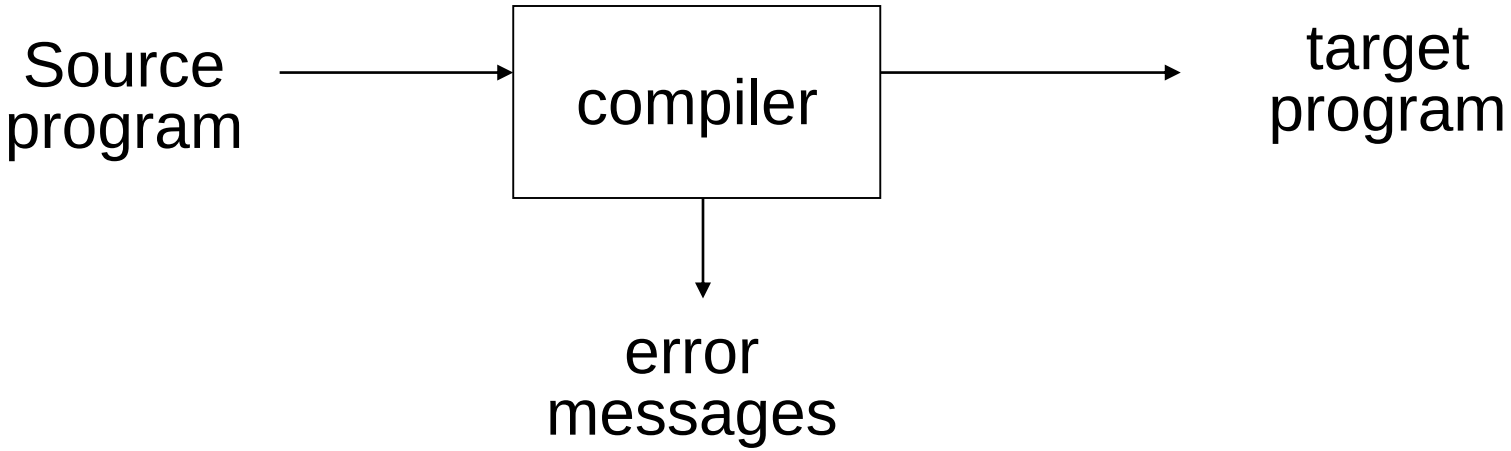
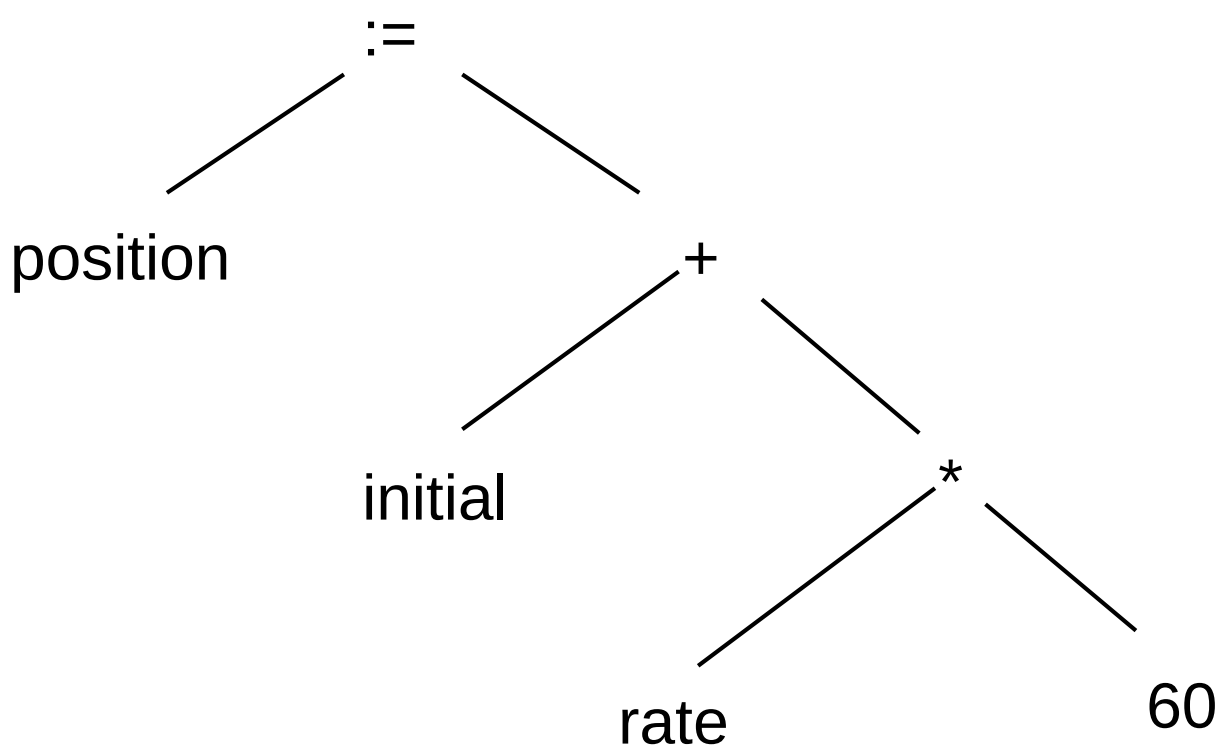

# Introduction

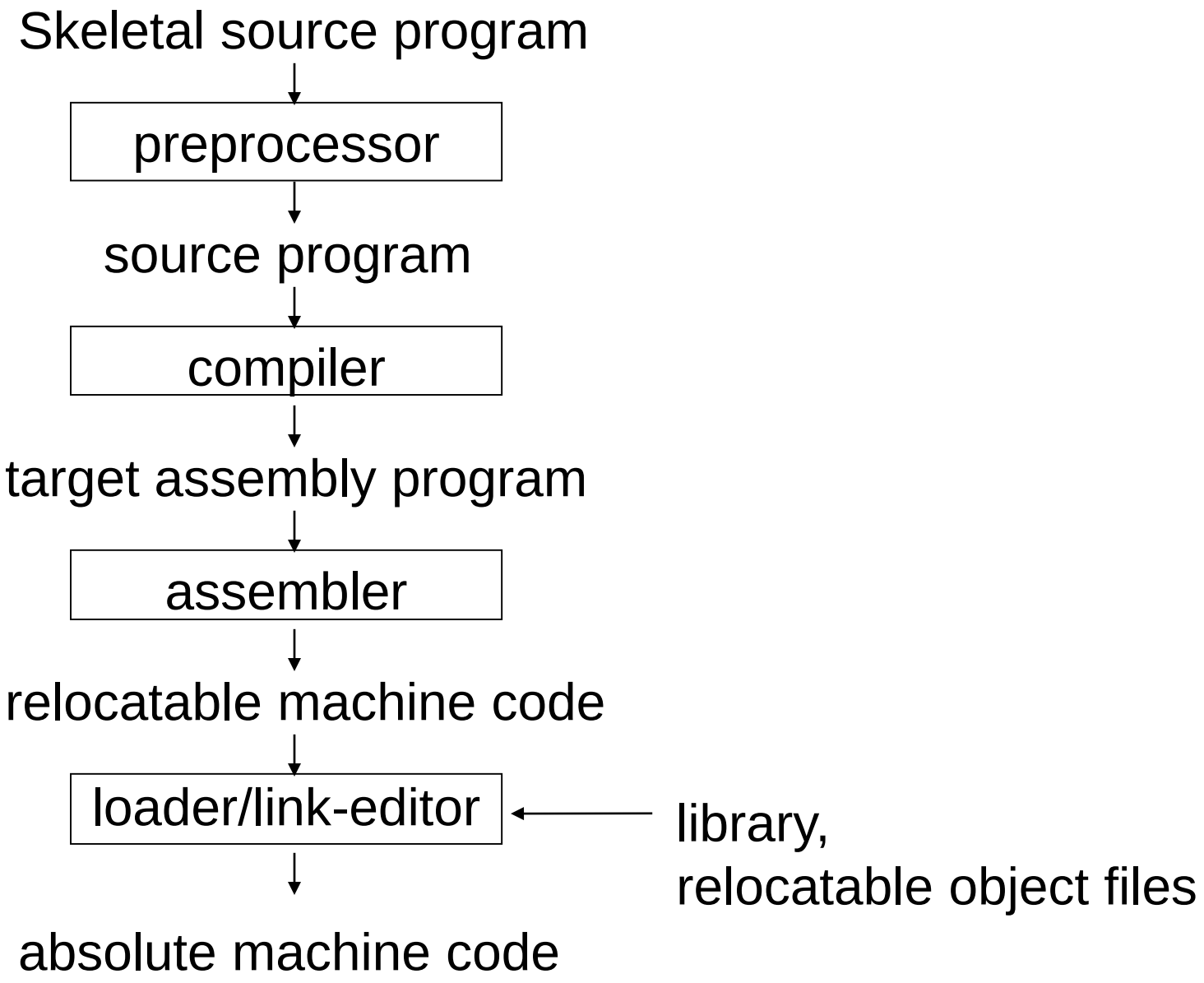



**A compiler**

- **The Analysis-Synthesis Model of Compilation**
  - There are two parts to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. Of the two parts, synthesis requires the most specialized techniques.




Syntax tree for position:=initial + rate* 60




A language-processing system

# Analysis of the source program

- In compiling, analysis consists of three phases:
  1. *Linear analysis*, in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

# Analysis of the source program

2. *Hierarchical analysis*, in which characters or tokens are grouped hierarchically into nested collections with collective meaning.
3. *Semantic analysis*, in which certain checks are performed to ensure that the components of a program fit together meaningfully.

# Lexical analysis

- In a compiler, linear analysis is called lexical analysis or scanning. For example, in lexical analysis the characters in the assignment statement

  Position := initial + rate * 60

  would be grouped into the following tokens:

1. The identifier position.
2. The assignment symbol :=.
3. The identifier initial.
4. The plus sign.
5. The identifier rate.
6. The multiplication sign.
7. The number 60.

- The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

# Syntax analysis

- Hierarchical analysis is called *parsing* or *syntax analysis*. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source program are represented by a parse tree such as the one shown in Fig.1.4.

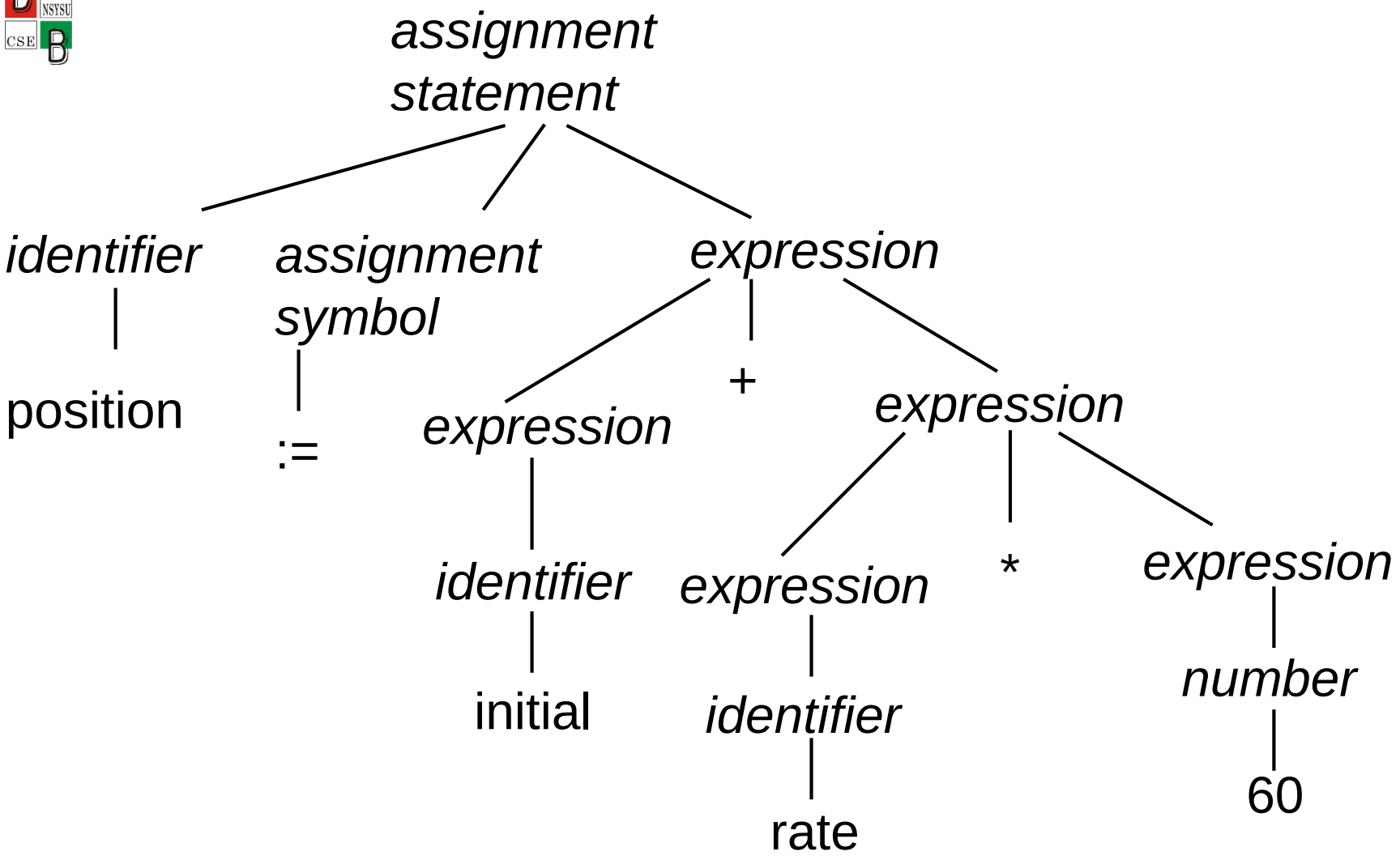



Fig.1.4. Parse tree for position:= initial + rate * 60

# Syntax analysis

- The hierarchical structure of a program is usually expressed by recursive rules. For example, we might have the following rules as parts of the definition of expressions:

# Syntax analysis

1. Any identifier is an expression.
2. Any number is an expression.
3. If expression1 and expression2 are expressions, then so are

   $expression_1$ + $expression_2$

   $expression_1$ * $expression_2$

   ( $expression_1$ )

# Syntax analysis

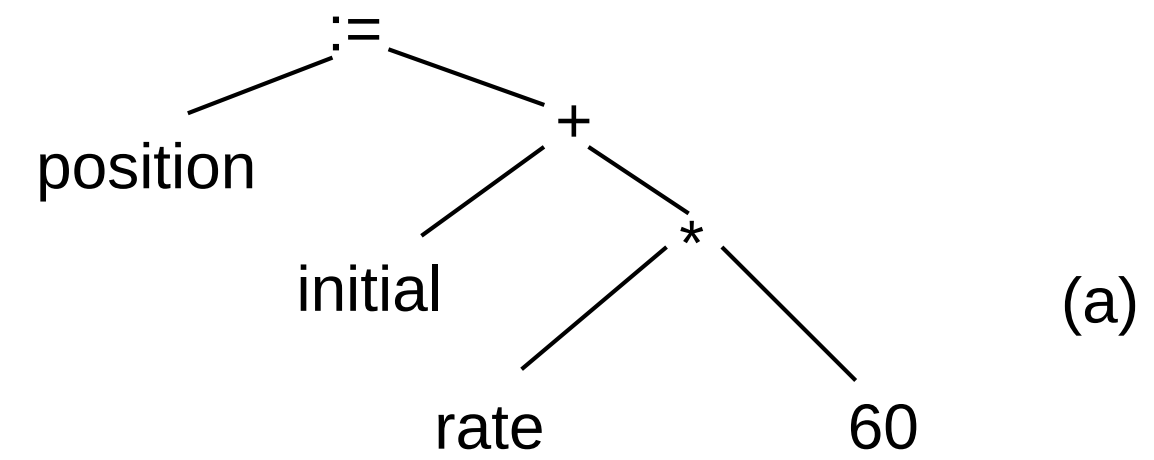
- Lexical constructs do not require recursion, while syntactic constructs often do. Context-free grammars are a formalization of recursive that can be used to guide syntactic analysis.

# Semantic Analysis

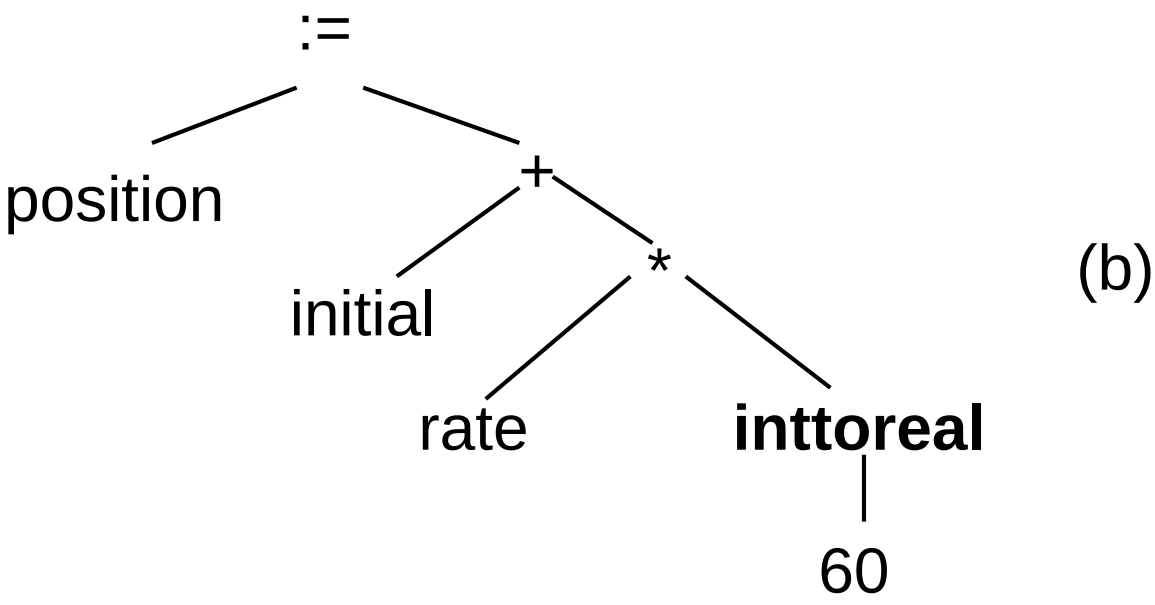
- The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.

# Semantic Analysis

- An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification. For example, many programming language definitions require a compiler to report an error every time a real number is used to index an array. However, the language specification may permit some operand coercions, for example, when a binary arithmetic operator is applied to an integer and real. In this case, the compiler may need to convert the integer to a real.




(a)




(b)

Semantic analysis inserts a conversion form integer to real

# Symbol-Table Management

- An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. These attributes may provide information about the storage allocated for an identifier, its type, its scope (where in the program it is valid), and, in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (e.g., by reference), and the type returned, if any.

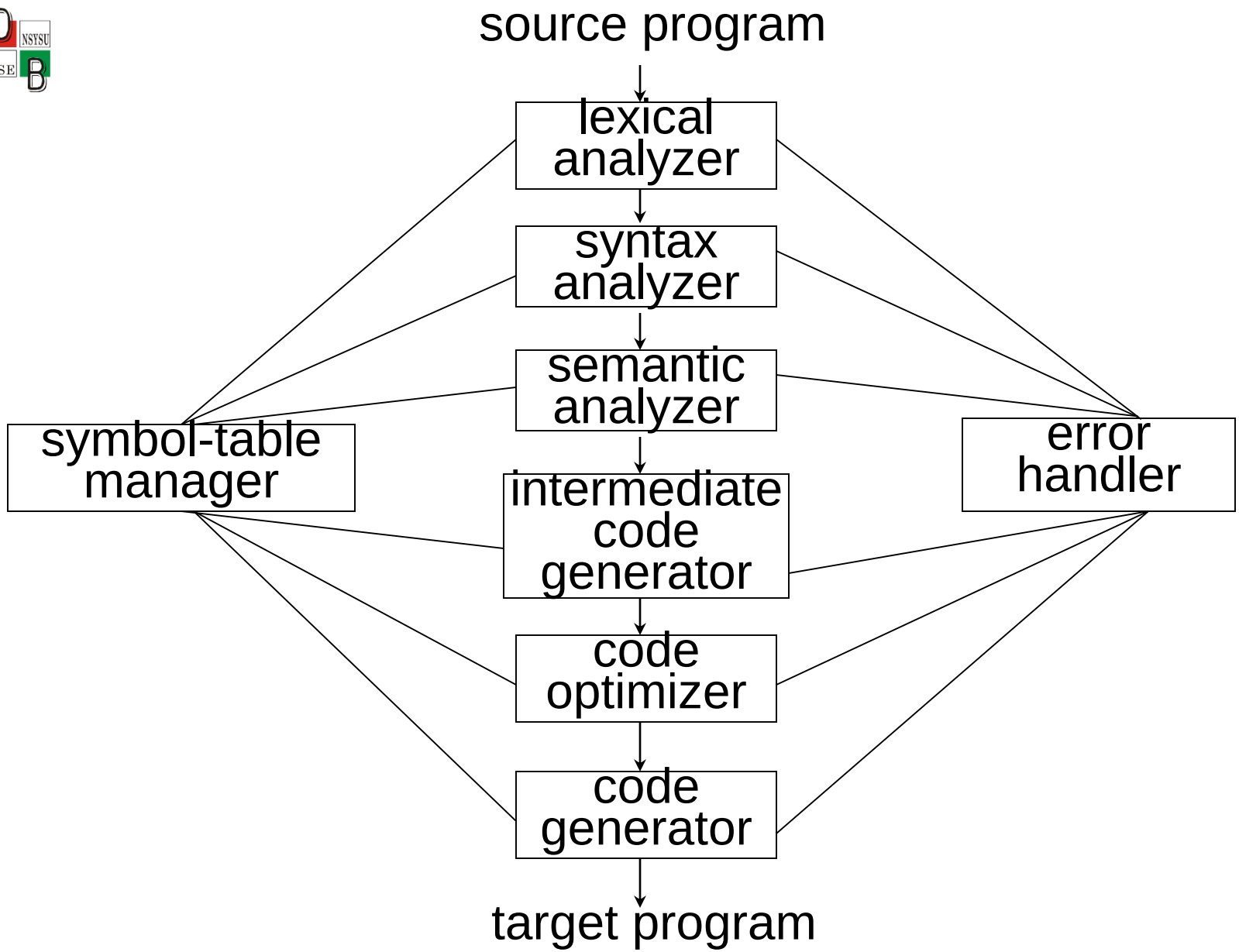



**Fig.1.9.** phases of a compiler.

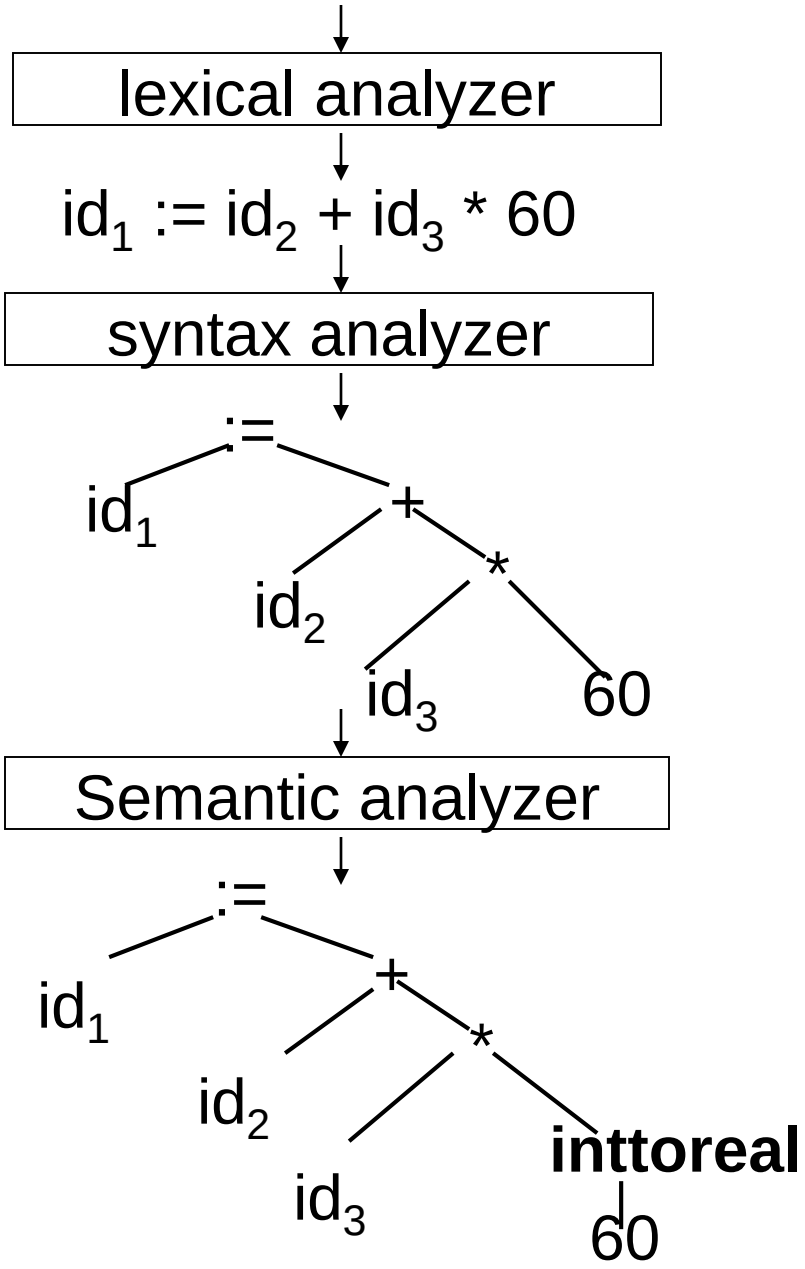
# Error Detection and Reporting

- Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected. A compiler that stops when it finds the first error is not as helpful as it could be.

# Intermediate Code Generation

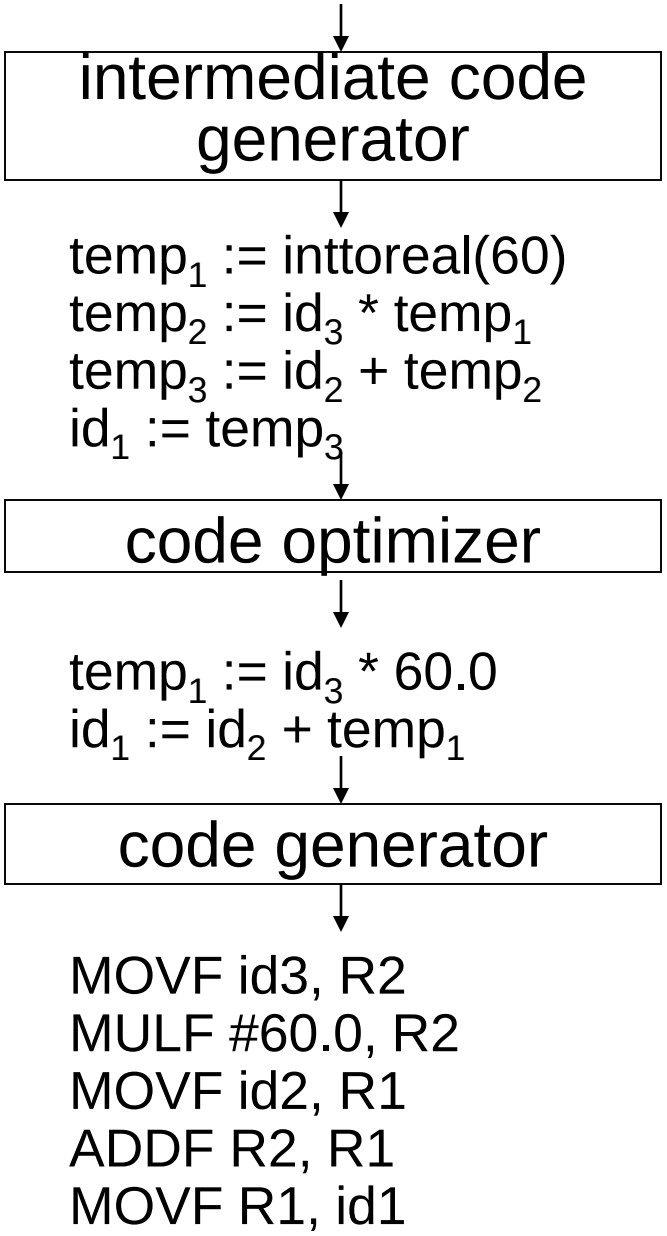
- After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. We can think of this intermediate representation as a program for an abstract machine. This intermediate representation should have two important properties; it should be easy to produce, and easy to translate into the target program.

position := initial + initial * 60




SYMBOL TABLE

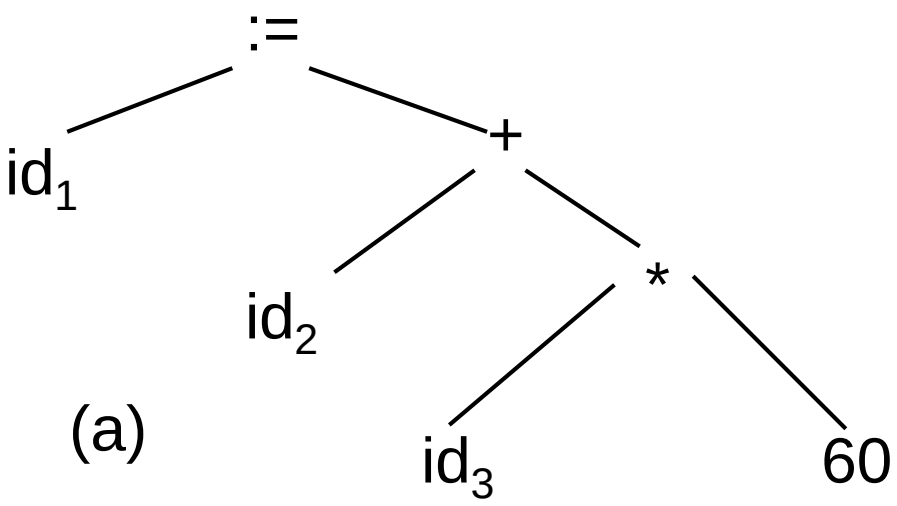
| | | |
|---|---|---|
| 1 | position | … |
| 2 | initial | … |
| 3 | rate | … |
| 4 | | … |



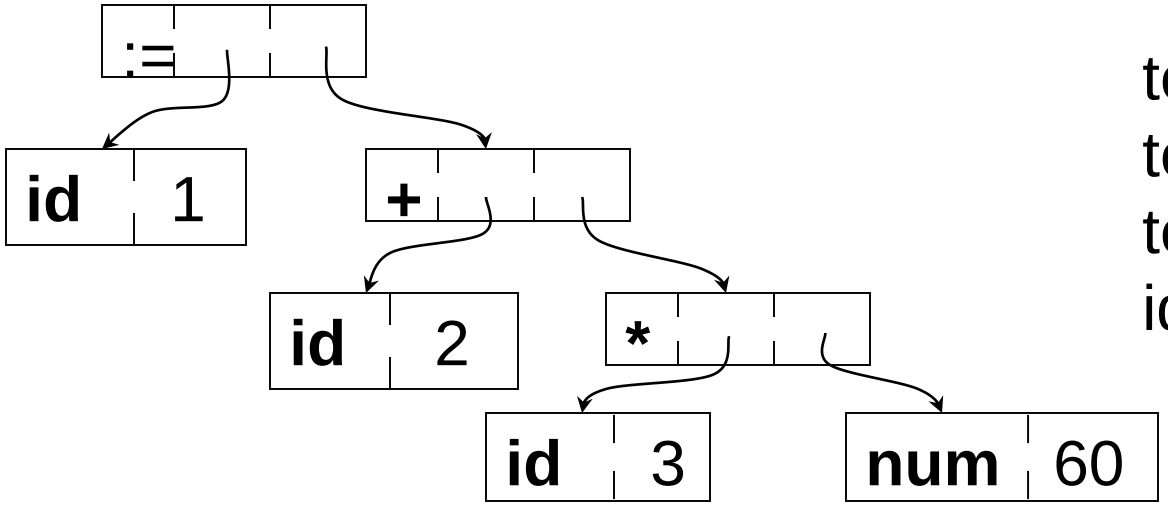

SYMBOL TABLE

| | | |
|---|---|---|
| 1 | position | … |
| 2 | initial | … |
| 3 | rate | … |
| 4 | | … |

Translation of a statement




(a)




(b)

$temp_1 := inttoreal(60)$
$temp_2 := id_3 * temp_1$
$temp_3 := id_2 + temp_2$
$id_1 := temp_3$

The data structure in (b) is for the tree in (a)

- This intermediate form has several properties. First, each three-address instruction has at most one operator in addition to the assignment. Thus, when generating these instructions, the compiler has to decide on the order in which operations are to be done; the multiplication precedes the addition in the source program of (1.1). Second, the compiler must generate a temporary name to hold the value computed by each instruction. Third, some “three-address” instructions have fewer than three operands, e.g., the first and last instructions in (1.3).

# Code optimization

- The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result. Some optimizations are trivial. For example, a natural algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation after semantic analysis, even though there is a better way to perform the same calculation, using the two instructions

```
temp1 := id3 * 60.0
id1 := id2 + temp1          (1.4)
```

# Code Generation

- the final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.

# Code Generation

- For example, using register 1 and register 2, the translation of the code of (1.4) might become

```
MOVF   id3,  R2
MULF   #60.0,  R2
MOVF   id2,  R1
ADDF   R2,  R1                    (1.5)
MOVF   R1,  id1
```