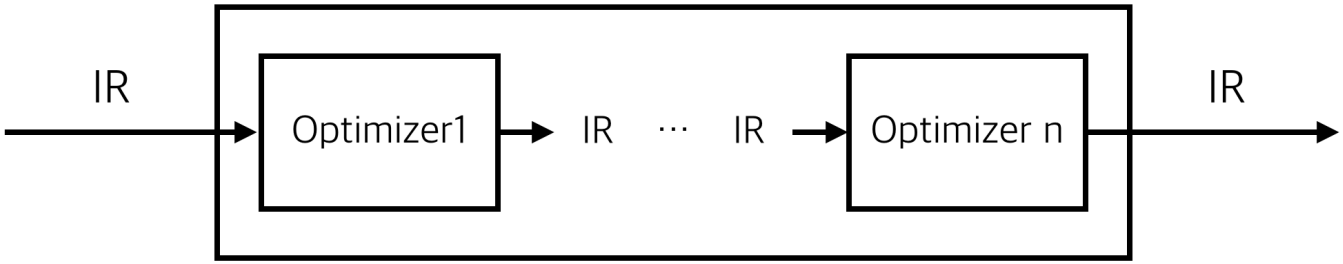


4×4 compilers

4 frontends + 4 backends

Middle-End (Optimizer)

Transform the intermediate code to have better performance:



(Example)

```
t1 = 10
t2 = rate * t1
t3 = init + t2
pos = t3
```

original IR

```
t1 = 10
t2 = rate * 10
t3 = init + t2
pos = t3
```

```
t2 = rate * 10
t3 = init + t2
pos = t3
```

```
t2 = rate * 10
pos = init + t2
```

final IR

Back-End

Generate the target machine code:

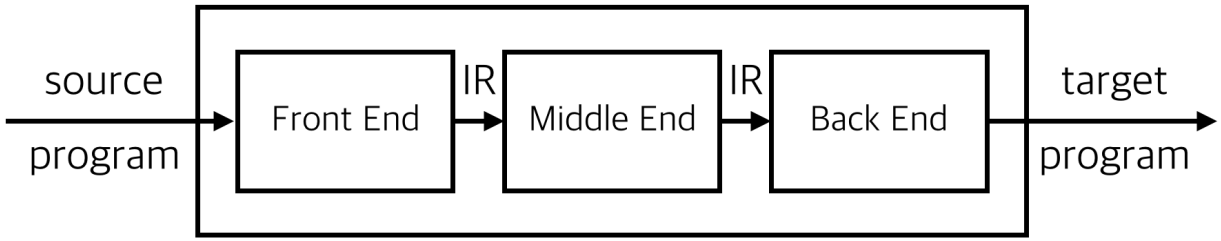
(Example) From the optimized IR

```
t2 = rate * 10
pos = init + t2
```

the back end generates the machine code (assuming two registers, `R1` and `R2`, are available)

```
LOAD  R2, rate
MUL   R2, R2, #10
LOAD  R1, init
ADD   R1, R1, R2
STORE pos, R1
```

Summary



- A compiler is a programming language translator.
- A modern compiler consists of three main phases.
  - Front-end understands the syntax and semantics of source program.
  - Middle-end improves the efficiency of the program.
  - Back-end generates the target program.