Chapter 6

Programming Languages and their Translators

6.1 Interpreters versus Compilers

There are two general kinds of programming language translators:

- **Interpreters**:

  transform programs directly into command ("behaviour") sequences that run on a **virtual machine**

  + good for *rapid prototyping* of experimental software

  + performs error checking at runtime

- **Compilers**:

  translate programs into low-level machine code, which can then run on an **actual machine**

  + compiled machine code is generally faster (vs. interpreted)

  - compilation itself takes time

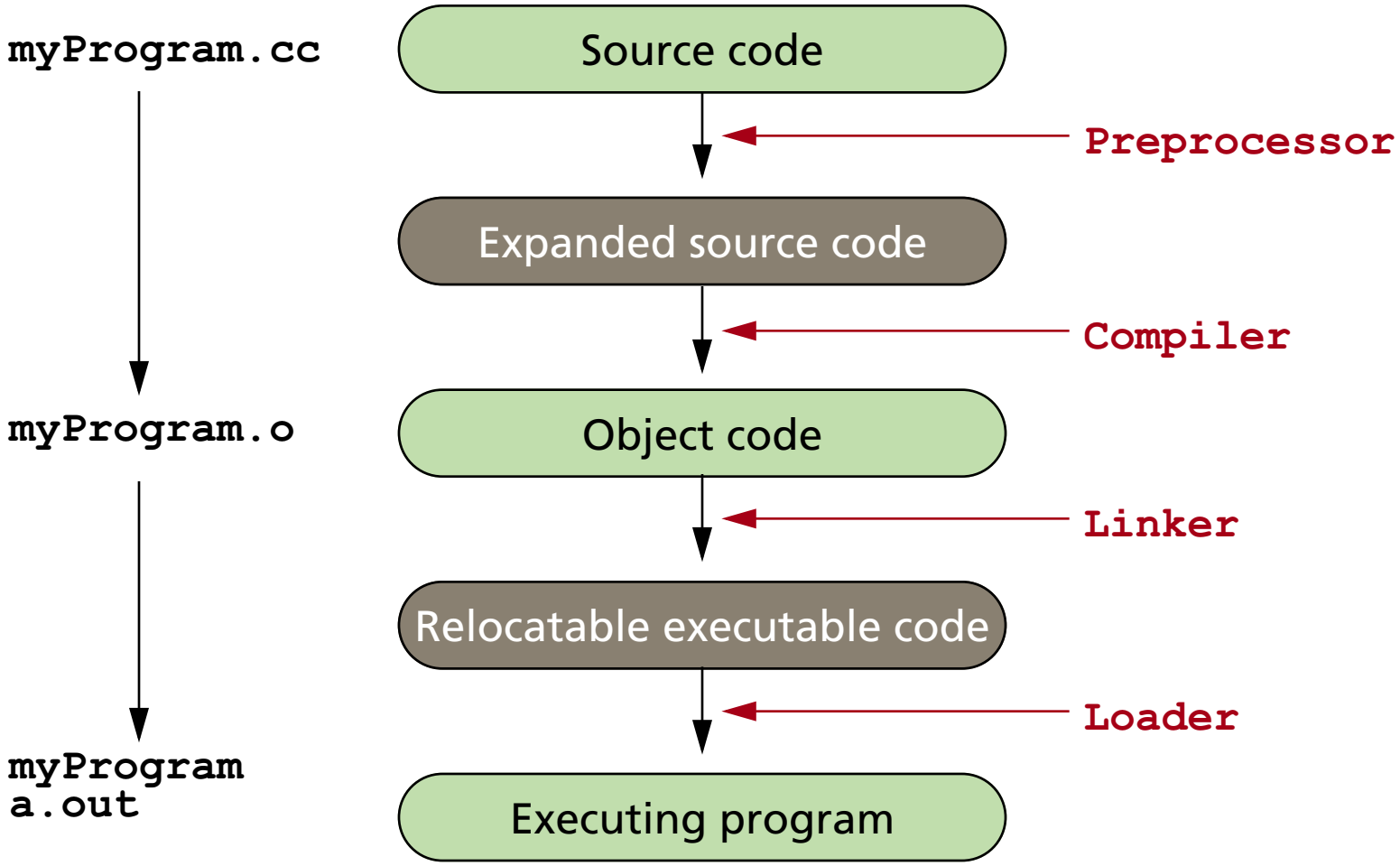
  - to enable error checking *before* runtime, programmers have to provide more information (e.g., type information)

6.2 How Compilers Produce Machine Code

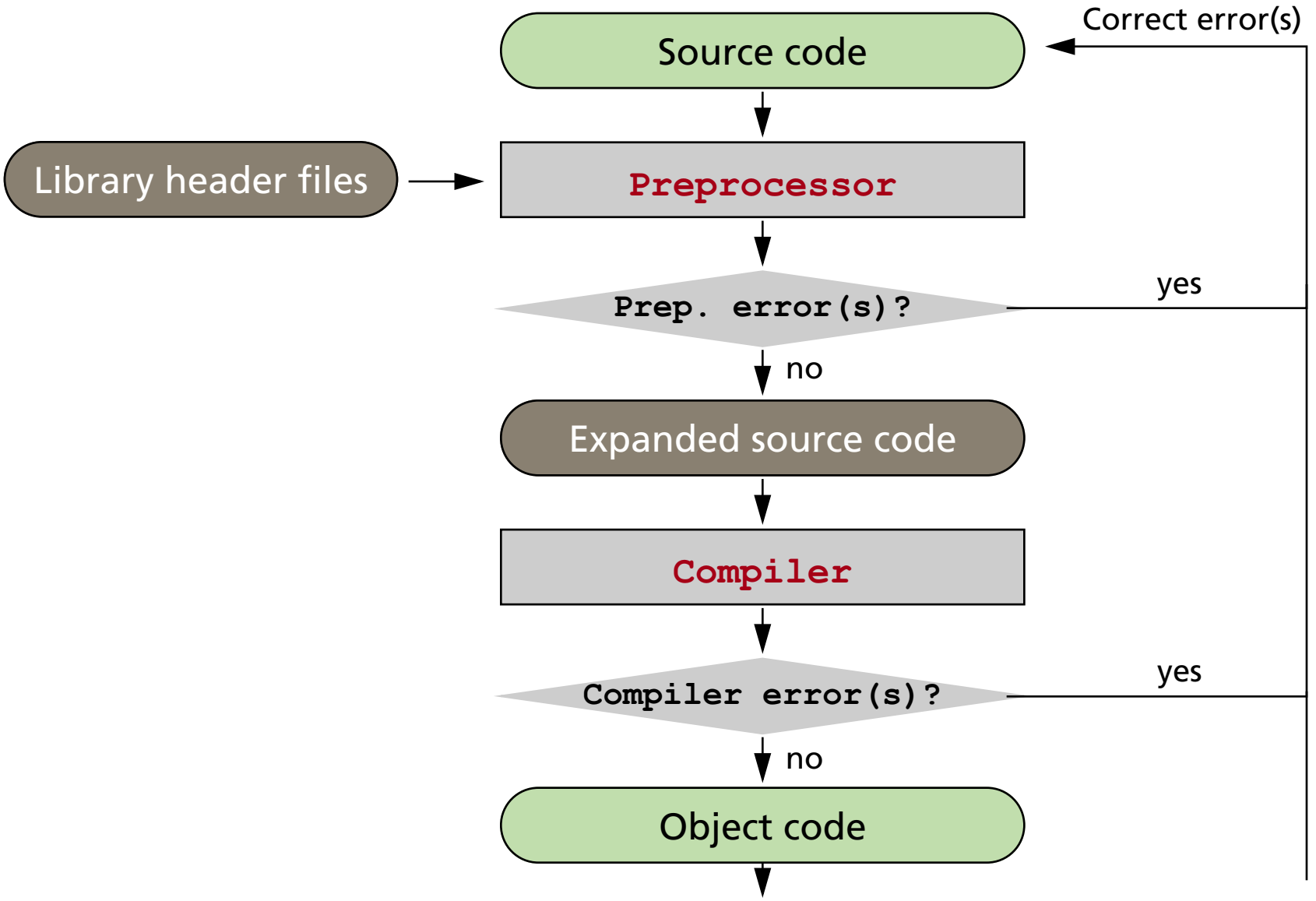
An example C++ program, ready to be compiled:

```
#include <iostream.h>

void main()
{
  int a;
  float b, c;

  a = 2; b = 3.1415;
  c = a * b;

  cout << c << endl;
}
```

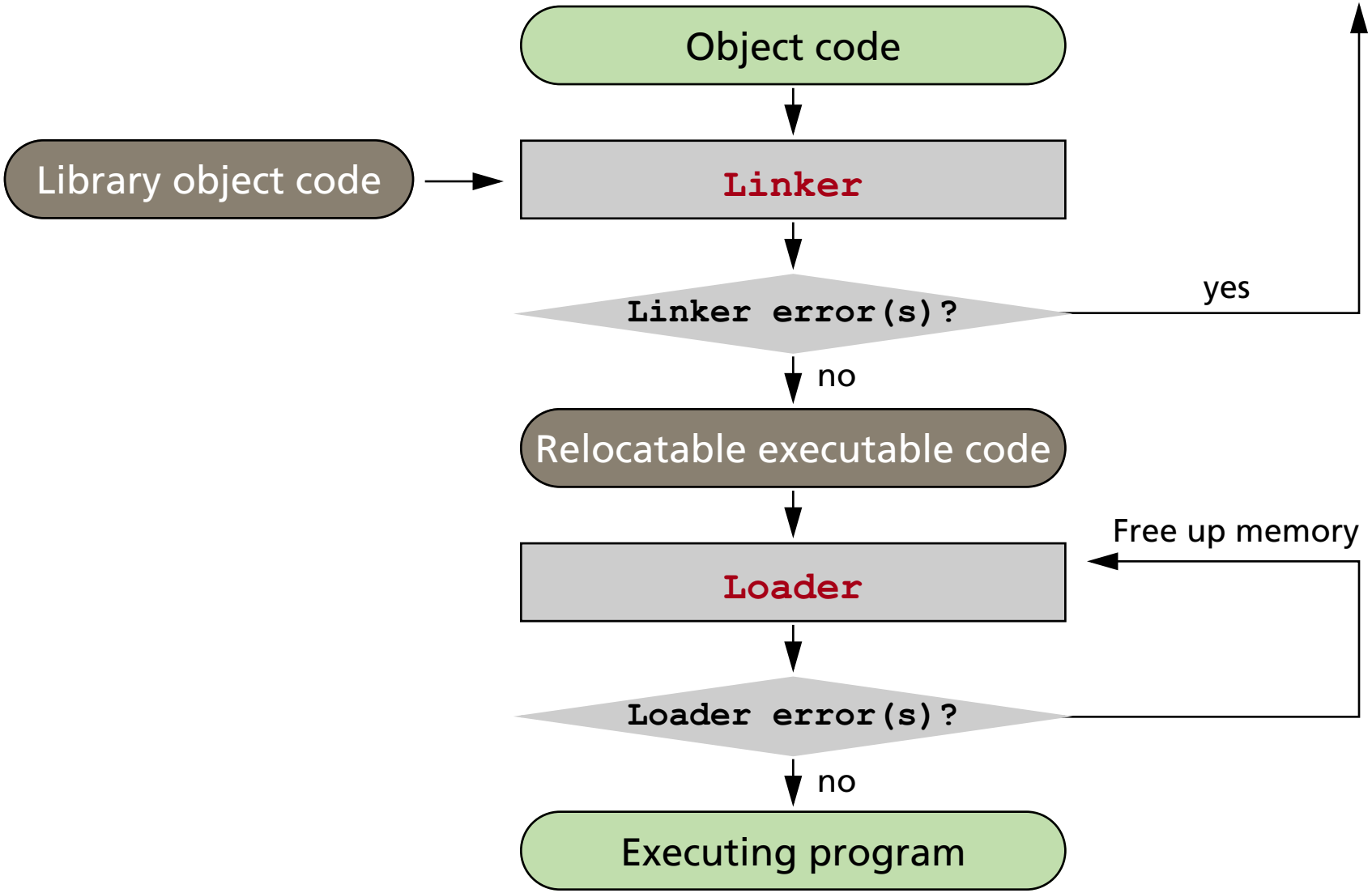
6.2.1 Stages of Translating a C++ Program



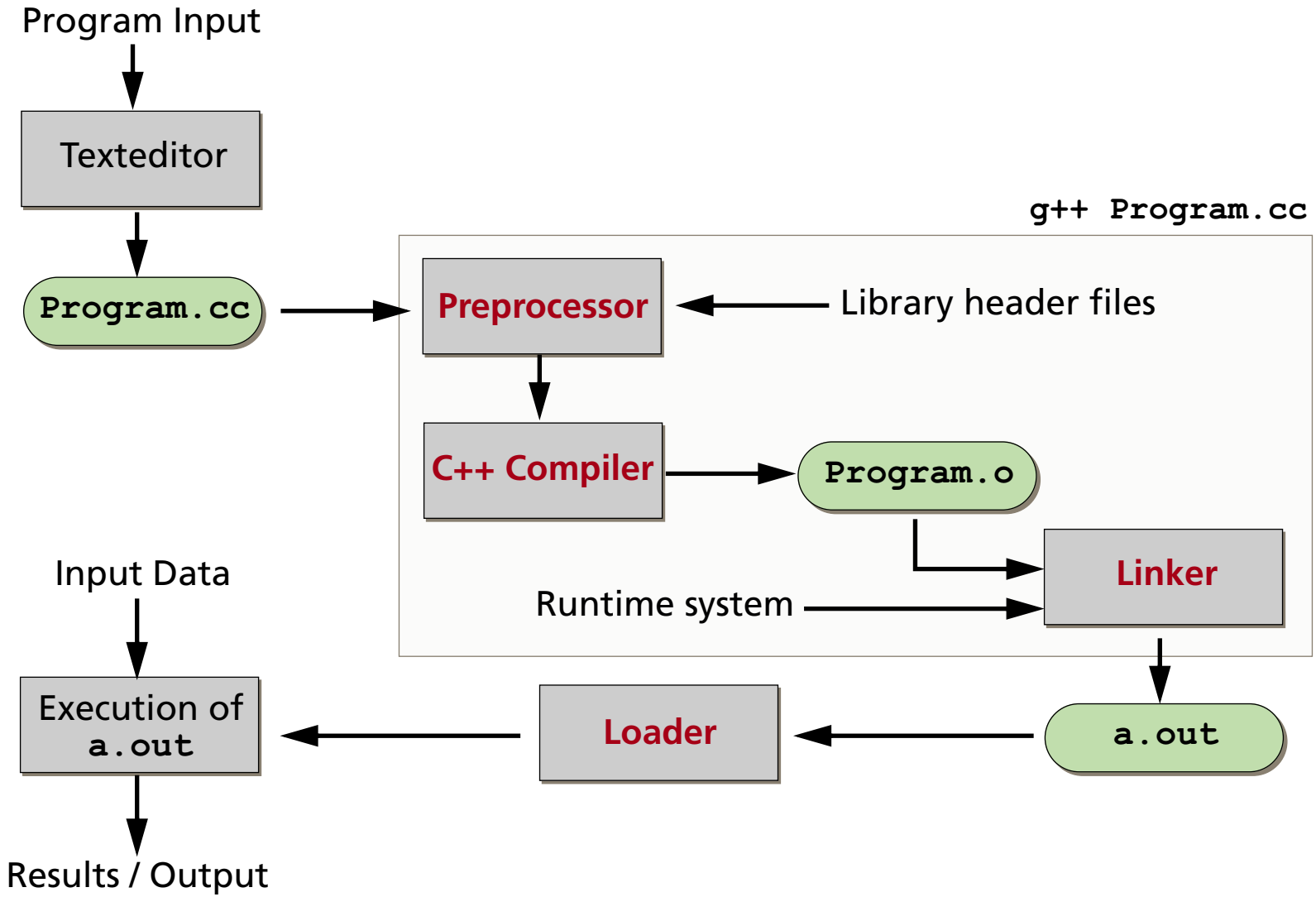
Stages of Translating a C++ Program (Part 1)



Stages of Translating a C++ Program (Part 2)



An Alternative View of the Translation Stages



- **Preprocessor**:
  - Looks for compiler directives (e.g., `#include <streamio.h>`).
  - Inserts predefined code and macros from the include files.

  ⇒ Produces **expanded source code**.

- **Compiler**:
  - Checks the expanded code for syntax errors (*syntax analysis*).
  - Checks for missing variable declarations (*semantic analysis*).
  - Checks for missing type information (*semantic analysis*).
  - … and much more (see for details later!)

  ⇒ Produces **object code**.

- **Linker**:
  - Links the object code with other code required for the program to run (library object code; e.g., specific code for input/output, mathematical functions, etc.)

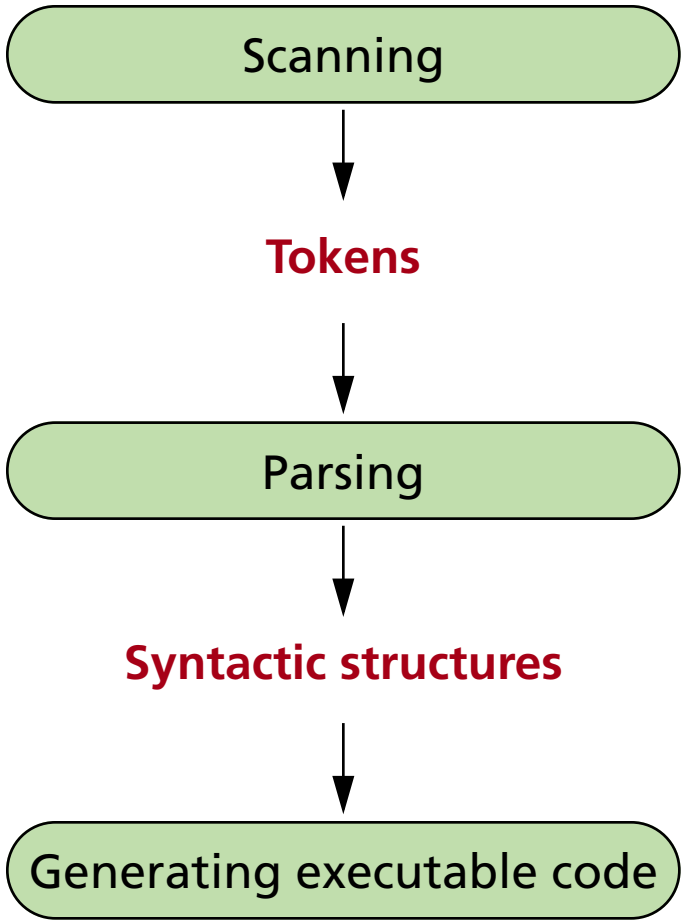
    ⇒ Produces **relocatable executable code** with relative addressing.

- **Loader**:
  - Loads executable code into memory.
  - The actual starting address of the program code is called **base address.**
  - All addresses within the program are **relative addresses** with respect to the base address.

    ⇒ Runs the program.

6.2.2 Components of a Compiler

A simple view of a compiler



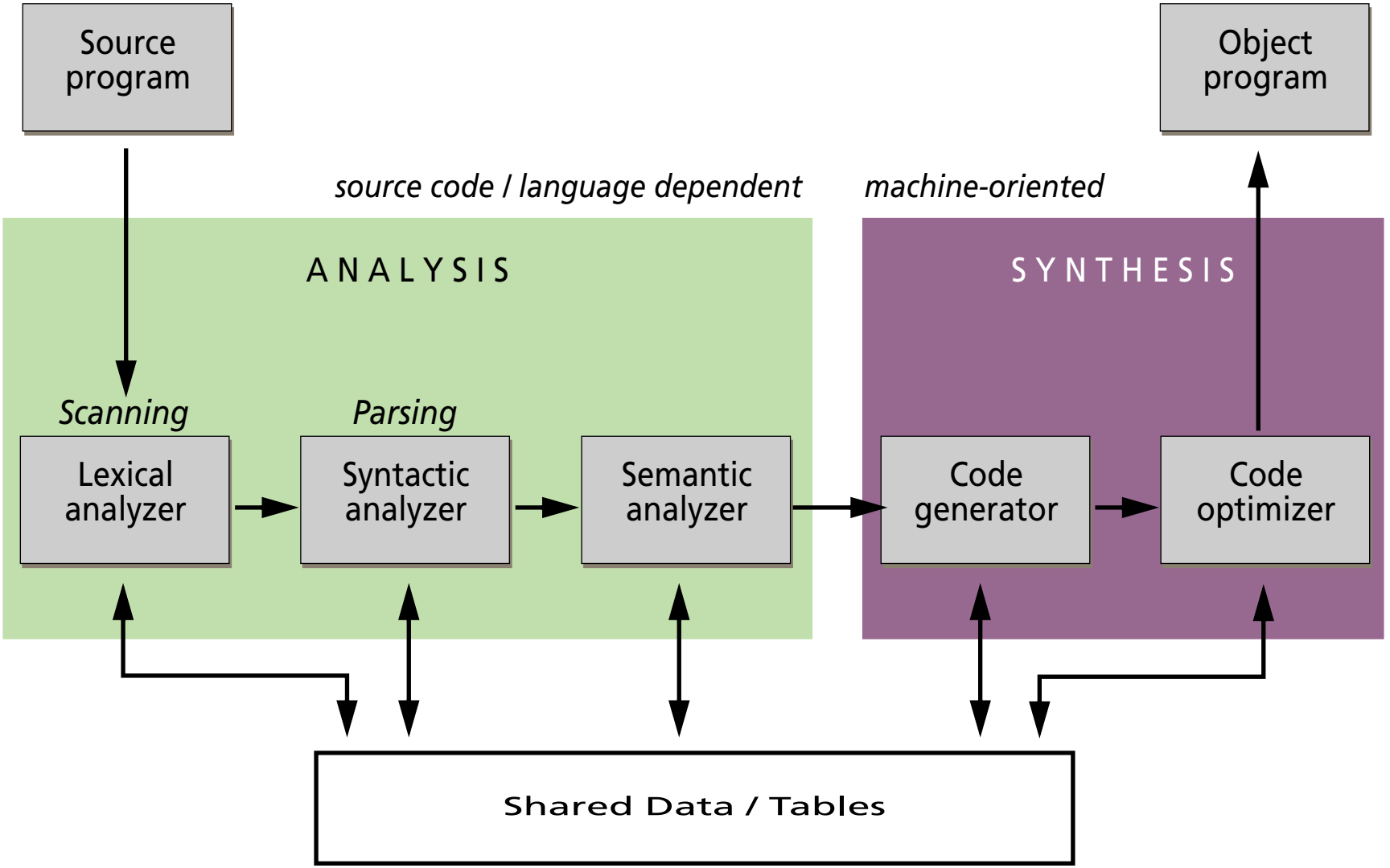
Identify the "words" and punctuation marks.

Identify the "sentences"

and their structures

⇒ meaning (semantics)

A more detailed view of a compiler



6.3 Describing Syntactic Structures

6.3.1 The Backus-Naur Form (BNF)

The BNF[1] was invented in the late 1950s to describe syntactic structures of programming languages (ALGOL 60) and syntactic structures in general.

The BNF works like a word **replacement system**:

a → b | c | a a

Applying this rule (once per line) produces a sequence of words:

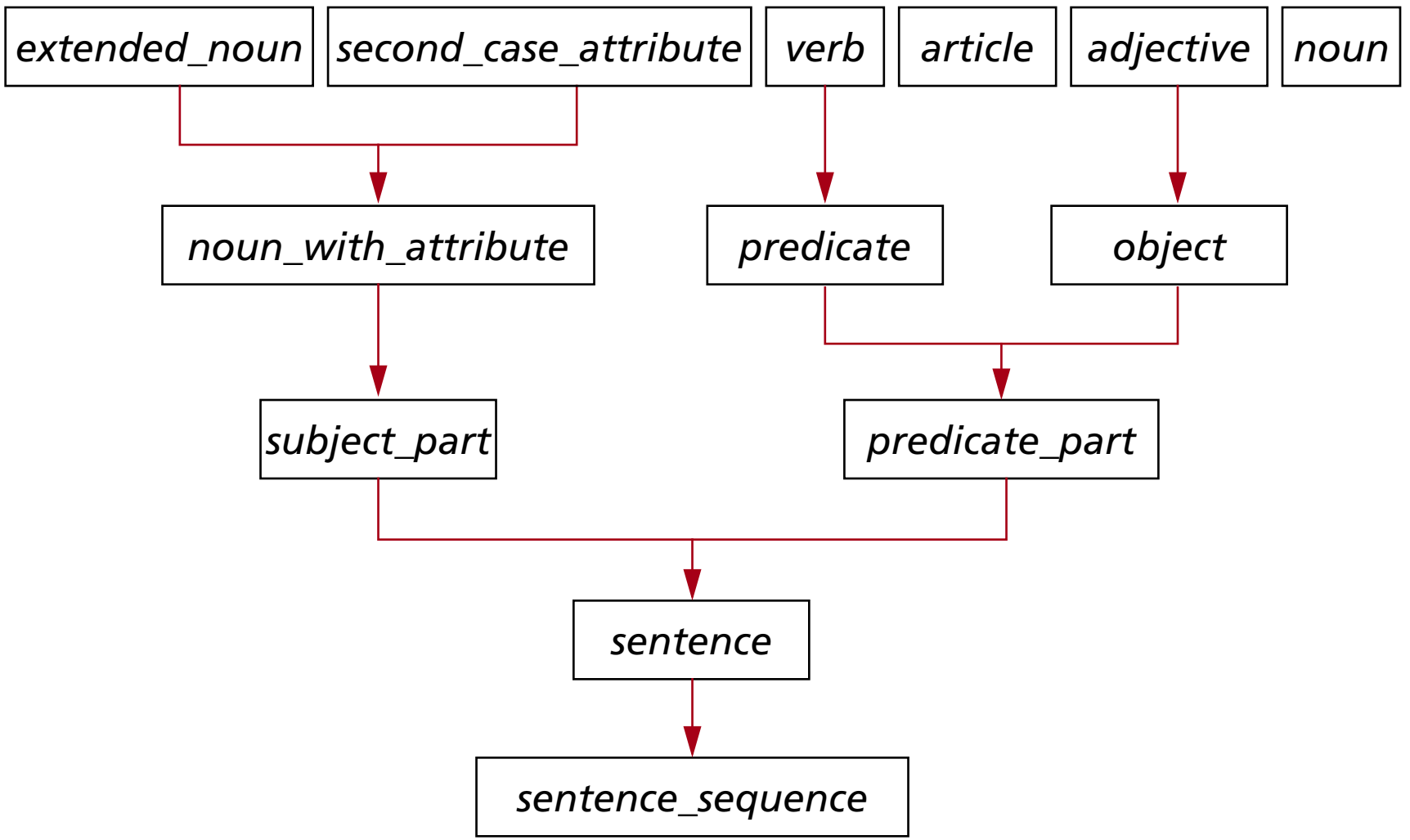
a
a a
a a a (This is just one possibility!)
b c b

1. John Backus and Peter Naur were members of an international committee to develop a precise notation for describing syntax.

Example (1): Description of English sentence structures

sentence_sequence	→	*sentence* . *sentence_sequence*
	|	*sentence* .
sentence	→	*subject_part predicate_part*
subject_part	→	*extended_noun*
	|	*noun_with_attribute*
noun_with_attribute	→	*extended_noun*
	|	*second_case_attribute*
predicate_part	→	*predicate object*
predicate	→	*verb*
object	→	*article noun*
	|	*article adjective noun*

extended_noun	→	**The students**
	|	**The teaching assistants**
second_case_attribute	→	**of CPSC 231**
	|	**of CPSC 233**
verb	→	**like**
	|	**enjoy**
article	→	**the**
adjective	→	**weekly**
	|	**daily**
noun	→	**lecture**
	|	**chat**
	|	**learning**
	|	**labs**


The students of CPSC 231 enjoy the weekly labs .
extended_noun
second_case_attribute
verb
article
adjective
noun
noun_with_attribute
predicate
object
subject_part
predicate_part
sentence
sentence_sequence

Example (2): Description of basic C++ program structures

program	→	*includes* *declarations* *main*
includes	→	**#include <***lib***>**
lib	→	**iostream.h** | **math.h** | …
declarations	→	*single_declaration*; | *single_declaration*; *declarations*
single_declaration	→	*type identifiers*
type	→	**float** | **int** | …
identifiers	→	*single_identifier* | *single_identifier*, *identifiers*

main → **main() {**
declarations
instructions
return(0)
};

instructions → *single_instruction*; |
single_instruction; *instructions*

single_instruction → *assignment* |
cin_instruction |
cout_instruction |
conditional |
loop | …

assignment → *identifier* = *expression* |
identifier = …

(etc., this example grammar is not complete)

6.3.2 Semantics ("Meaning") Derived from Syntactic Structures

Example: analysing the arithmetic expression 12 * (9 + 8 / 2)

BNF for arithmetic expressions:

Expression:	E	$\rightarrow$	$T + E \mid T - E \mid T$
Term:	T	$\rightarrow$	$F * T \mid F / T \mid F$
Factor:	F	$\rightarrow$	$(E) \mid N$
Number:	N	$\rightarrow$	$DN \mid D$
Digit:	D	$\rightarrow$	$0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

A derivation for the arithmetic expression 12 * (9 + 8 / 2) looks as follows:

$$E \rightarrow T \rightarrow F * T \rightarrow N * T \rightarrow DN * T \rightarrow 1N * T \rightarrow 1D * T \rightarrow 12 * T \rightarrow$$

$$12 * F \rightarrow 12 * (E) \rightarrow 12 * (T + E) \rightarrow 12 * (F + E) \rightarrow 12 * (N + E) \rightarrow$$

$$12 * (D + E) \rightarrow 12 * (9 + E) \rightarrow 12 * (9 + T) \rightarrow 12 * (9 + F/T) \rightarrow$$

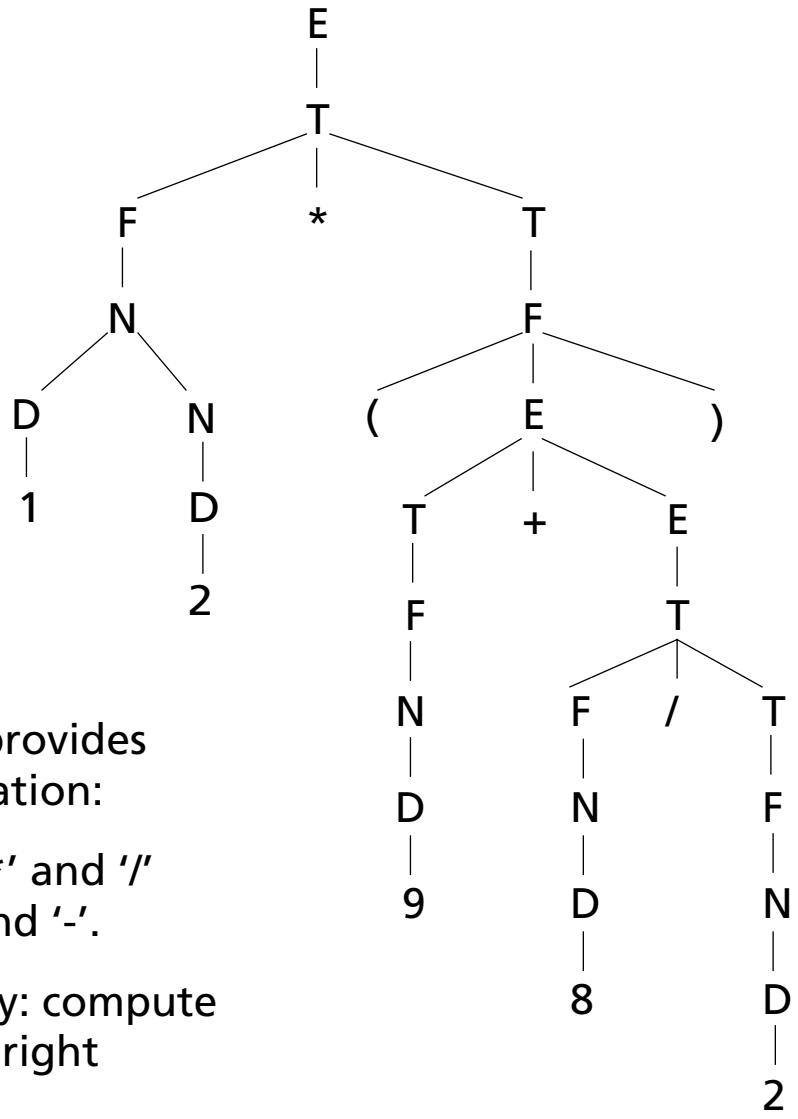
$$12 * (9 + N/T) \rightarrow 12 * (9 + D/T) \rightarrow 12 * (9 + 8/T) \rightarrow 12 * (9 + 8/F) \rightarrow$$

$$12 * (9 + 8/\mathrm{N}) \rightarrow 12 * (9 + 8/\mathrm{D}) \rightarrow 12 * (9 + 8/2)$$

This is only one of many possible derivations!

Here we used a **left** derivation, i.e., the left-most (non-terminal) symbol is always expanded first.

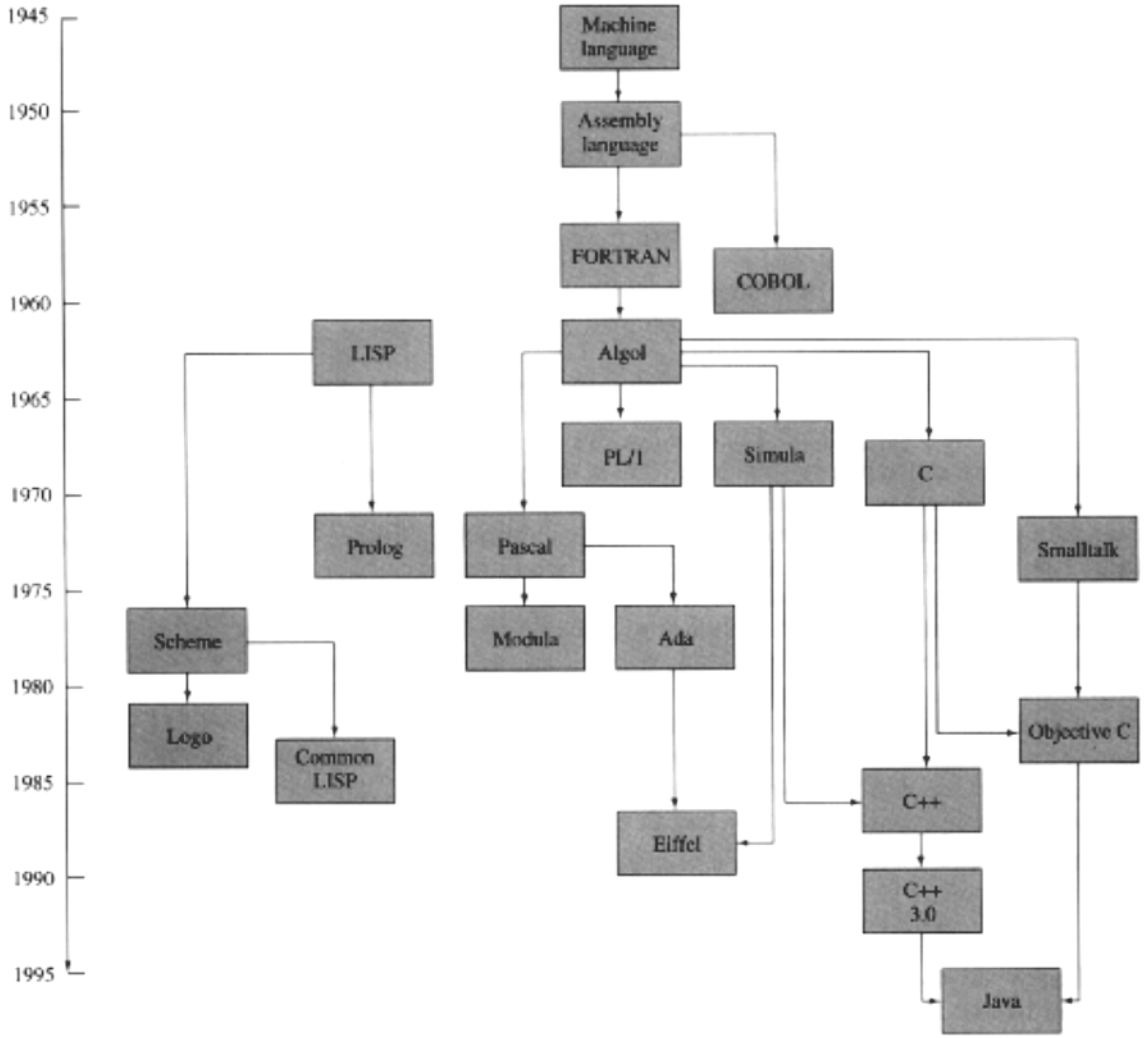
A Parse Tree for 12 * (9 + 8 / 2)



The syntax tree provides **semantic** information:

- priority of '*' and '/' before '+' and '-'.
- same priority: compute from left to right

6.4 An Overview of Programming Languages



6.5 References

- G. Blank and R. Barnes, *The Universal Machine*, Boston, MA: WCB/ McGraw-Hill, 1998. Chapters 4.1, 4.2, 4.4.2, and 4.5.