27 Mar 2025 – Compilers

Programming Languages

→ notations for describing computations to people and to machines

program
↓
translate into a form which can be executed by a computer

~~glorified translator~~

→ Software systems that do this

↓

Compilers

→ How to design and implement compilers
→ few basic ideas
→ Principles here will be applicable at many other places in C.S.

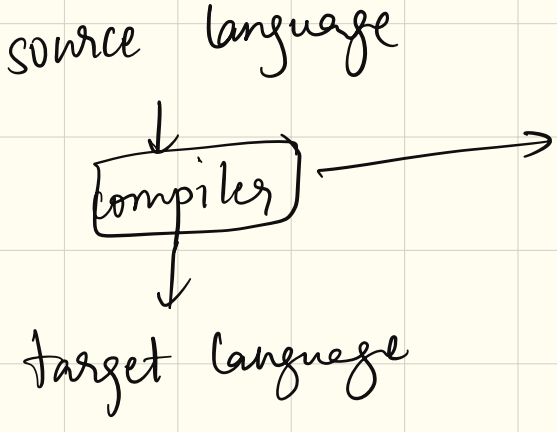
Chap 1 → introduce diff forms of language translators

→ give a high-level overview of a typical compiler

→ trends in programming languages

→ Rel^{n} between Compiler Design and CS theory

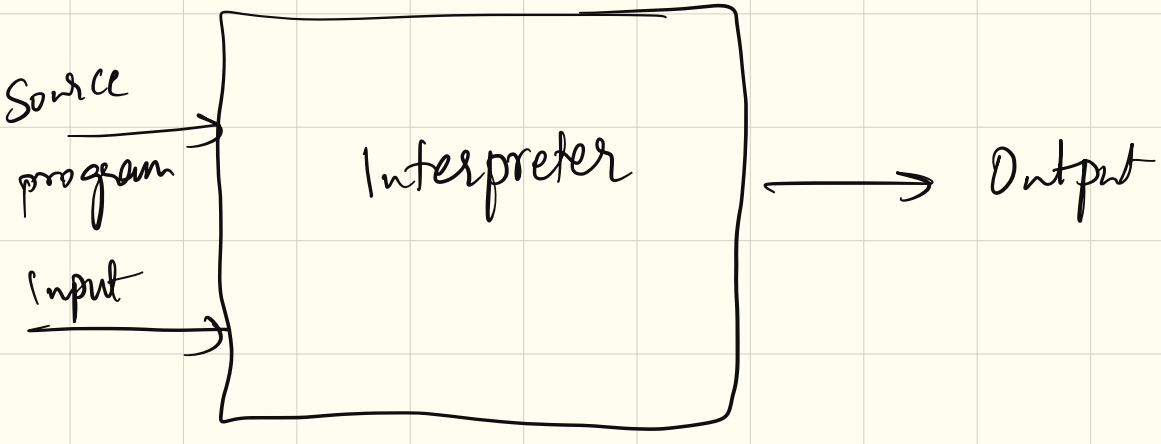
1.1 Language Processors



→ important role:
detect errors during translation and report

Interpreter → another kind of language processor

instead of producing target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by user



Compiler
produce machine-language target program

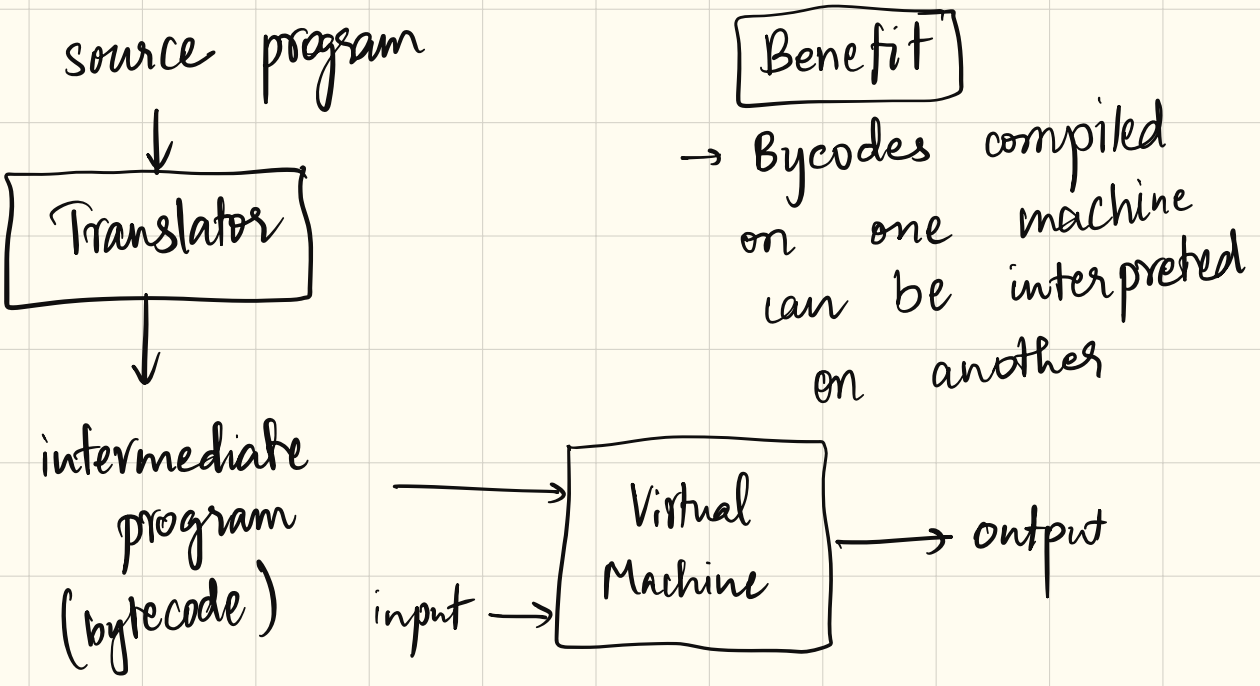
usually faster than

Interpreter
mapping inputs to outputs

→ Interpreter can, however, provide better error diagnostics because it executes source program statement by statement
⊕ has input.

Example 1.1: Java language processors combine both compilation and interpretation

---



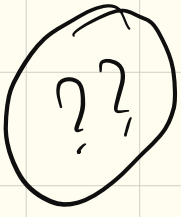
Benefit

→ Bycodes compiled on one machine can be interpreted on another

Fast compilation for Java → Just-in-time compilers

↓

Translate bytecode into machine language immediately before they run the intermediate program to process the input.

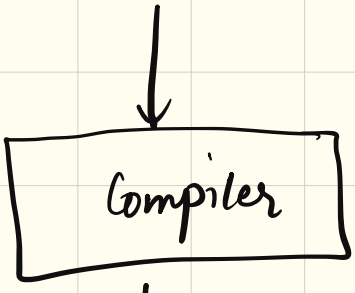


Other programs may be required to create an executable target program:

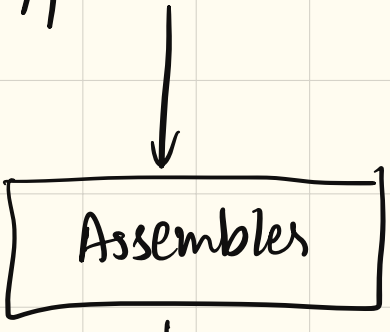
① stored program may be stored into modules — sep. files

→ collecting source program ⊕ expand macros in source language statements } Preprocessor

② Modified source program



may produce assembly Language program as o/p → easier to produce as output + easier to debug



③

Relocatable machine code

generate a relocation table

Compiler add placeholder or relative address

Machine code: low-level instructions that CPU understands

Relocatable: can be moved to a different memory address

needs to be linked with diff. libraries

④ Large programs are compiled in pieces, relocatable machine code

↓

linked with other relocatable object files and library files

→ code that runs on the machine

Linker: resolves memory addresses (code in one file may refer to a location in another file)

↓

Loader: puts together all of the executable object files

A compiler that translates HLL ⟶ another HLL is called source-to-source translator

Advantages to using C as a target for source-to-source?

→ C is lightweight

* C is low-level enough to be close to the metal, but still high-level enough to be portable, readable and compatible with standard toolchains

Portability

↓

Write your compiler once and let gcc and clang do the platform specific optimization

Toolchain maturity

↓

C ecosystem: many debuggers, profilers, analyzers, etc. You instantly get to piggyback decades of tooling

Interoperability

↓

easy to link libraries

Simple runtime model:

No classes, no inheritance, no exceptions, no runtime type info.

C++ as target ?

- way more complex
- less predictable performance
- Toolchains and ABI compatibility can be trickier.

Rust ? ✗

What kind of languages would benefit from targetting C++ instead of C?

Assembler tasks:

① Symbol table management

- keep track of labels

```
LOOP: add x1, x2, x3;
```

- map them to memory addresses

② Instruction translation:

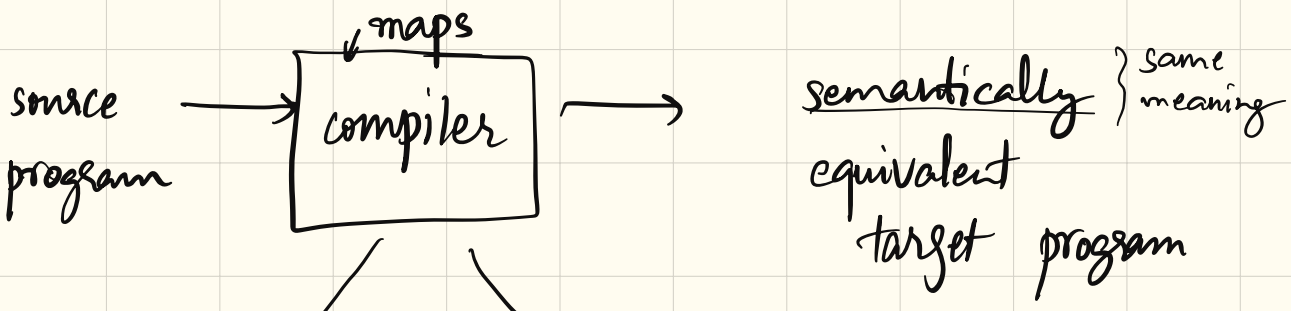
MOV A, B ⟶ machine code

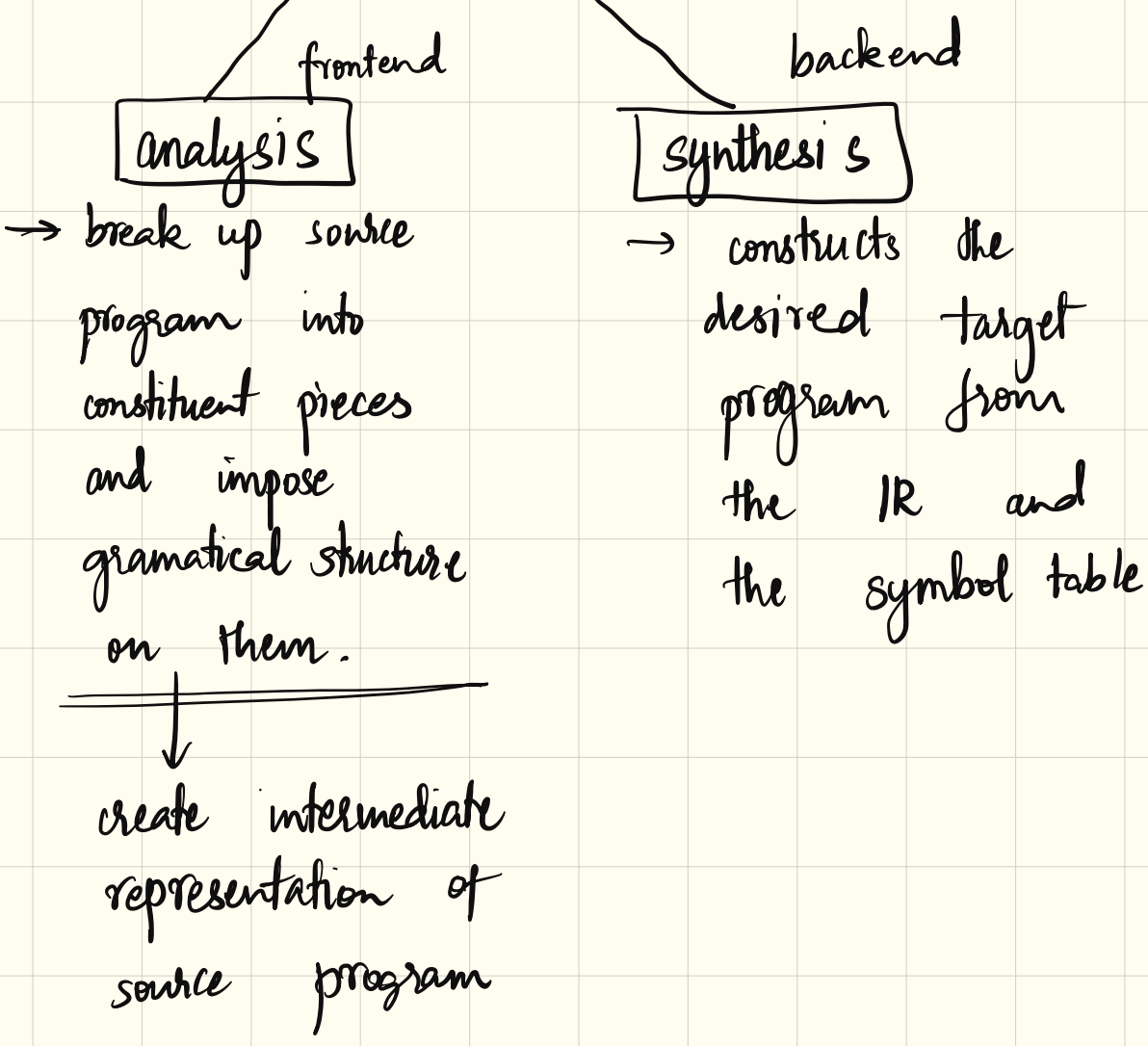
(MOV A, B: human readable machine code)

③ Address resolution, relocation & linking

The structure of a compiler

Up to this point,





→ If analysis part detects that the source program is either syntactically ill formed or semantically unsound

↓

provide informative messages for user to take action

→ Analysis part collects info about the source program and stores it in a DS called symbol table.

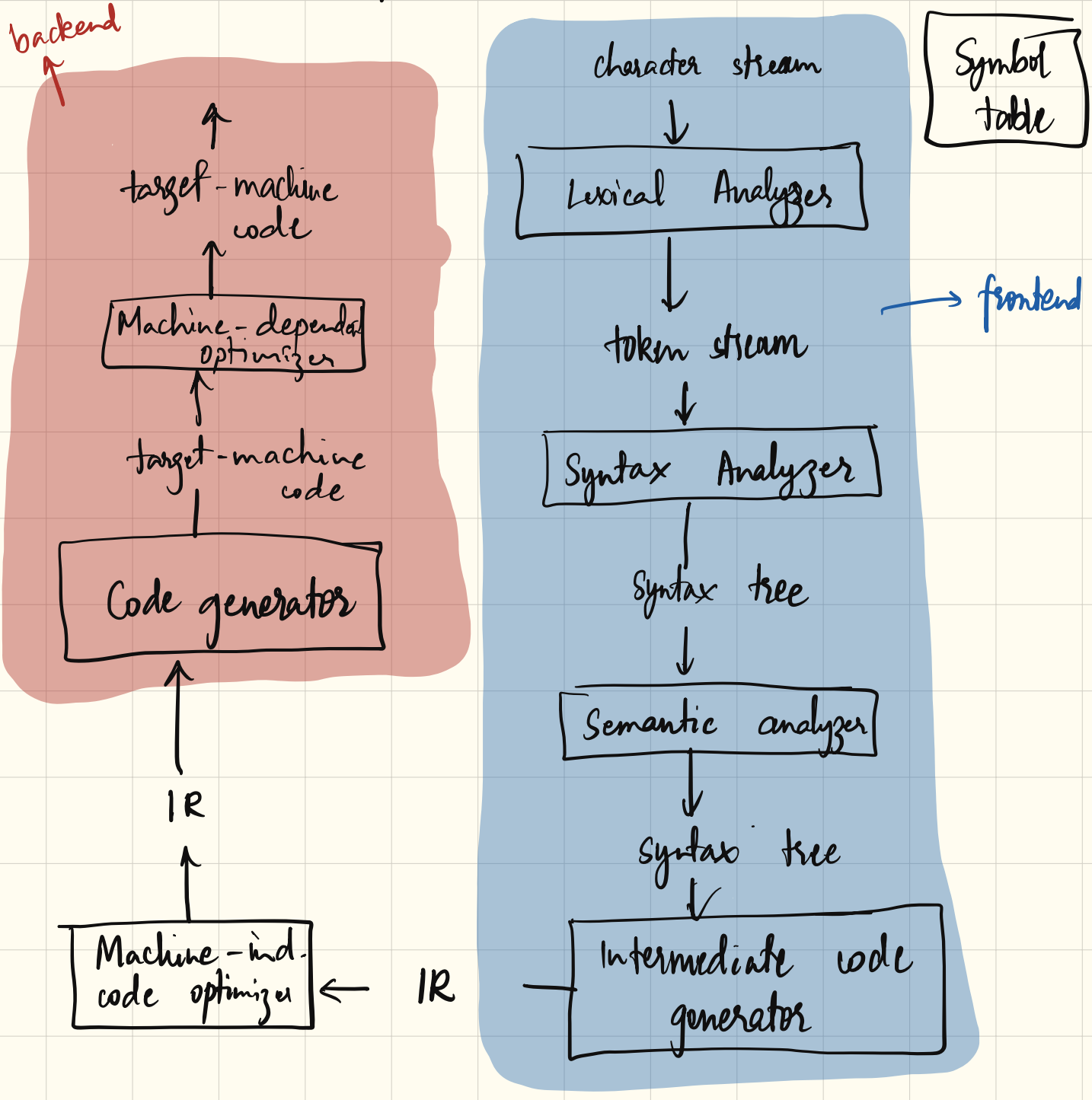
↓

passed along with IR to synthesis part.

→ Compilation operates as a sequence of phases, each of which transforms one representation of

the source program into another.

Typical decomposition of phases:



→ Symbol table is used by all phases of the compiler.

information about source program

→ Some compilers have a machine independent optimization phase b/w frontend and backend.

→ perform transformations on the IR, so that backend can produce better target program

Machine independent
Mache dependent

→ one or the other optimizations may be missing

Lexical Analysis

→ aka scanning.

→ reads stream of characters making up the source program and groups the characters into **lexemes**

For each lexeme, LA produces as output a token:

< token-name, attribute-value >

- token-name → passed to syntax analyser
- attribute-value → points to an entry in the symbol table for this token.

token name → abstract symbol used during syntax analysis

Information from the symbol table is needed for semantic analysis and

code generation.

Ex: Source program contains:

position = initial + rate * 60

could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyser:

① position

↳ lexeme mapped to a token

<id, 1>

- id → abstract symbol standing for: identifier
- 1 → points to symbol table entry for position

Symbol table entry for identifier: → name, type

holds info about the identifier.

② `=` → assignment symbol

↓

lexeme mapped into the token

⟨ = ⟩

↓

needs no attribute value

we could have used any abstract symbol for token name, such as assign but for notational convenience, lexeme itself becomes the name of the abstract symbol.

③ initial

↳ lexeme mapped into the token ⟨ id, 2 ⟩

↳ points to the STE for initial.

④ + ⟶ <+>

lexeme token

⑤ rate ⟶ <id, 3>

⑥ * ⟶ <*>

⑦ 60 ⟶ <60>

lexeme

Technically:
<number, 4>
4 points to symbol table for internal representation of integer 60

Blanks separating lexemes ⟶ discarded by LA

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

Chapter 2 → tokens for no.

Chapter 3
↳ building Lex Analysers

Syntax Analysis

→ Second phase

→ Parsing

→ Uses the first components of the tokens (produced by LA) to create a tree-like Intermediate Representation (IR).

→ depicts the grammatical structure of the token stream.

position = initial + rate * 60

↓

Lex Analyzer

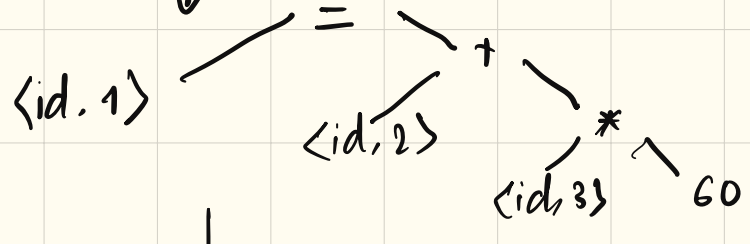
↓

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

↓

Syntax Analyser

↓



Symbol table

1	position	...
2	initial	...
3	rate	...

↓

Semantic Analayser

↓

Intermediate Code generator

↓

Code optimizer

↓

Code generator

Tree shows the order in which operations must be performed.

↓

consistent with the usual conventions of arithmetic.

Subsequent phases → use the grammatical structure to help analyze the source program and generate target program.

Chap 4

CFG to specify grammatical structure of Programming languages

discuss algorithms for constructing efficient syntax analysers

Semantic Analysis