CS3501
Compiler Design

INTRODUCTION

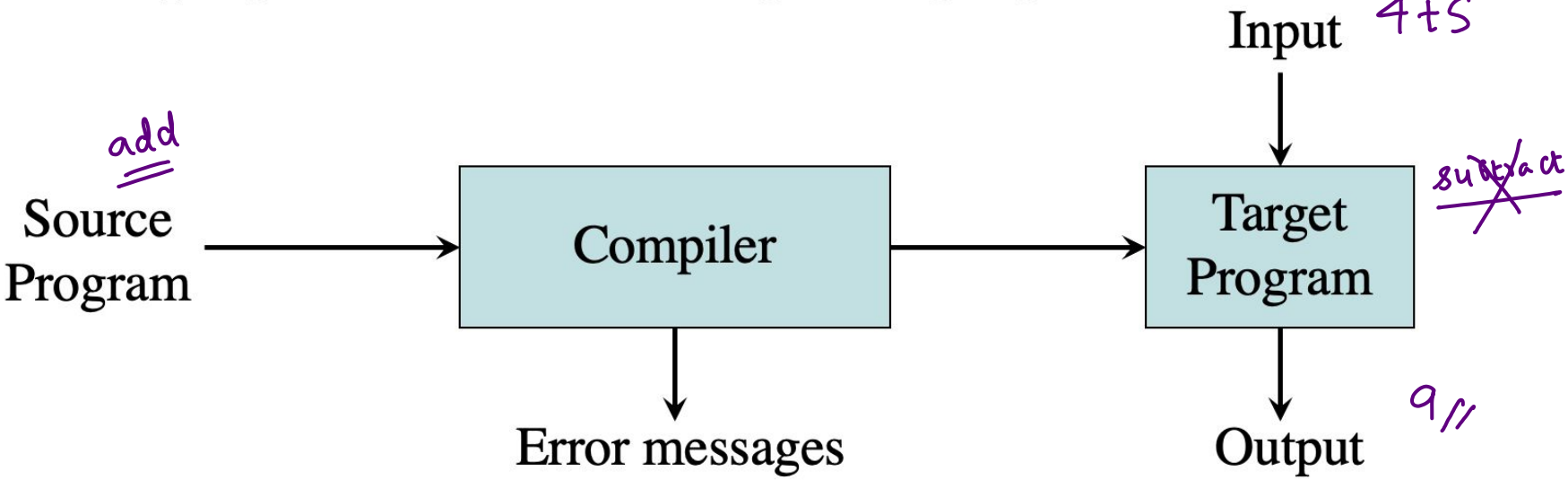
Why Compiler?

What is this new language? I can understand only 0 and 1!!!!!!!!!!!!!!

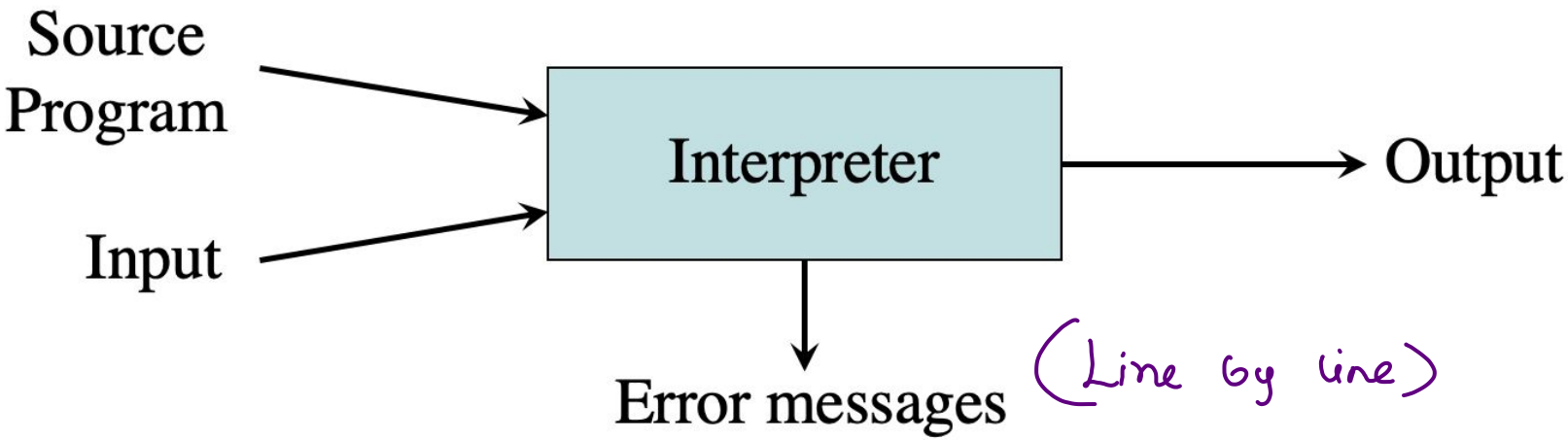
```
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
sum = num1 + num2
print("Sum:", sum)
```

Address	Machine Code
0000	1010000100000010
0001	1010000100000011
0010	0100000001000001
0011	1100000100000010
0100	0110000000000000
0101	0000000000000000

- "*Compilation*"
  - Translation of a program written in a source language into a semantically equivalent program written in a target language



- "*Interpretation*"
  - Performing the operations implied by the source program



Compiler	Interpreter
A compiler translates the entire source code in a single run.	An interpreter translates the entire source code line by line.
It consumes less time i.e., it is faster than an interpreter.	It consumes much more time than the compiler i.e., it is slower than the compiler.
It is more efficient.	It is less efficient.
CPU utilization is more.	CPU utilization is less as compared to the compiler.
Both syntactic and semantic errors can be checked simultaneously.	Only syntactic errors are checked.
The compiler is larger.	Interpreters are often smaller than compilers.
It is not flexible.	It is flexible.
The localization of errors is difficult.	The localization of error is easier than the compiler.
A presence of an error can cause the whole program to be re-organized.	A presence of an error causes only a part of the program to be re-organized.
The compiler is used by the language such as C, C++.	An interpreter is used by languages such as Java.


Thank you!

CS3501
Compiler Design

Language Processing System

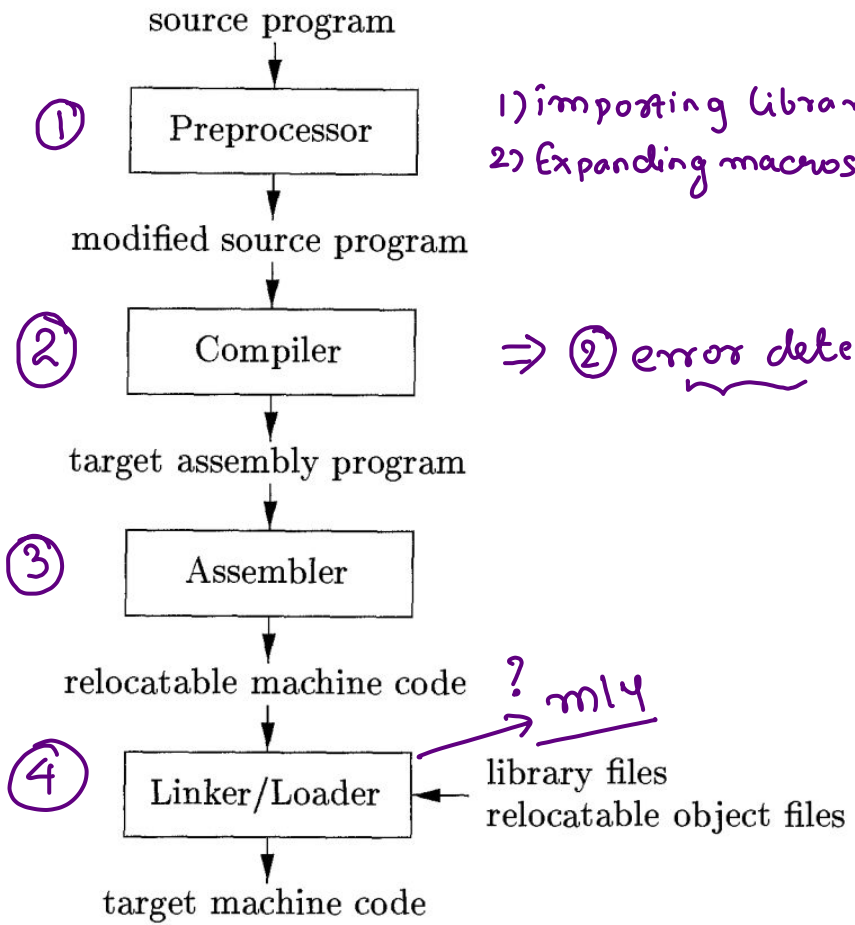


Figure 1.5: A language-processing system

Simple Example

```
#include<stdio.h>
int main(){
printf("Hello World!!");
```

PRE-PROCESSING

This is the very first stage that a source code goes through. The following tasks are completed during this stage:

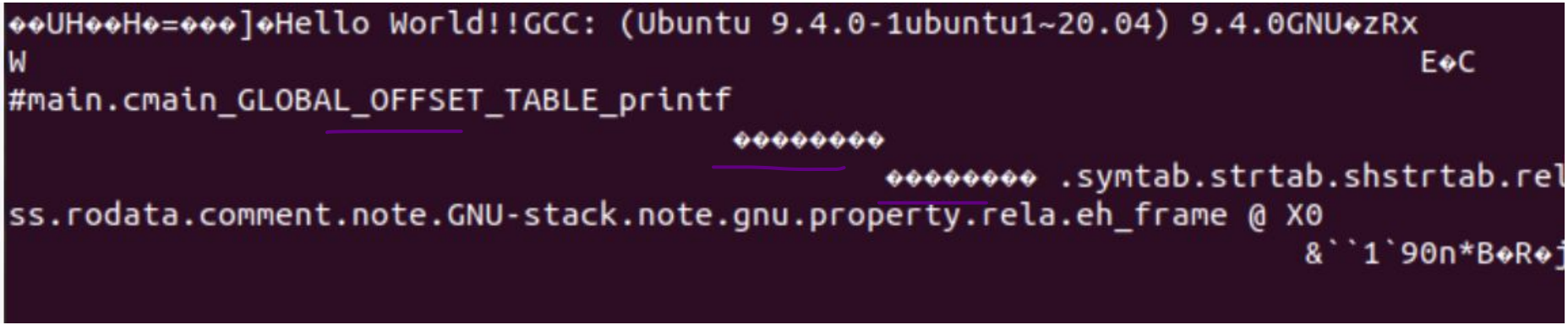
- Comment Removal
- Macros Expansion
- Expansion of the files contained

```
# 858 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 873 "/usr/include/stdio.h" 3 4

# 2 "main.c" 2

# 2 "main.c"
int main(){
printf("Hello World!!");
}
```

Compiler: The compiler takes the preprocessed C code and translates it into assembly code (sometimes referred to as object code) specific to the target machine's architecture. The compiler performs syntax and semantic analysis, type checking, and optimization to generate efficient assembly code.

```
        .file   "main.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello World!!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
```

Assembler – The assembler takes the generated assembly code and translates it into machine code or binary code that the computer's CPU can directly execute. The assembler will produce an object file containing the machine code representation.



Linker:

In many cases, a C program consists of multiple source files (e.g., "main.c" and other separate source files). The linker is responsible for combining all the object files and resolving any references between them to create an executable file. In this simple example, there's only one source file, so the linker's job is relatively simple.

Loader:

The loader is responsible for loading the executable file into memory and preparing it for execution. It allocates memory for the program, resolves memory addresses, and starts the program's execution. The loader is an essential part of the process when creating an executable file that can be run by the operating system.


Thank you!

CS3501
Compiler Design

Phases of Compiler

The Analysis-Synthesis Model of Compilation

- There are two parts to compilation:
  - *Analysis* determines the operations implied by the source program which are recorded in a tree structure
  - *Synthesis* takes the tree structure and translates the operations therein into the target program

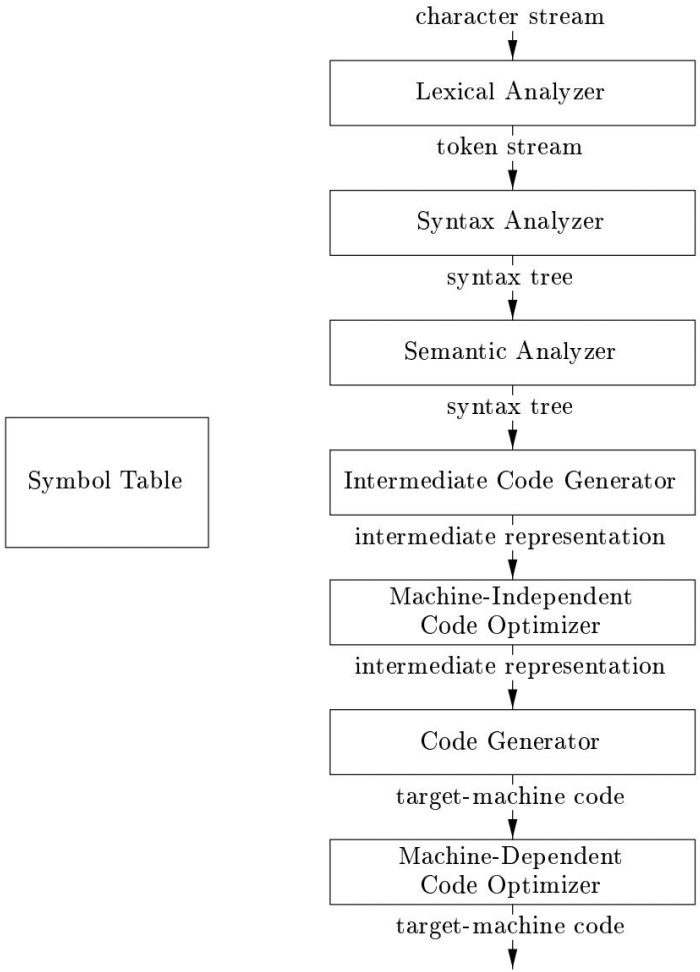
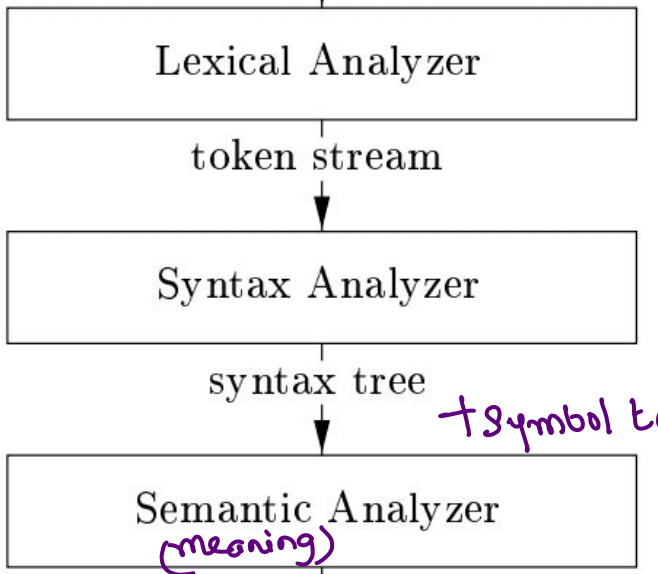


Figure 1.6: Phases of a compiler



src pgm → a = b + c → operators

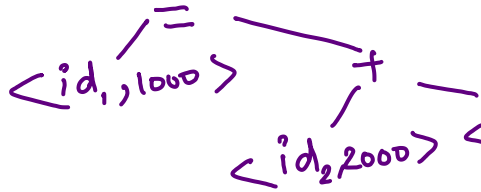
operands

→ tokens/lexemes (smallest indivisible unit)

1) a → <id1, 1000>
2) = → <=>
3) b → <id2, 2000>
4) + → <+>
5) c → <id3, 2500>

↑ abstract name ↕ pos. in symbol table

⊗ no entry for operators



+ symbol table

⊗ leaf → operands
non-leaf → operators
⊗ takes care of precedence

① ⊗ meaningful w.r.t lang

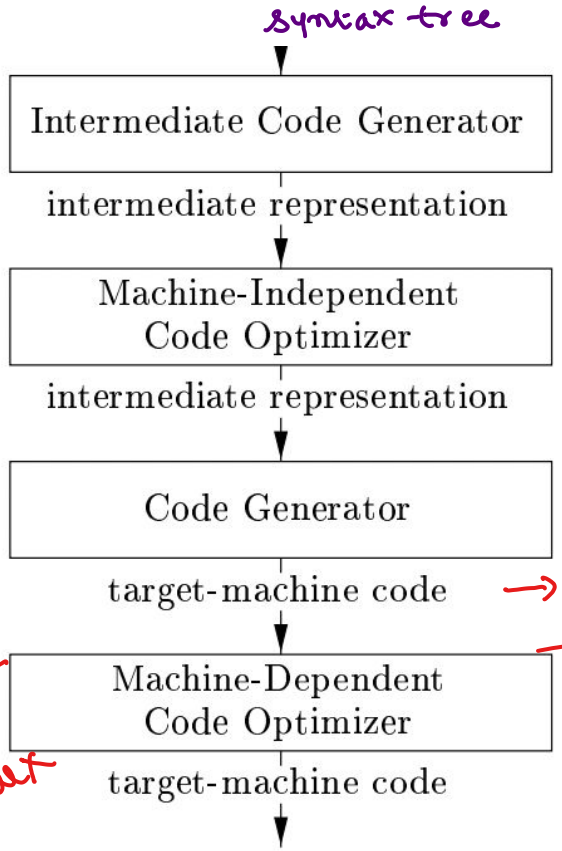
b + c = d ✗
4 = a ✗

② type checking

b = 4 c = "hello"
a = b + c ✗

③ type coercion

b = 4 c = 5.2
↳ float //



apple
Win
linux

⊗ easy to produce
⊗ easy to translate
eg. syntax tree

Src → Target pgm

Src → ICG → Target

2 ⊗ Three address code

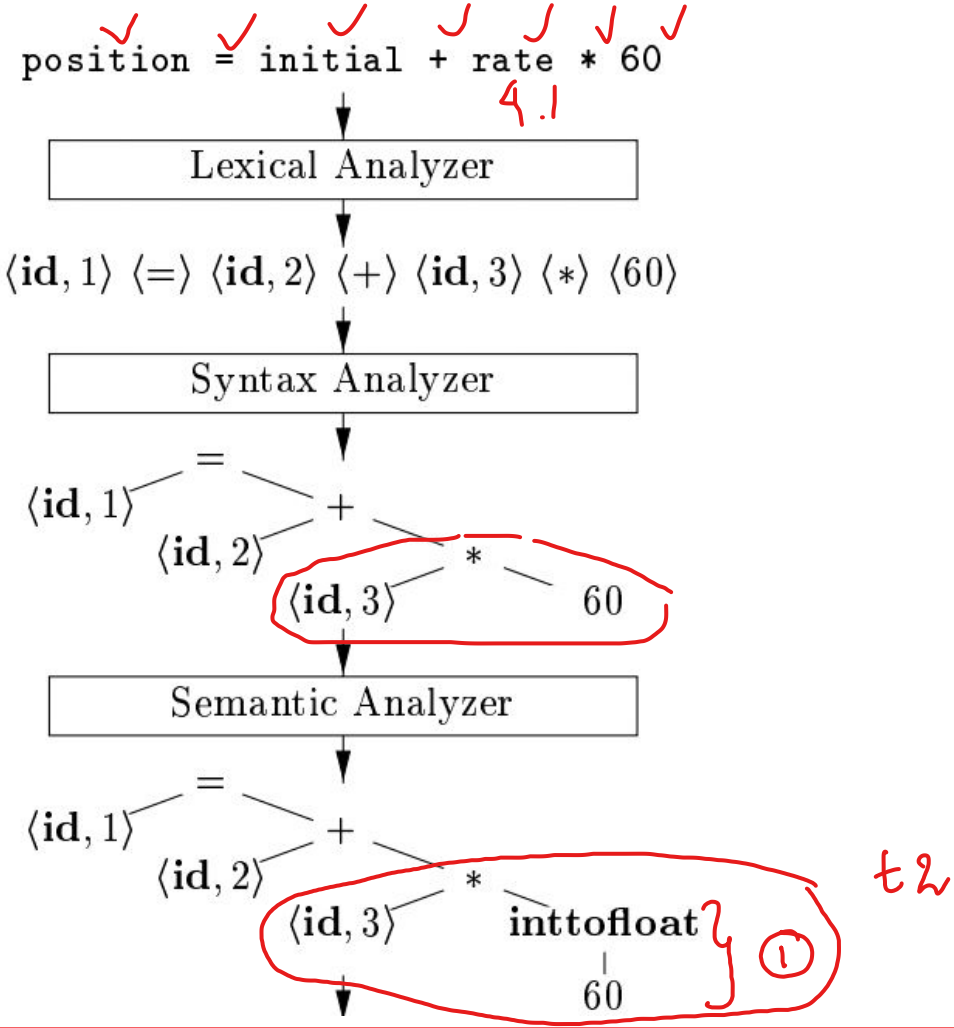
1) @ most one operator in right
a = b + c ✓
a = b + c + d ✗
2) temp name to hold value computed
temp = b + c
a = temp + d
3) few inst can have < 3 operands

→ improve eff.
→ shorter & faster
→ remove unwanted opn.

→ seq. of machine inst.
→ req ⇒ optimal use

Symbol table
→ data about data ① name - type - scope
② fn → parameters, type, return

Eg for phases



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

SYMBOL TABLE

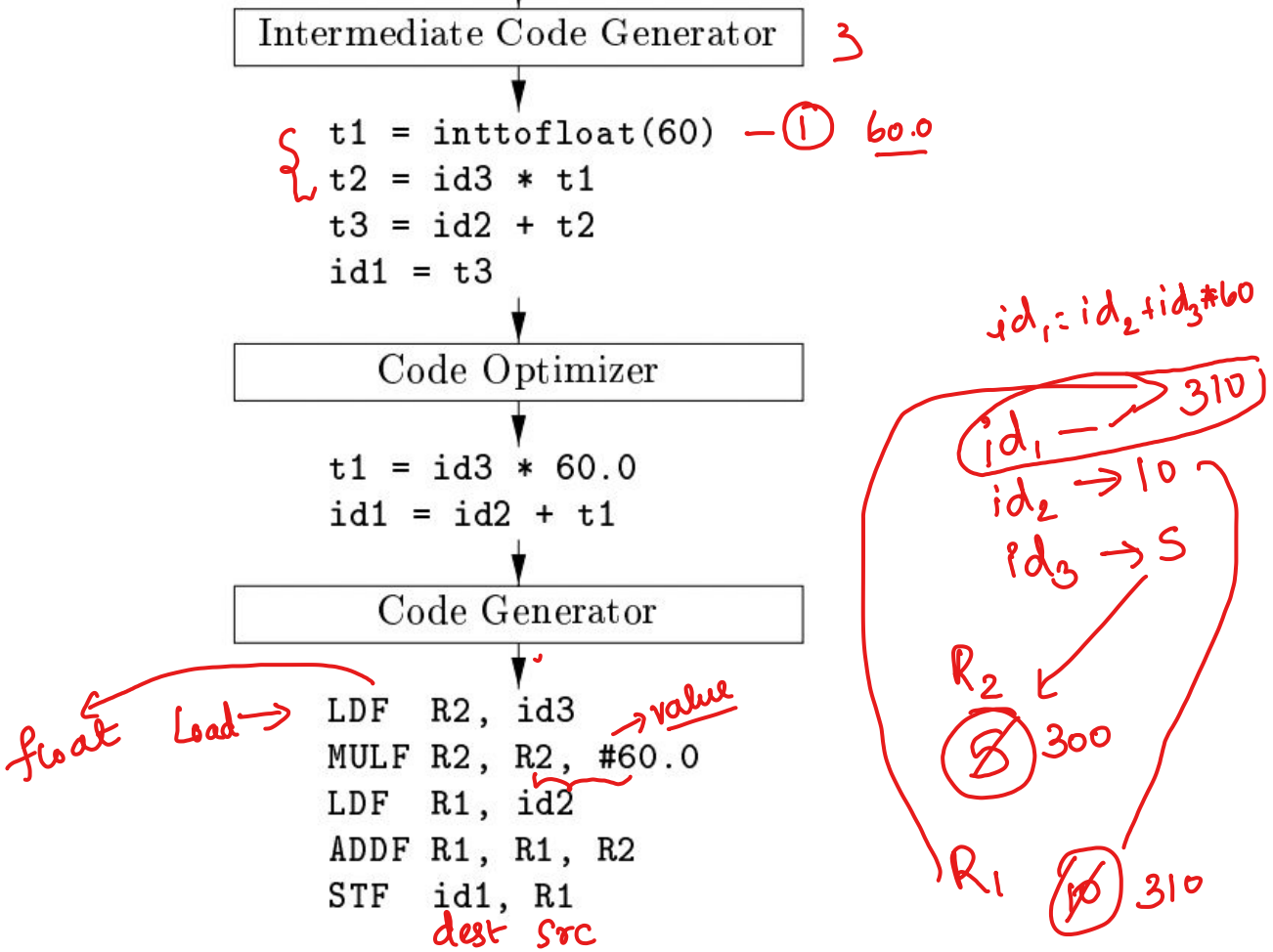


Figure 1.7: Translation of an assignment statement

1.2.1 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

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

that it passes on to the subsequent phase, syntax analysis. 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.

that it passes on to the subsequent phase, syntax analysis. 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.

For example, suppose a source program contains the assignment statement

$$\texttt{position = initial + rate * 60} \tag{1.1}$$

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. `position` is a lexeme that would be mapped into a token $\langle \mathbf{id}, 1 \rangle$, where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for `position`. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol `=` is a lexeme that is mapped into the token $\langle = \rangle$. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. `initial` is a lexeme that is mapped into the token $\langle \mathbf{id}, 2 \rangle$, where 2 points to the symbol-table entry for `initial`.

4. `+` is a lexeme that is mapped into the token $\langle + \rangle$.

5. `rate` is a lexeme that is mapped into the token $\langle \mathbf{id}, 3 \rangle$, where 3 points to the symbol-table entry for `rate`.

6. `*` is a lexeme that is mapped into the token $\langle * \rangle$.

7. `60` is a lexeme that is mapped into the token $\langle 60 \rangle$.[1]