Compilers
Week 1.1

General Notes

Introduction

- What is a ***Compiler***?
- What is a ***source code***? How do you define it?
- How do you differentiate between ***source code*** and ***machine language***?

- What is a ***high-level language***?
  - Human readable language,
  - Human can understand it, and
  - Can maximize productivity.

Introduction

- ***Programming languages***: are notations for describing computations to people and to machines.
    - All the software running on all the computers was written in some programming language.
- ***But***, before a program can be run, it first must be ***translated*** into a form in which it can be executed by a computer.

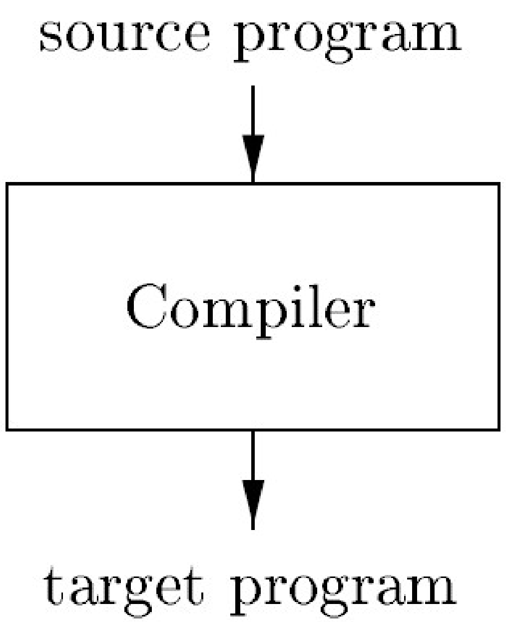
→ <u>The software systems that do this translation are called **compilers**</u>.

Language Processors

- A compiler is a program that can:

  1. Read a program in one language, ***the source language,*** and
  2. Translate it into a <u>*semantically*</u> equivalent program in another language, ***the target language***.

- An important role of the compiler is to ***report*** any ***errors*** in the <u>*source program*</u> that it detects during the translation process.

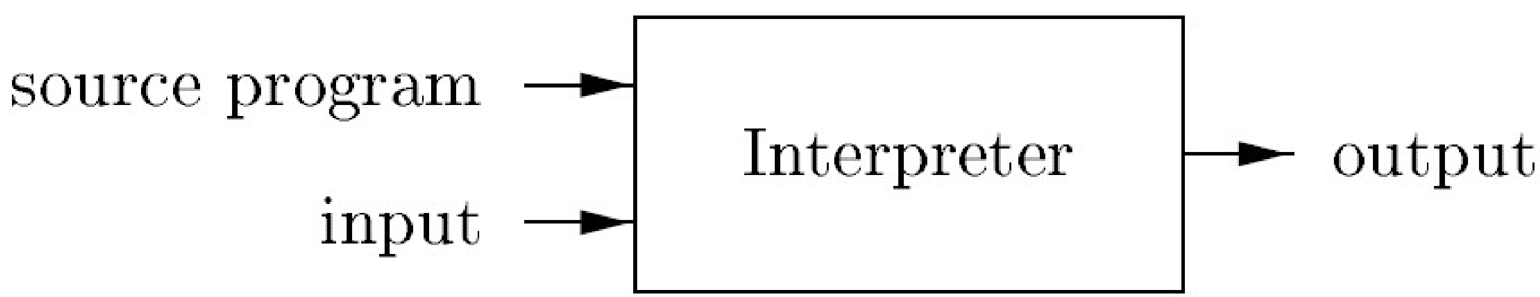


Language Processors

- If the ***target program*** is an executable ***machine-language*** program, it can then be called by the user to process <u>*inputs*</u> and produce <u>*outputs*</u>.

Language Processors

- An ***interpreter*** is another common kind of *language processor*
- Instead of producing a target program as a translation, an interpreter *directly executes* the operations specified in the source program on inputs supplied by the user.



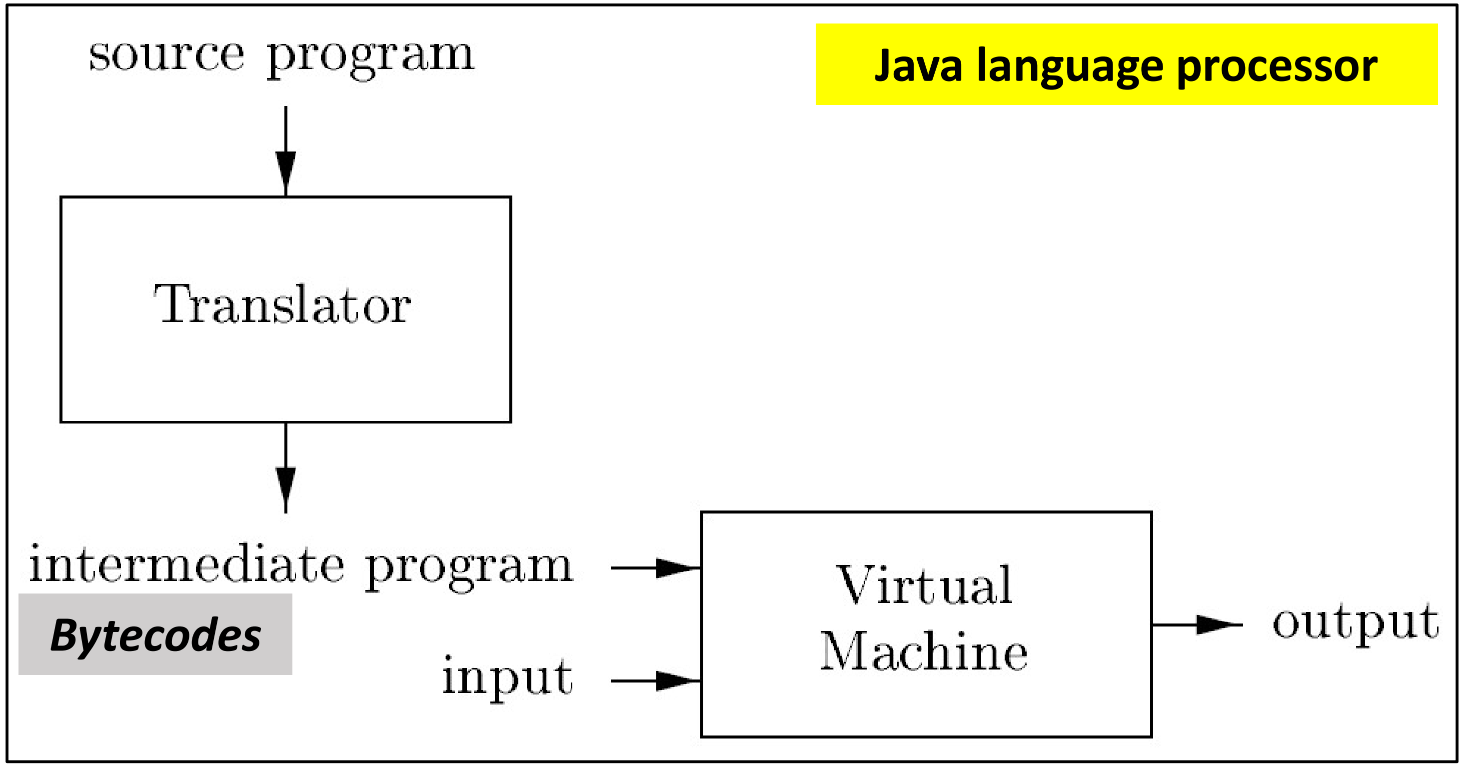
Language Processors

- The *machine-language* target program produced by a *compiler* is usually much ***faster*** than an *interpreter* at mapping inputs to outputs.

- However, an *interpreter* can usually give ***better*** *error diagnostics* than a *compiler*, because it executes the source program statement by statement.

Language Processors

- Example:
  - Combines compilation and interpretation.
  - Called ***Hybrid Compiler.***

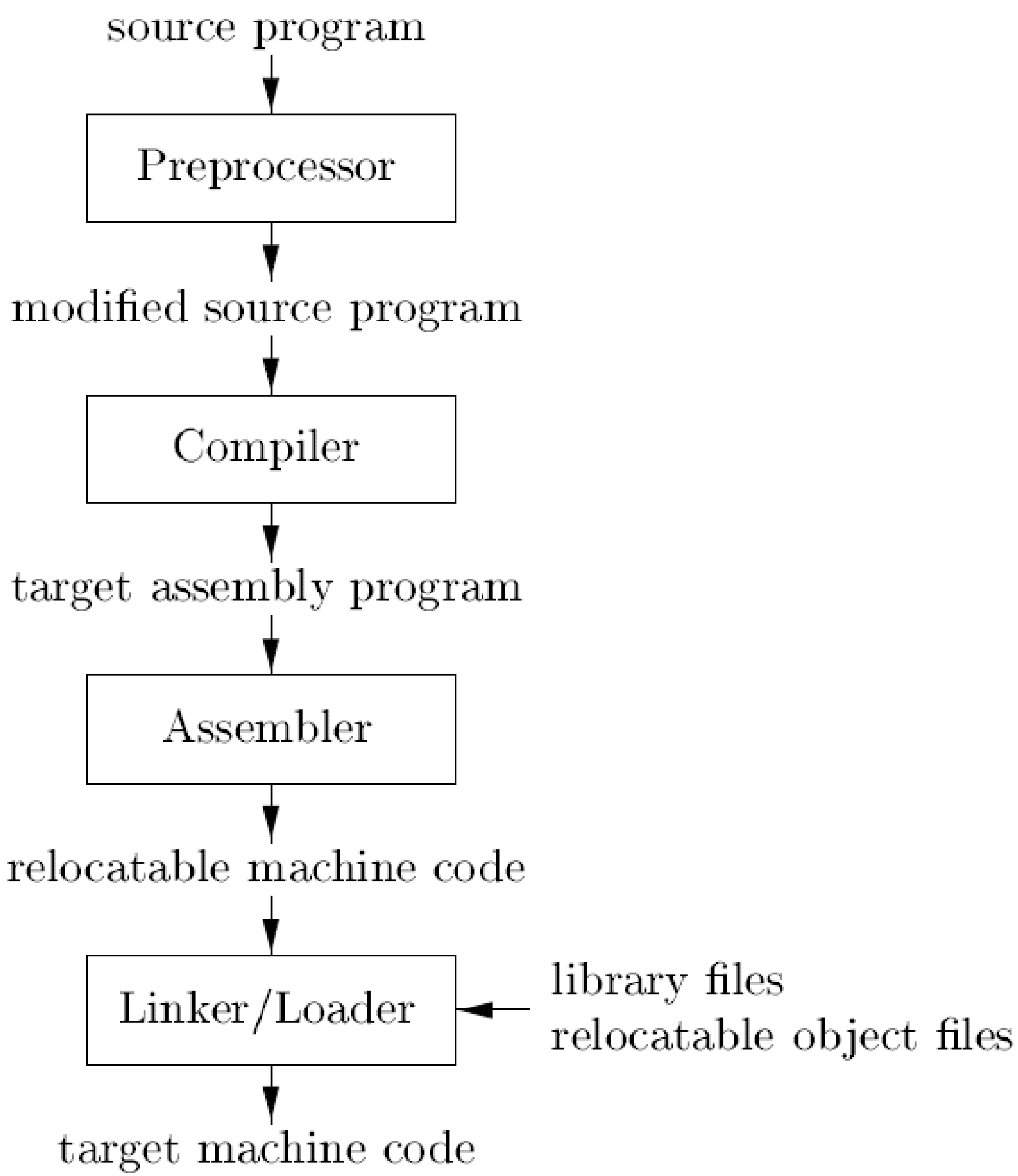


Compilers

General Notes

Language Processors

- In addition to a ***compiler***, several *other* programs may be required to create an *executable target* program.

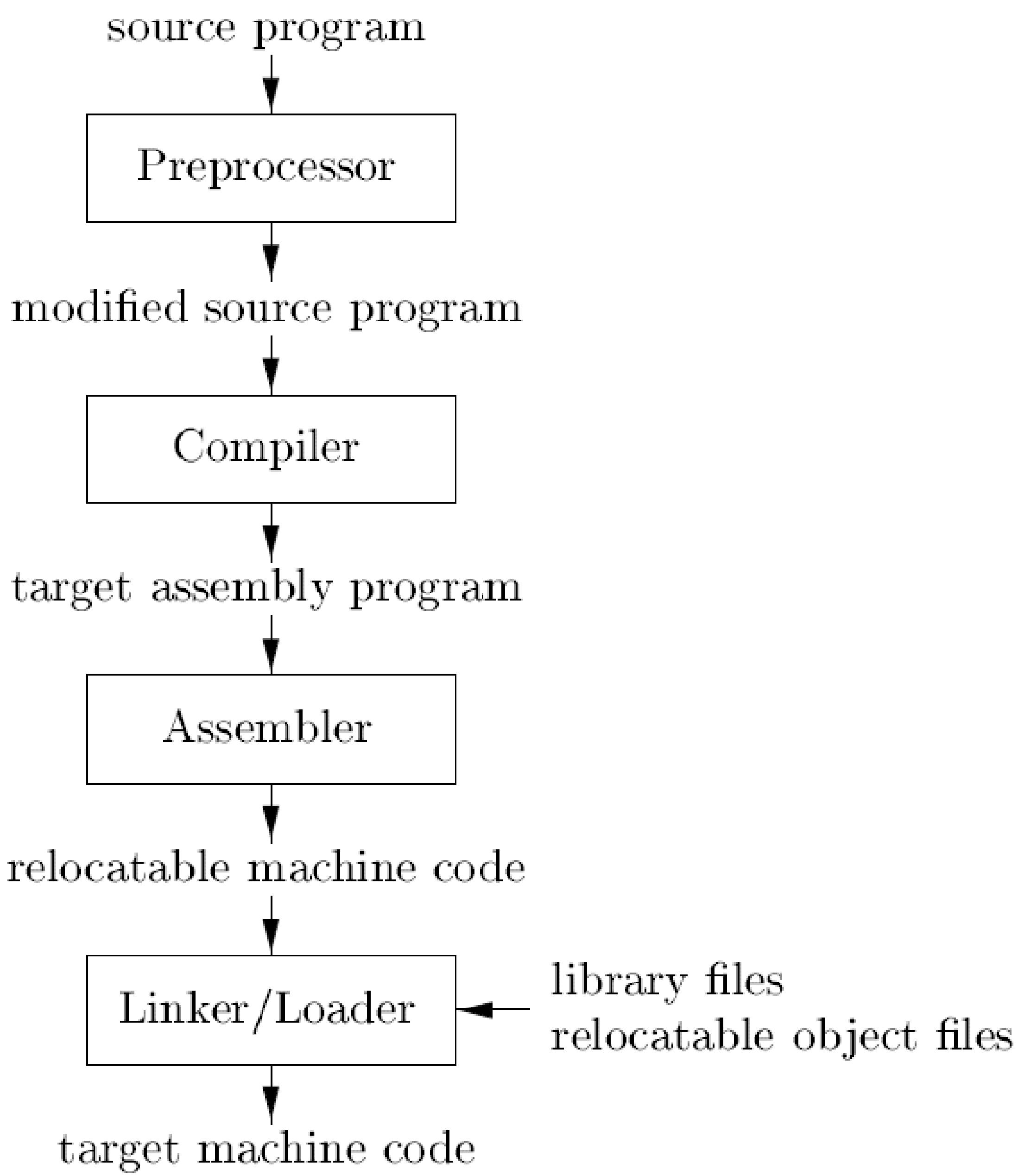


Language Processors

- ***Pre-processor:***

A *source program* may be divided into *modules* stored in separate files. The task of *collecting* the source program is done by the ***pre-processor***.

It also expand ***Macros*** → to source language statements.

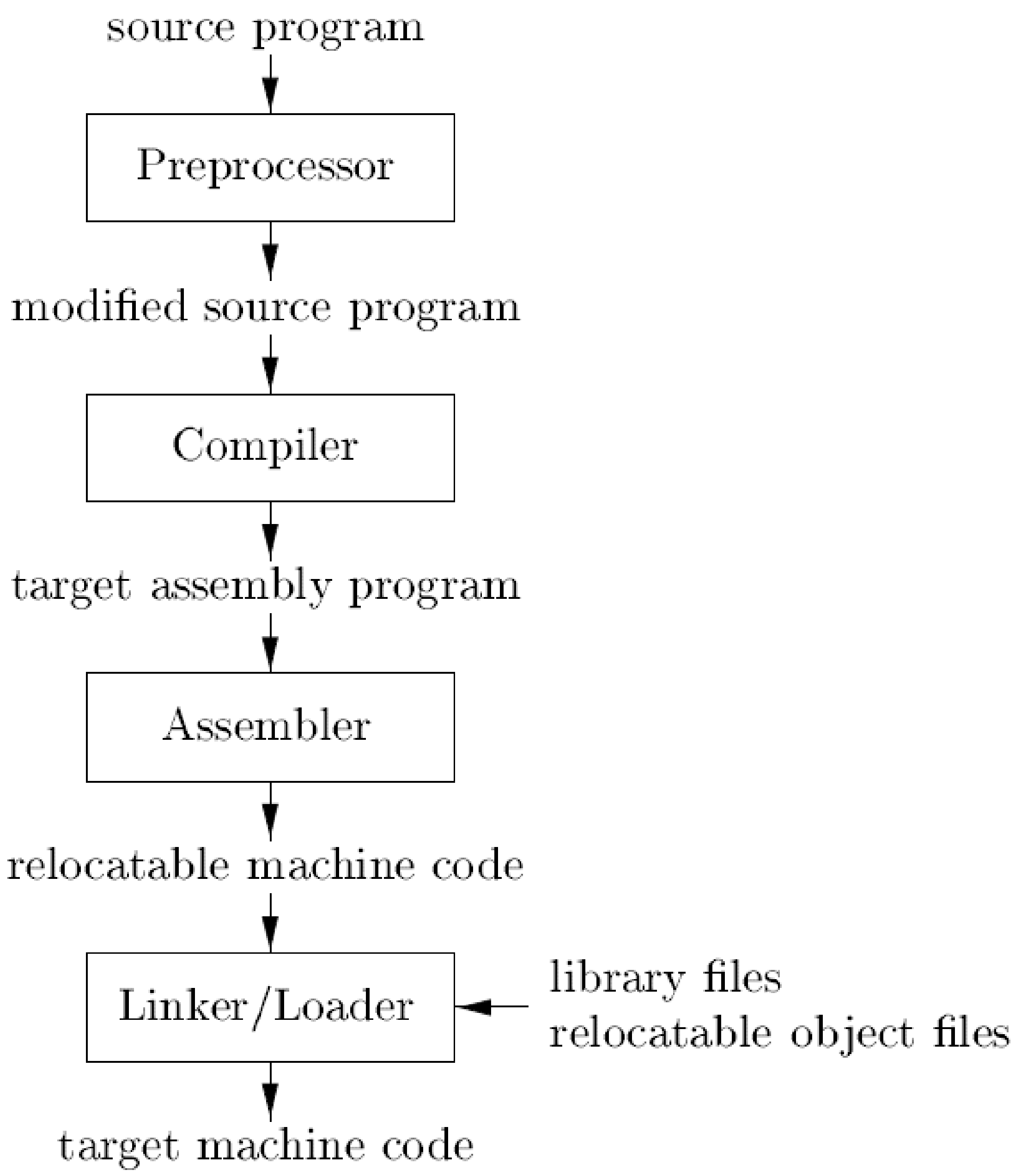


Language Processors

- ***Compiler:***

The ***compiler*** is fed by the modified *source program* and may produce an *assembly-language* program as its output.

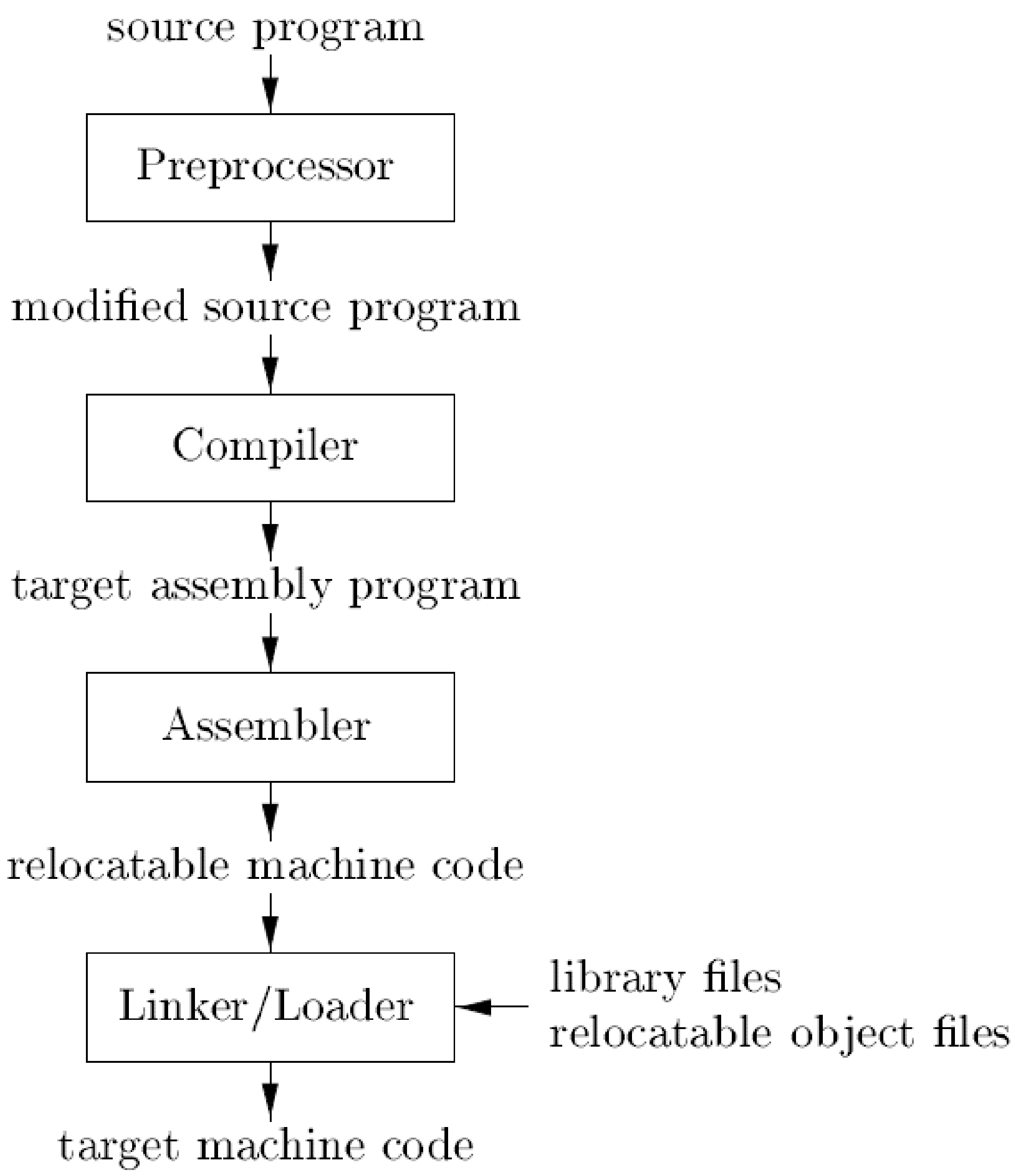
Assembly language is easy to produce and debug.



Language Processors

- ***Assembler:***

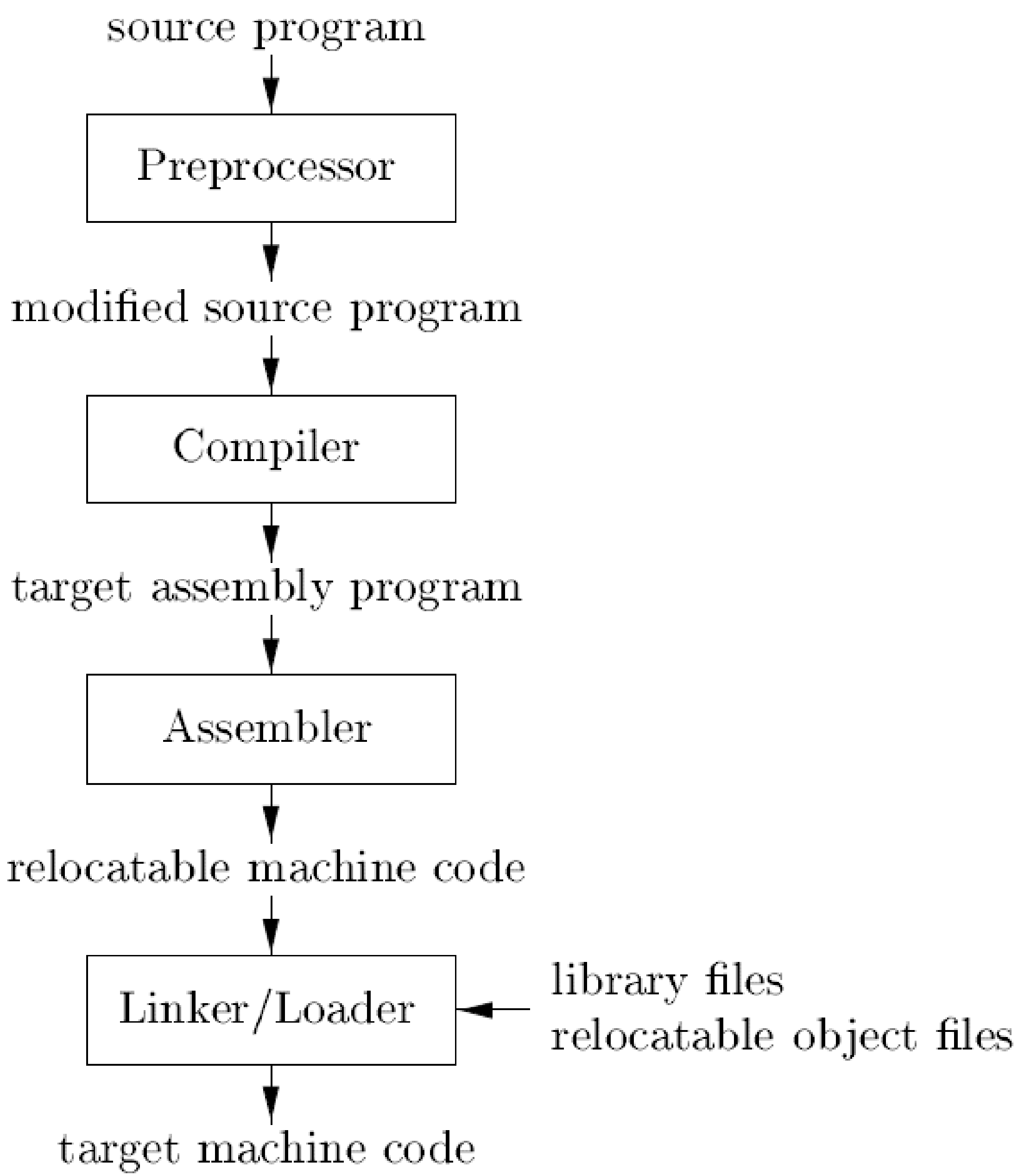
The *assembly language* is then processed by the ***Assembler*** that produces relocatable ***machine code*** as its output.



Language Processors

- ***Linker/Loader:***

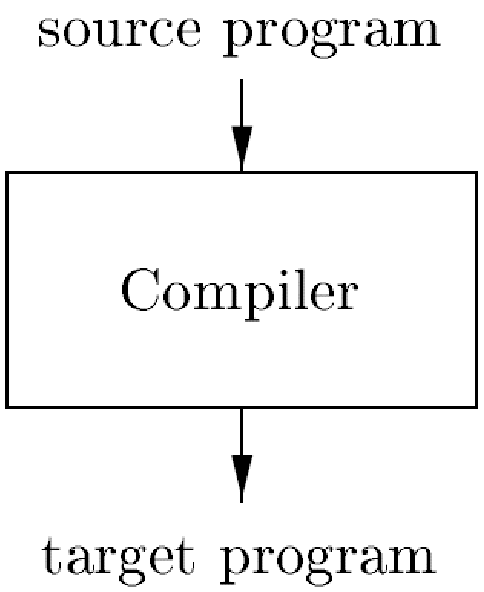
  - Large programs are often compiled in pieces.
  - Relocatable *machine code* may have to be linked together with other relocatable object files & library files into the code that actually runs on the machine.
  - It resolves *external memory addresses*, where the code in one file may refer to a location in another file.
  - The loader then puts together all of the executable object files into memory for execution.



The Structure of a Compiler

- A ***compiler*** as a single box that maps a ***source program*** into a semantically equivalent ***target program***.

- There are two parts to this mapping:
  - **Analysis**, and
  - **Synthesis**.



The Structure of a Compiler

- ***Analysis Part (Front End):***

  1. **Breaks** up the source program into constituent pieces and **imposes** a *grammatical structure* on them, then
  2. **Uses** this structure to create an *intermediate representation* (IR) of the source program.
  3. **Detects** if the source program is either syntactically ill formed or semantically unsound → it must provide **informative messages**.

- Also, it collects *information* about the source program and stores it in a *data structure* called a ***symbol table***, which is passed along with the **IR** to the ***synthesis*** part.

The Structure of a Compiler

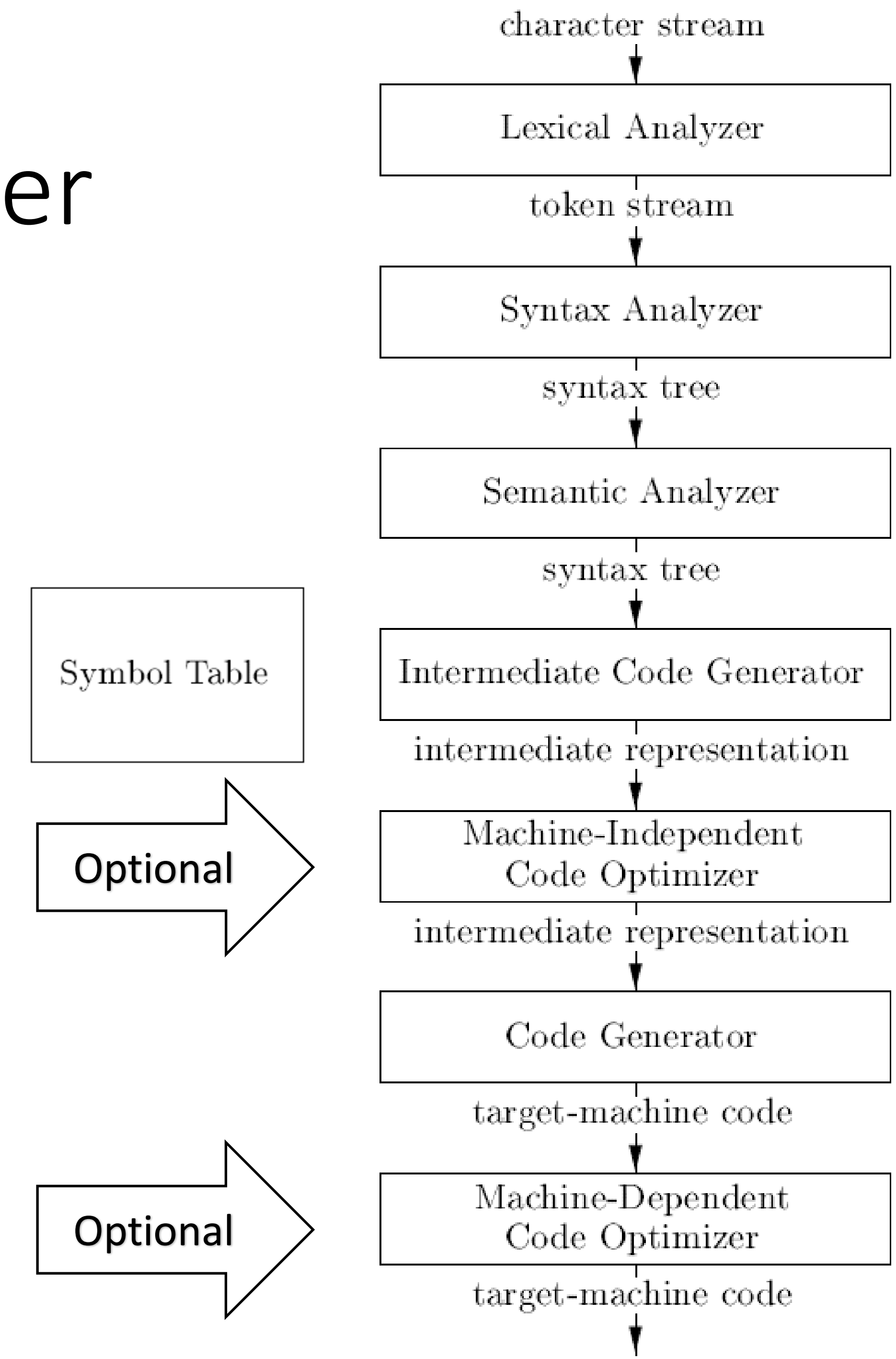
- ***Synthesis Part (Back End):***

  - Is fed by the **IR** and the information in the ***symbol table*** → *constructs* the desired <u>*target program*</u>.

  - Its main tasks are:
    - Instructions selection,
    - Instructions scheduling, and
    - Register allocation.

The Structure of a Compiler

- A compiler operates in ***phases*** each of which transforms one representation of the source program to another.



Lexical Analysis - Scanning

- The *first* phase: ***Lexical Analysis*** or ***Scanning*** carried out by ***Lexical Analyzer.***
- Reads the *stream* of characters making up the <u>*source program*</u> and <u>*groups*</u> the characters into <u>*meaningful*</u> sequences called ***lexemes***.
- For each lexeme, the *analyzer* produces as output a ***token*** of the form:

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

- These tokens are passed to the next phase, i.e. ***Syntax Analysis***.

Lexical Analysis - Scanning

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

- The ***token-name***:
  - Is an abstract symbol used during syntax analysis.
- The ***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***.

Lexical Analysis - Scanning

```
position = initial + rate * 60
```

- The characters in this assignment could be grouped into the following ***lexemes*** and mapped into the following ***tokens*** passed on to the <u>*syntax analyzer*</u>:

Blanks are discarded by the lexical analyzer.

Lexeme	Token
`position`	**`<id, 1>`**
`=`	**`<=>`**
`Initial`	**`<id, 2>`**
`+`	**`<+>`**
`rate`	**`<id, 3>`**
`*`	**`<*>`**
`60`	**`<60>`**

`id` → **`identifier`** is an abstract symbol.

`1, 2, 3` → symbol-table entry. For identifier, it holds info such as <u>*type*</u> and <u>*name*</u>.

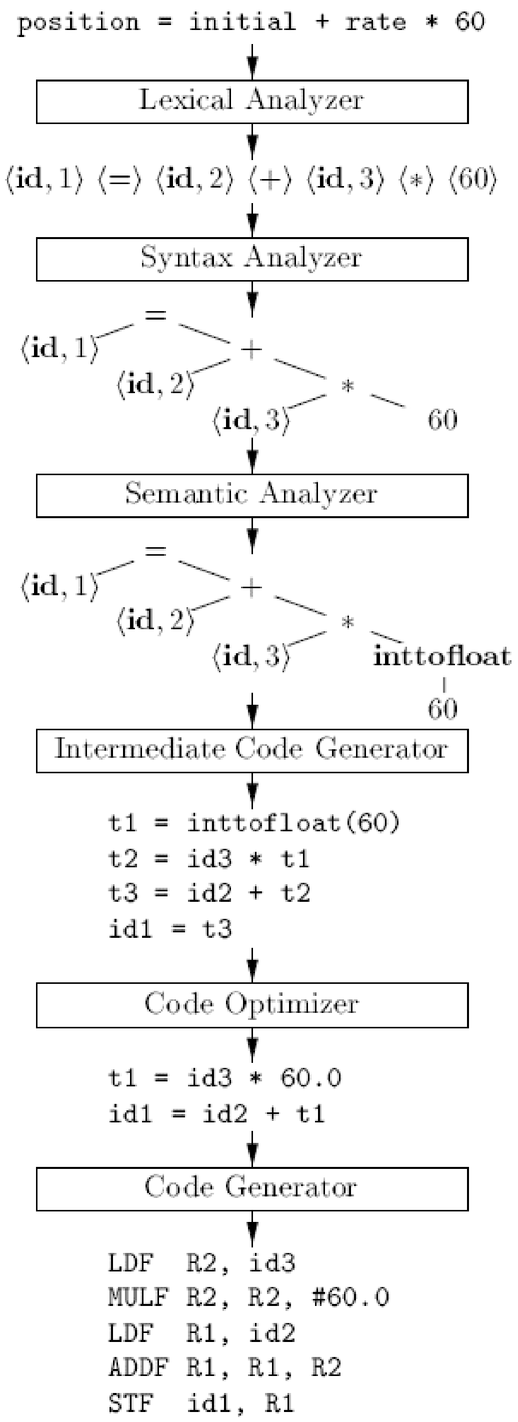
Lexical Analysis - Scanning

- The representation after the lexical analysis is:

$$\langle \mathbf{id}, 1 \rangle \ \langle = \rangle \ \langle \mathbf{id}, 2 \rangle \ \langle + \rangle \ \langle \mathbf{id}, 3 \rangle \ \langle * \rangle \ \langle 60 \rangle$$

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

SYMBOL TABLE



Syntax Analysis - Parsing

- The *second* phase: ***Syntax Analysis*** or ***Parsing*** carried out by ***Syntax Analyzer.***

- The parser uses the *first* components of the *tokens* produced by the lexical analyzer to create a ***tree-like IR*** that depicts the *grammatical structure* of the token stream.

- A typical IR is a ***syntax tree:***
  - interior node represents an operation, and
  - the children of the node represent the arguments of the operation.

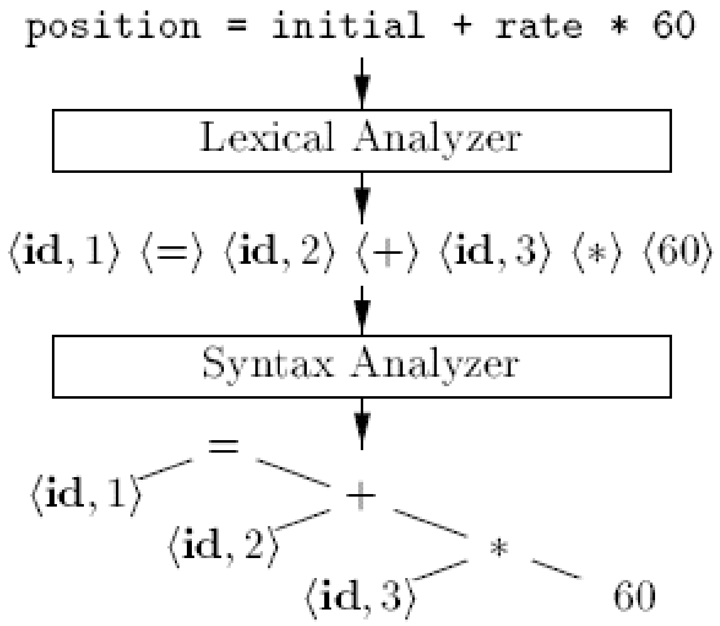
Syntax Analysis - Parsing

- The tree has an *interior* node labelled ***** with **<id, 3>** as its left child and the integer **60** as its right child.

- The node **<id, 3>** represents the identifier **`rate`**. The node labelled ***** makes it explicit that we must first multiply the value of **`rate`** by **60**.

- The node labelled **+** indicates that we must add the result of this multiplication to the value of **`initial`.**

- The root of the tree, labelled **=**, indicates that we must store the result of this addition into the location for the identifier **`position`**.



Semantic Analysis

- The *third* phase: ***Semantic Analysis***

- Uses the <u>*syntax tree*</u> and the *information* in the <u>*symbol table*</u> to check the source program for <u>***semantic consistency***</u> with the language definition.

- Gathers <u>*type information*</u> and saves it in either the <u>*syntax tree*</u> or the <u>*symbol table*</u>, for subsequent use during <u>*intermediate-code generation*</u>.

Semantic Analysis

- An important part of semantic analysis is ***type checking***, where the compiler checks that each *operator* has *matching operands*.
  - E.g.: many programming language definitions require an array index to be an integer; the compiler must report an *error* if a floating-point number is used to index an array.

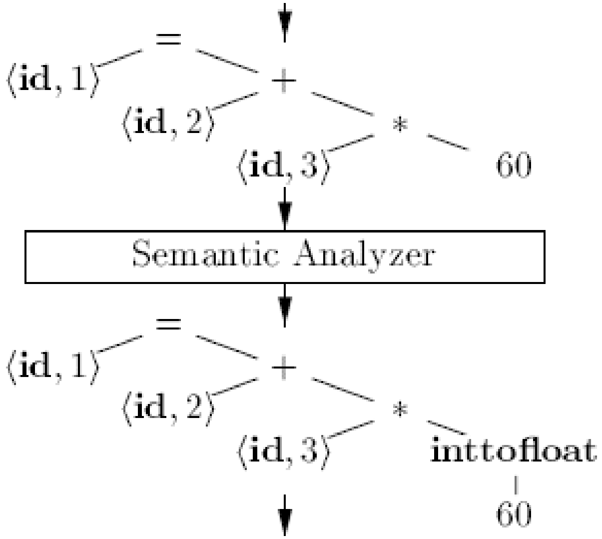
- The language specification may permit some type conversions called ***coercions***.
  - E.g.: a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers.
    - If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Semantic Analysis

Example:

If the `position`, `initial`, and `rate` have been declared to be floating-point numbers, and that the lexeme **`60`** by itself forms an integer, then

- The *type checker* in the *semantic analyzer* discovers that the operator is applied to a floating-point number **`rate`** and an integer **`60`** → the integer may be converted into a floating-point number.



Intermediate Code Generation

- After <u>*syntax*</u> and <u>*semantic*</u> analysis of the source program, many compilers generate an explicit ***low-level*** or ***machine-like*** IR (a kind of a program for an *abstract machine*.)

- IR Properties:
    1. Easy to produce.
    2. Easy to translate into the target machine.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Three-address code: 1) one operator on the right side, 2) a temporary name to hold the value computed by a 3-address instruction, 3) some instructions have less than three operands.

Code Optimization

- *Machine-independent* phase.
- Improve the intermediate code so that *better (faster) target* code will result.
- Other objectives: shorter code, target code with less power consumption.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

1) **`inttofloat`** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0.
2) **`t3`** is used only once to transmit its value to **`id1`**

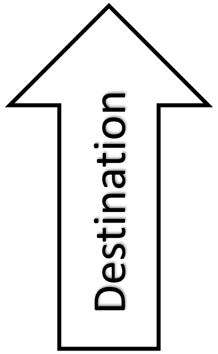
```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation

- It takes as input an ***IR*** 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.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.
- A crucial aspect of code generation is the judicious assignment of registers to hold variables.

Code Generation

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```



- **F** is each instruction means dealing with floating point numbers.
- **`id3, id2, id1`** are addresses in the memory.

Compilers

General Notes

Symbol-Table Management

- A compiler *records* the ***variable names*** used in the source program and collects ***information*** about various ***attributes*** of each name:
  - The *storage* allocated for a name, its *type*, its *scope.*
  - In the case of procedure names, such things as the *number and types* of its *arguments*, the *method of passing* each argument, and the *type returned*.

- The ***symbol table*** is a *data structure* containing a record for each *variable name*, with fields for the *attributes* of the name.

- The data structure should be designed to allow the compiler to ***find*** the record for each name quickly and to ***store*** or ***retrieve*** data from that record quickly.

The Grouping of Phases into Passes

- The discussion of phases deals with the ***logical organization*** of a compiler.
- In an implementation, activities from *several phases* may be ***grouped*** together into a ***pass*** that reads an input file and writes an output file.

  - The *front-end* phases: *lexical analysis*, *syntax analysis*, *semantic analysis*, and *intermediate code generation*, might be grouped together into ***one pass***.
  - *Code optimization* might be an ***optional pass***.
  - Then there could be a *back-end* ***pass*** consisting of *code generation* for a particular target machine.

The Grouping of Phases into Passes

- Some compiler collections have been created around carefully designed IRs that allow the ***front end*** for a <u>*particular language*</u> to interface with the ***back end*** for a <u>*certain target machine*</u>.

- Thus, we can produce compilers for *different source languages* for *one target machine* by combining different front ends with the back end for that target machine.

- Similarly, we can produce compilers for *different target machines*, by combining a front end with back ends for different target machines.

Compiler-Construction Tools

- *General software-development tools:*
  - Language editors,
  - Debuggers,
  - Version mangers,
  - Profilers,
  - Test harnesses,
  - So on…

- *Specialized software-development tools:*
  - Use:
    - *Specialized languages* for specifying and implementing specific components
    - Quite *sophisticated algorithms*.
  - The most successful tools *hide* the details of the generation algorithm and *produce* components that can be easily integrated into the remainder of the compiler.

Compiler-Construction Tools

Parser generators: automatically produce syntax analyzers from a grammatical description of a programming language.

Scanner generators: produce lexical analyzers from a regular-expression description of the tokens of a language.

Syntax-directed translation engines: produce collections of routines for walking a parse tree and generating intermediate code.

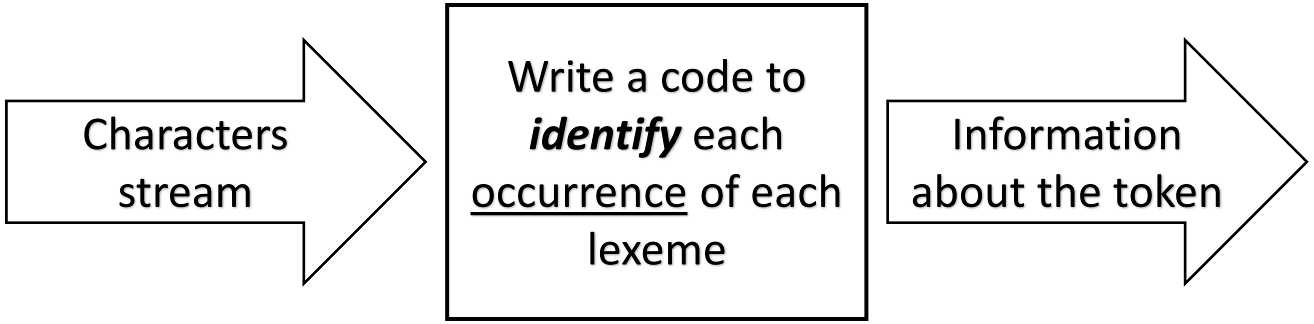
Code-generator generators: produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

Data-flow analysis engines: facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

Compiler-construction toolkits: provide an integrated set of routines for constructing various phases of a compiler.

Lexical Analysis

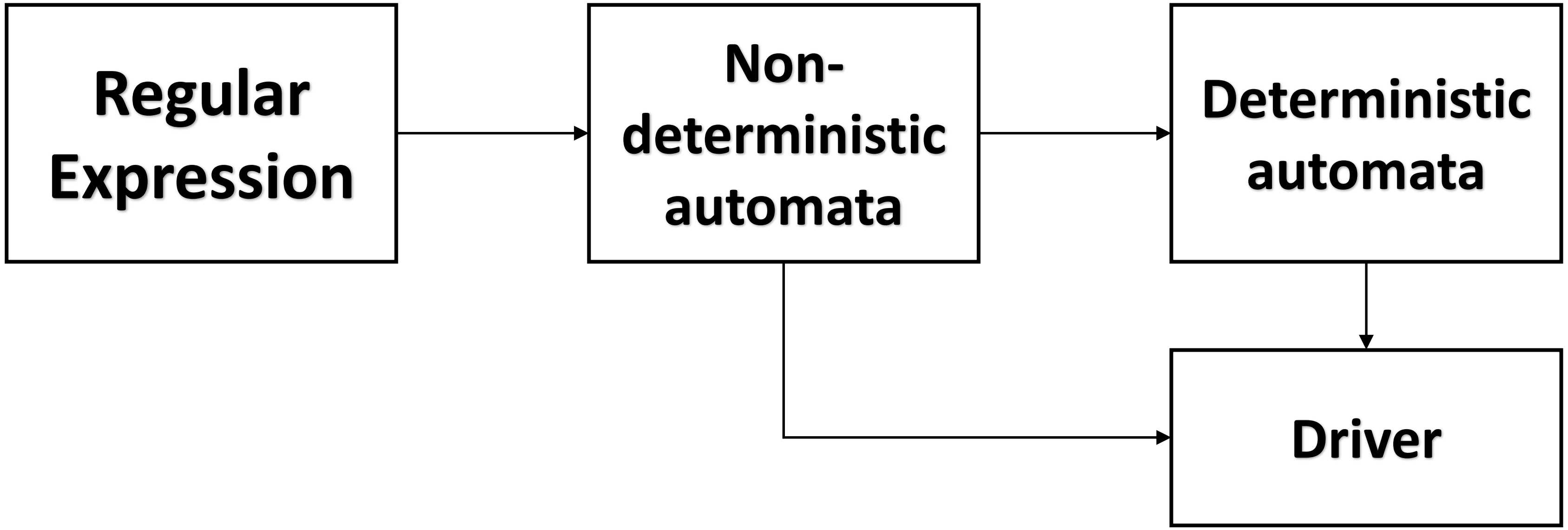
- To implement a lexical analyzer, we need to have a ***description*** for the ***lexemes*** of each token.



- Use a *lexical-analyzer generator* that takes the lexemes patterns. An example of this is *Lex* (*Flex*).

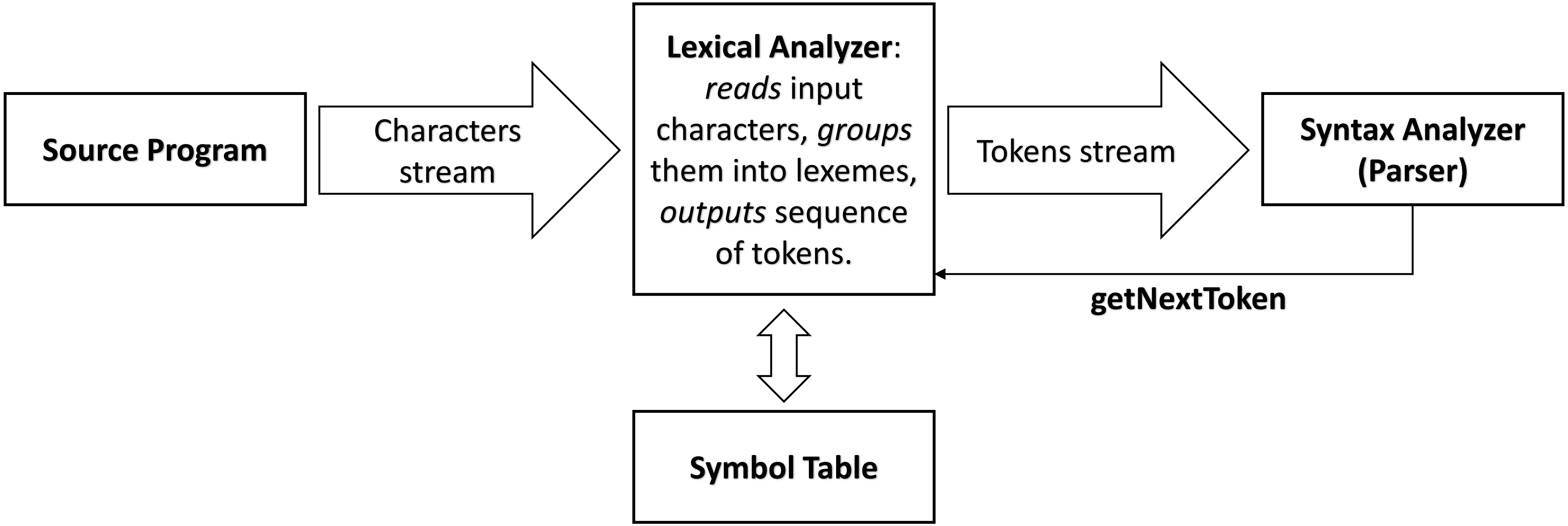
Lexical Analysis

- To specify lexeme patterns, a convenient notation, called ***Regular Expression***, is used.



A driver is a code which simulates the automata and use them as a guide to determining the next token. It forms the nucleus of the lexical analyzer

The Role of the Lexical Analyzer



When the lexical analyzer discovers a lexeme constituting an *identifier*, it enters that lexeme.

The Role of the Lexical Analyzer

- It may perform certain other tasks besides identification of lexemes:
  - Stripping out comments and whitespace (blank, newline, tab, etc.
  - Correlating error messages generated by the compiler with the source program by tracking the number of newline characters seen.

Scanning: simple processes that don't require tokenization, such as deletion of comments and consecutive whitespace compaction.

→

Lexical analysis: proper is the more complex portion, which produces *tokens* from the output of the scanner.

Lexical Analysis vs. Syntax Analysis

- Why having two phases: Lexical Analysis and Syntax Analysis:
    1. Simplicity of design
    2. Compiler efficiency is improved
    3. Portability is enhanced

Tokens, Patterns, and Lexemes

- ***A token***:
  - Is a pair consisting of a *token name* and an optional *attribute value*.
  - The *token name*: is an abstract symbol representing a kind of lexical unit.
    - E.g., a particular *keyword*, or a sequence of input characters denoting an *identifier*. The token names are the input symbols that the parser processes.

Tokens, Patterns, and Lexemes

- ***A pattern***:
  - Is a *description* of the form that the lexemes of a token may take.
  - In the case of a *keyword* as a token, the pattern is just the sequence of characters that form the keyword.
  - For *identifiers* and some other tokens, the pattern is a more complex structure that is matched by many strings.

Tokens, Patterns, and Lexemes

- ***A lexeme***:
  - A *sequence of characters* in the source program that matches the ***pattern*** for a token and is identified by the *lexical analyzer* as an *instance* of that token.

Tokens, Patterns, and Lexemes

- Example:

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

```
printf("Total = %d\n", score);
```

Tokens, Patterns, and Lexemes

- In many programming languages, the following classes cover most or all of the tokens:
  - One token for each *keyword*.
  - Tokens for the *operators*, either individually or in classes
  - One token representing all *identifiers*.
  - One or more tokens representing *constants*, such as numbers and literal strings.
  - Tokens for each *punctuation symbol*, such as **( ) , : ;**

Attributes for Tokens

- If more than one lexeme matches a pattern, the lexical analyzer must provide additional information about the particular lexeme that matched.

- E.g. if the pattern for token <u>*number*</u> matches 0 and 1, then it is important for code generator to know which lexeme was found in the source program.

Lexeme → *<token-name, attribute-value>*

Attributes for Tokens

- For the token ***id*** *(identifier)*, we need to associate with it a great deal of information, such as:
  - its lexeme,
  - its type,
  - and the location at which it is first found (in case an error message about that identifier must be issued)

- This information is kept in the *symbol table*.
- The appropriate *attribute-value* for ***id*** is a pointer to the *symbol table* entry for that identifier.

Attributes for Tokens

- Example: `E = M * C ** 2`

attribute-value

<**id**, pointer to symbol-table entry for `E`>

No attribute-value <**assign_op**>

<**id**, pointer to symbol-table entry for `M`>

<**mult_op**>

<**id**, pointer to symbol-table entry for `C`>

<**exp_op**>

<**number**, integer value 2>

Input Buffering

Reading the source program:

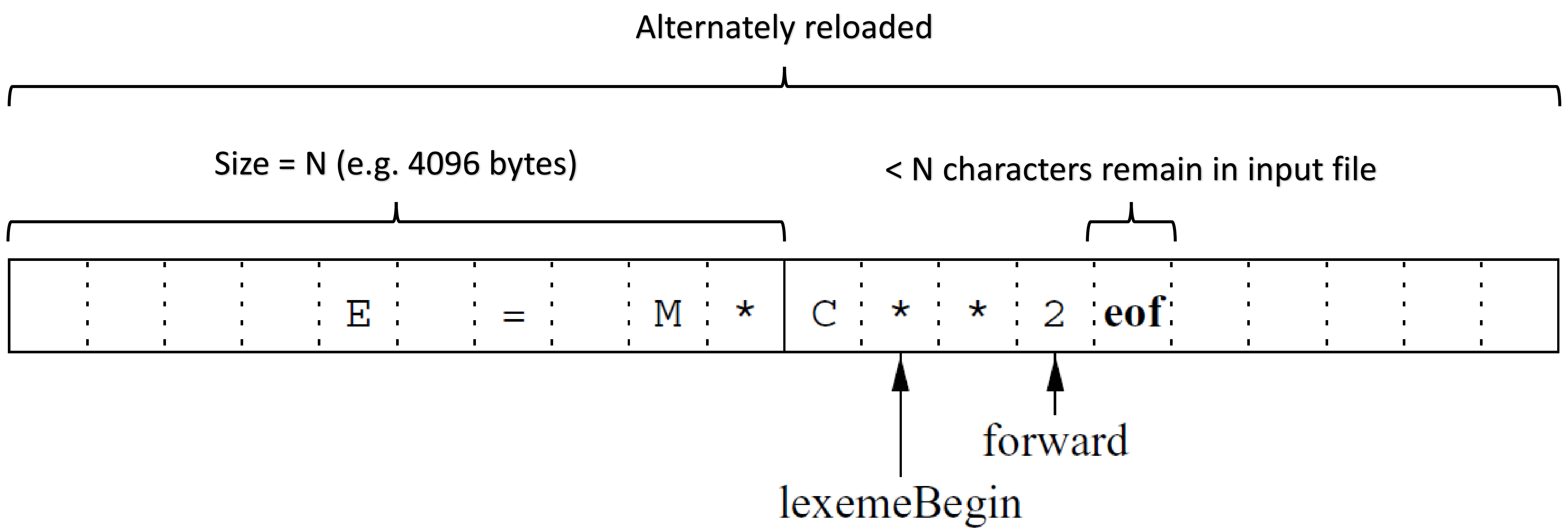
- ***<u>Fact:</u>*** we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.

- E.g. we cannot be sure we've seen the end of an ***identifier*** until we see a character that is not a letter or digit, and therefore is not part of the lexeme for ***id***.

- In C, operators like **-**, **=**, or **<** could also be the beginning of a two-character operator like **->**, **==**, or **<=**.

Buffer Pairs

- A specialized buffering techniques to reduce the amount of overhead required to process a single input character.



Buffer Pairs

- Pointer **`lexemeBegin`**: marks the beginning of the current lexeme
- Pointer **`forward`**: scans ahead until a pattern match is found.