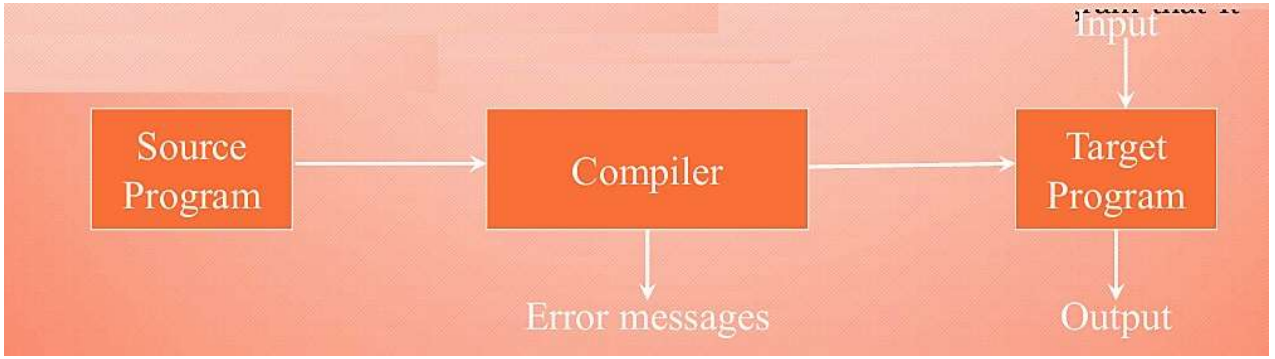
Compiler Design

Detailed Compiler Phases with Example

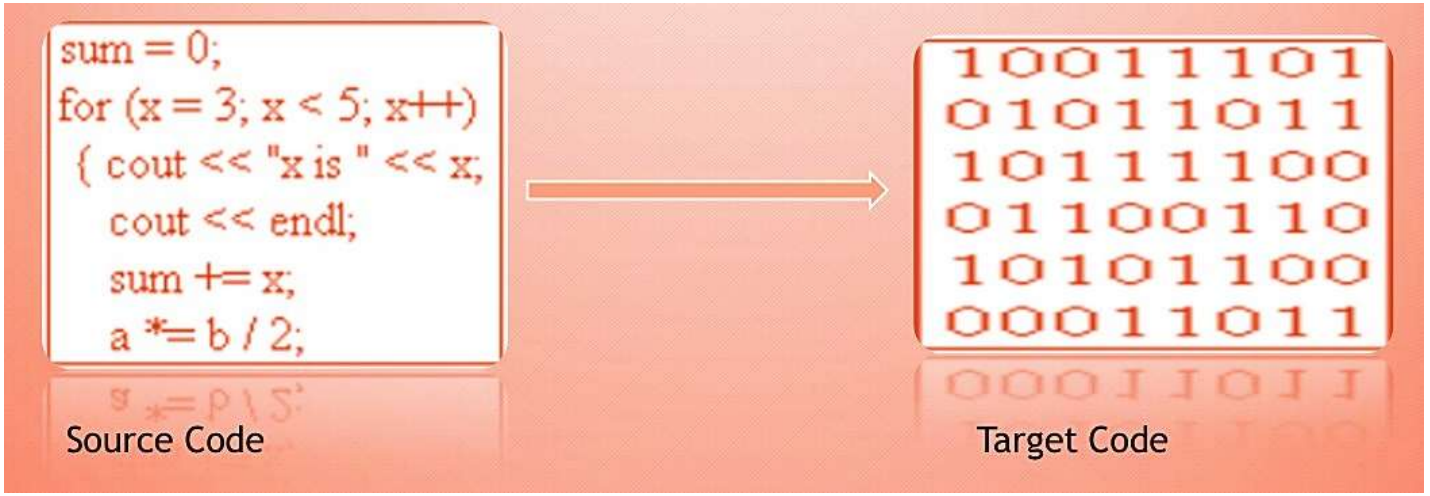
Compiler (The Language Processors)

- Compiler a system software → read a program in one language (source language) and translate it into an equivalent program in another language (target language).
- An important role of the compiler is to ***report any errors*** in the source program that it detects during the translation process.



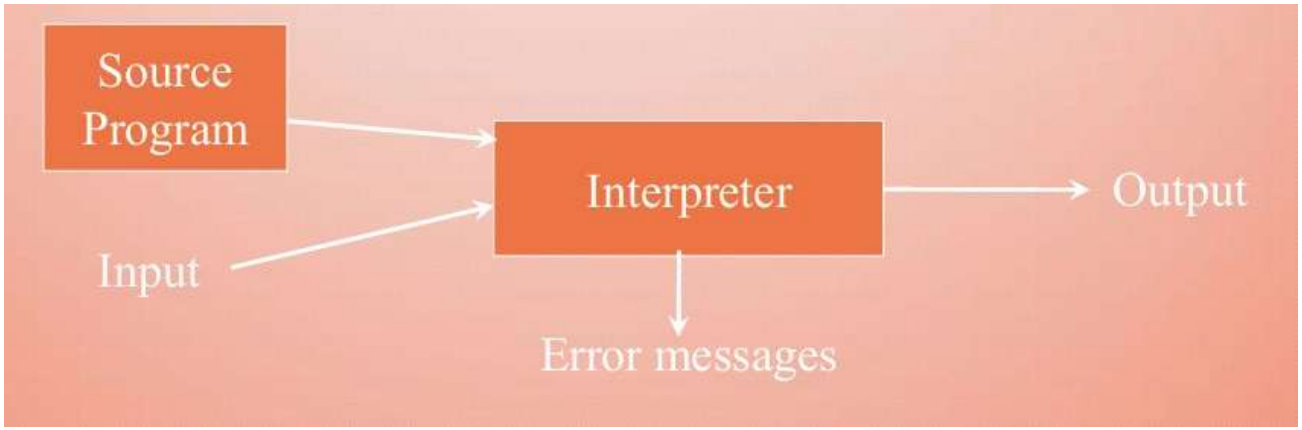
- If the target program is an *executable machine-language program*, it can then be called by the user to process inputs and produce outputs.

Example


sum = 0;
for (x = 3; x < 5; x++)
{ cout << "x is " << x;
cout << endl;
sum += x;
a *= b / 2;
Source Code
10011101
01011011
10111100
01100110
10101100
00011011
Target Code

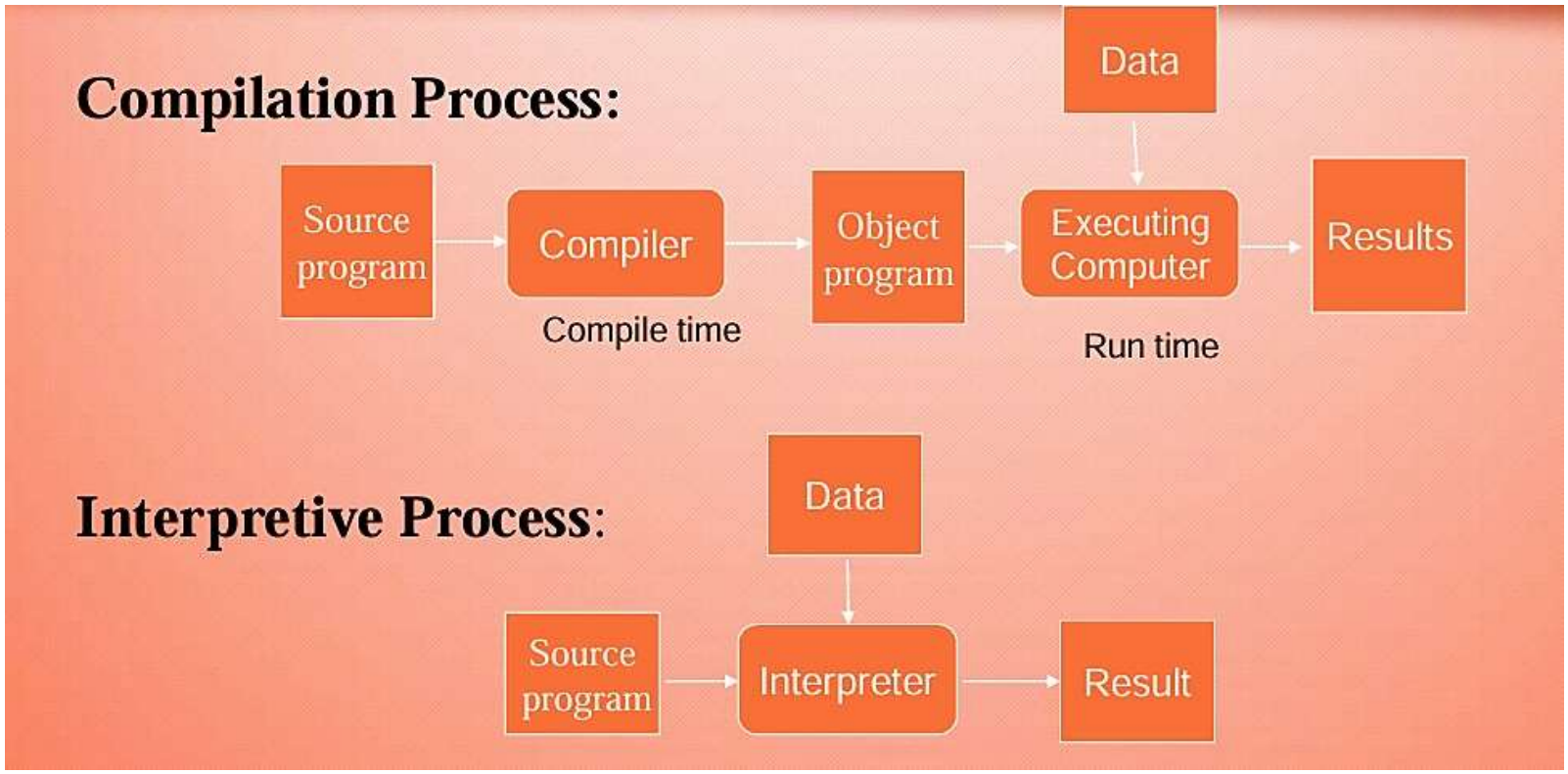
Contd.,

- An interpreter is another kind of language processor.
- Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program or inputs supplied by the user.



- The machine-language target program produced by a compiler much faster than an interpreter at mapping inputs to outputs .
- An interpreter, give better error diagnostics than a compiler, because it executes the source program statement by statement

Working Process of Compilers Vs Interpreter

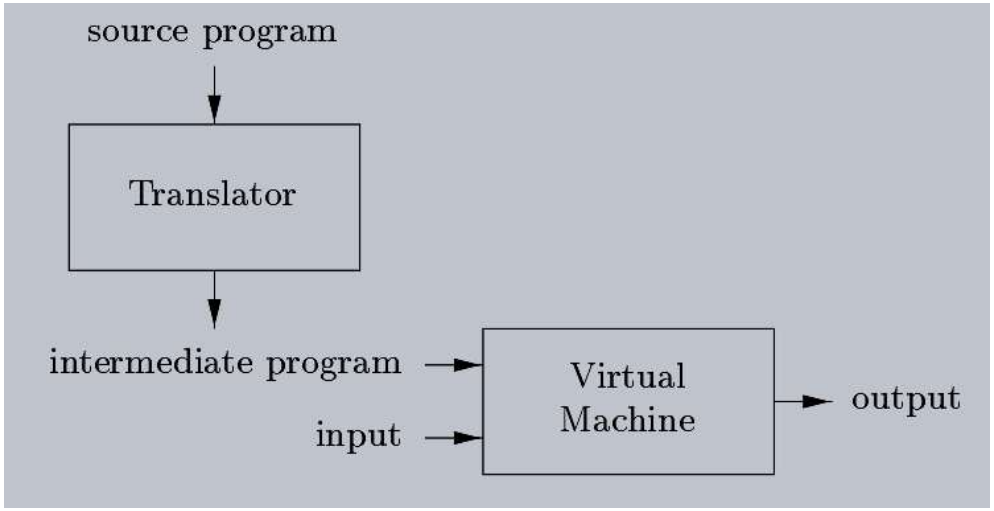


Contd.,

- Compiler
  - Compiler Takes **Entire program** as input.
  - Intermediate Object Code is **Generated**.
  - Conditional Control Statements are Executes **faster.**
  - **Memory Requirement : More** (Since Object Code is Generated).
  - Program need not be **compiled** every time.
  - **Errors** are displayed after **entire program** is checked.
  - Programming language like C, C++ use compilers.

- Interpreter
  - Interpreter Takes **Single** instruction as input.
  - **No** Intermediate Object Code is **Generated.**
  - Conditional Control Statements are Executes **slower**.
  - **Memory Requirement is Less**.
  - **Every time** higher level program is converted into lower level program.
  - **Errors are displayed for every instruction** interpreted (if any).
  - Programming language like Python, Ruby use interpreters.

Contd., Example



A hybrid compiler

- A Java source program 1st compiled into an intermediate form called bytecodes, then bytecodes interpreted by a virtual machine.
- *Benefit:* bytecodes compiled on one machine can be interpreted on another machine (across a network).
- *For faster processing*, just-in-time **(JIT)** compilers, translate bytecodes into machine language.
- Large programs compiled in pieces → with linkers and loaders.
  - linker → resolves external memory addresses, where the code in one file may refer to a location in another fille.
  - loader → then puts together all of the executable object files into memory for execution.

The Structure of Compilers

- Compiler a single box → Open the Box → → two parts to mapping: **analysis and synthesis.**

Analysis Part:

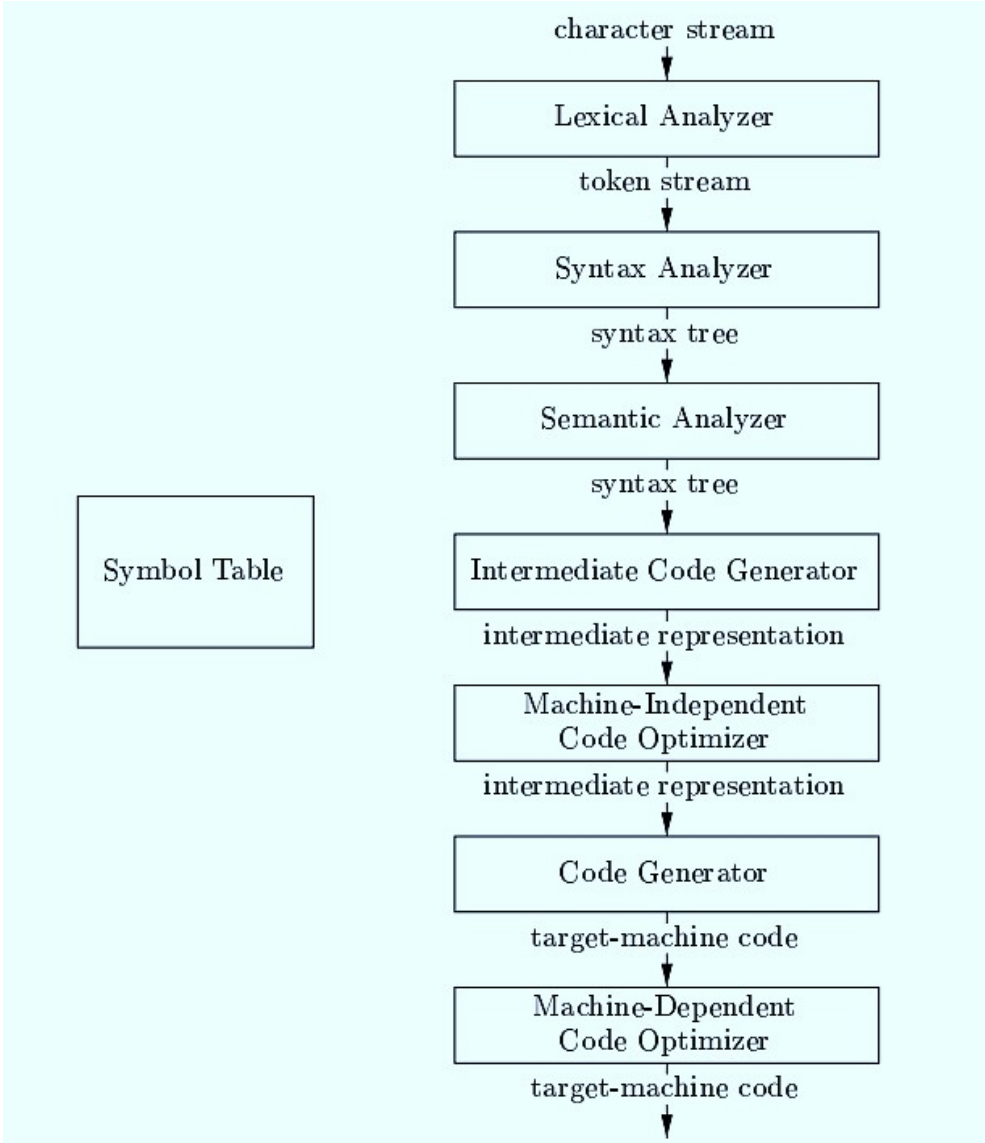
- Breaks-up source program into integral pieces and imposes a grammatical structure on them.
- It then uses this structure to create an intermediate representation of the source program.
- If it detects any either syntactically ill formed or semantically unsound, *then it must provide informative messages, so user can take corrective action.*
- It also **collects information about source program and stores it in a data structure** called symbol table, which passed along with intermediate representation to synthesis part.

Synthesis part

- It constructs desired target program from intermediate representation and the information in the symbol table.
- Synthesis back-end of the compiler; and the Analysis part is the front end.

Contd.,

If we examine more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another.



Lexical Analysis

- The **1st phase** of a compiler called lexical analysis or scanning.
- It reads stream of characters making up 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

<Token Name, Attribute-value>

- **Token-name** → abstract symbol used during syntax analysis,
- **Attribute-value** → points to an entry in symbol table for this token.

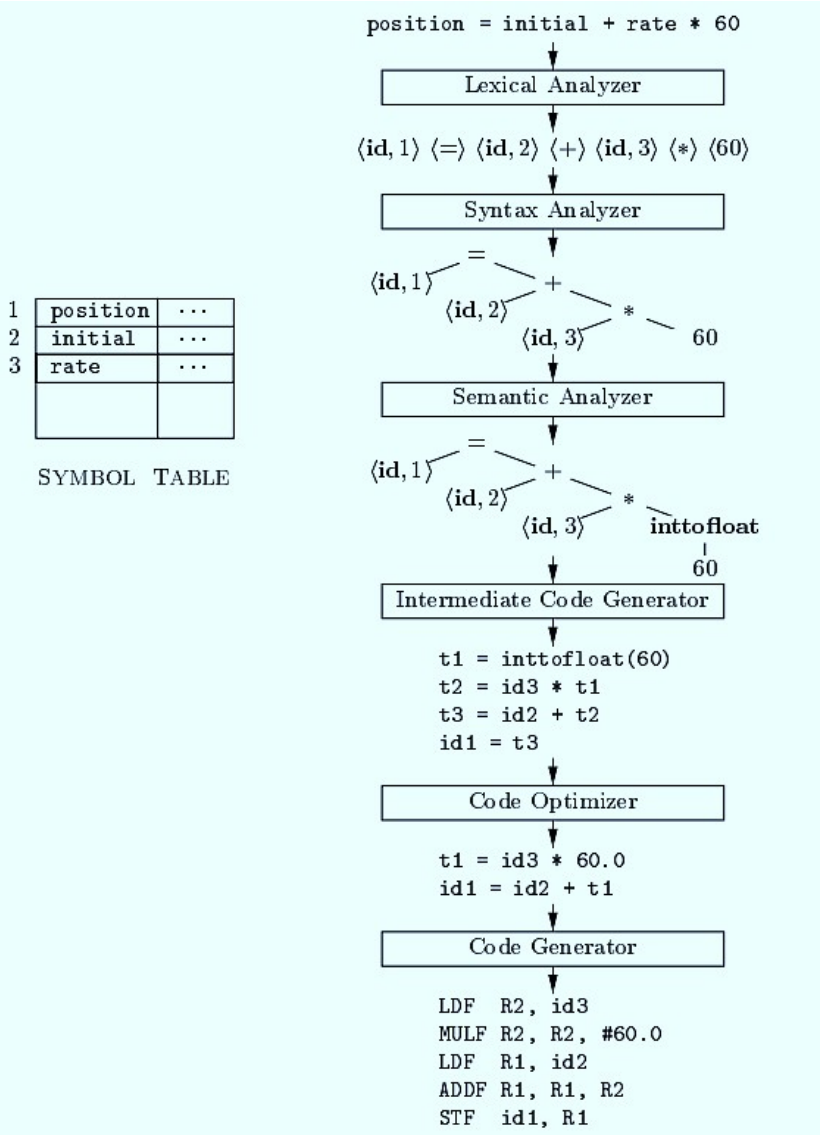
Information from the symbol-table entry is needed for semantic analysis and code generation

Example: suppose a source program contains the assignment statement

position = initial + rate * 60

- Following tokens passed on to the syntax analyzer
- position is lexeme that mapped into a token < id; 1 >, where id = abstract symbol standing for identier and 1 points to symbol table entry for position.
- The assignment symbol = is a lexeme that is mapped into the token < = >.
- **initial** is a lexeme that is mapped into the token <id; 2>, where 2 points to the symbol-table entry for initial.
- + is a lexeme that is mapped into the token <+>.
- rate is a lexeme that mapped into the token <id; 3>, where 3 points to the symbol-table entry for rate.
- * is a lexeme that is mapped into the token <*>.
- 60 is a lexeme that is mapped into the token <60>

After lexical analysis as the sequence of tokens **<id; 1> <=> <id; 2> <+> <id; 3> <*> <60>**



Translation of an assignment statement

Lexical Analysis … Contd.,

- In lexical analysis the characters in the assignment statement.

  position := initial + rate * 60

- would be grouped into the following tokens:

Type of Token	Value of Token
identifier	**position**
the assignment symbol	**:=**
identifier	**Initial**
the plus sign	**+**
identifier	**rate**
The Multiplication Sign	*****
numbers	**60**

Note: The types in the first column are usually represented by codes

Lexical Analysis … Contd.,

- The implementation of a lexical analyzer is based on finite state automata
  - which models how characters are read **one by one**, transitioning between states to recognize valid patterns (like identifiers or numbers).

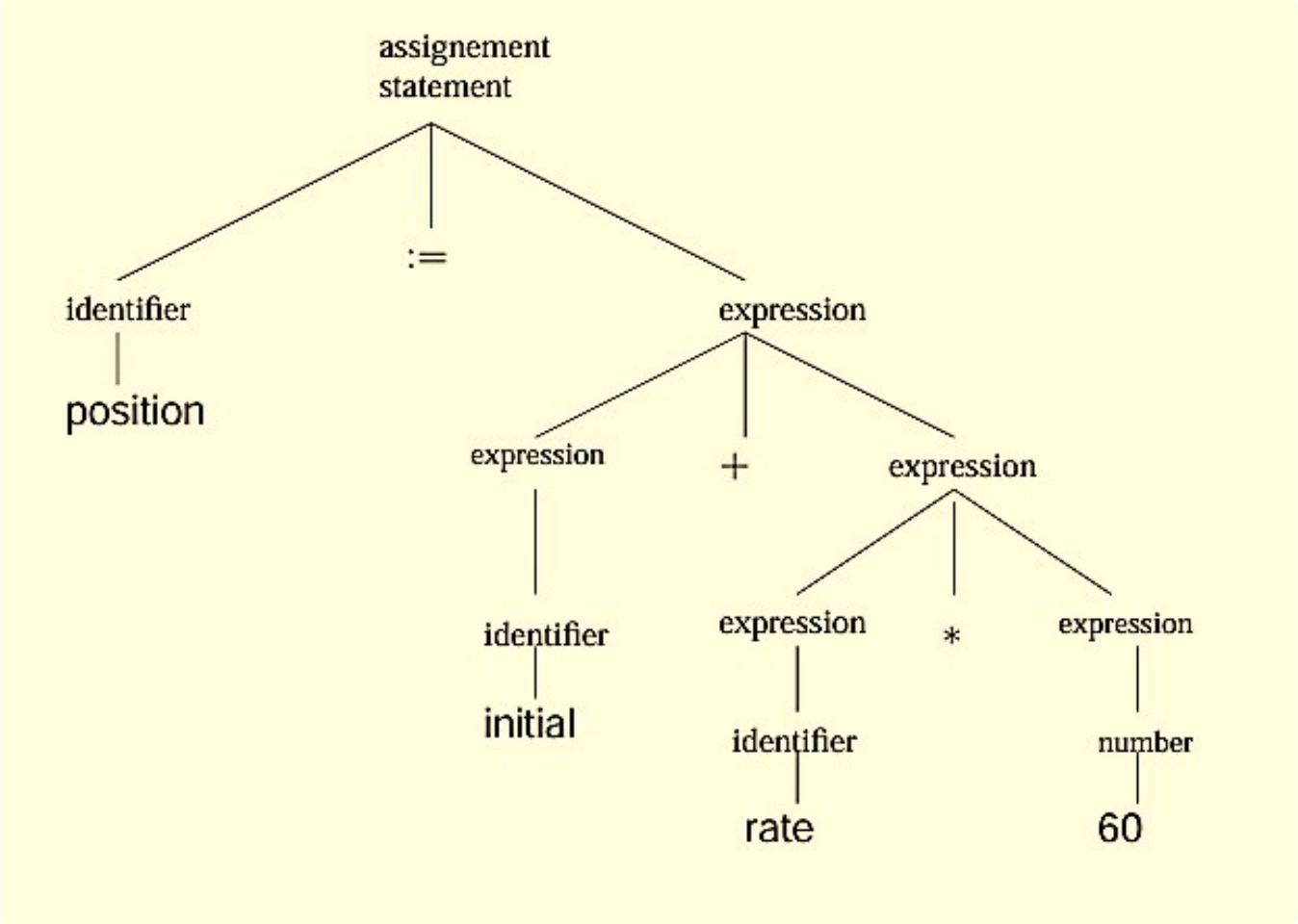
- Example:

letter (letter | digit | underline)*

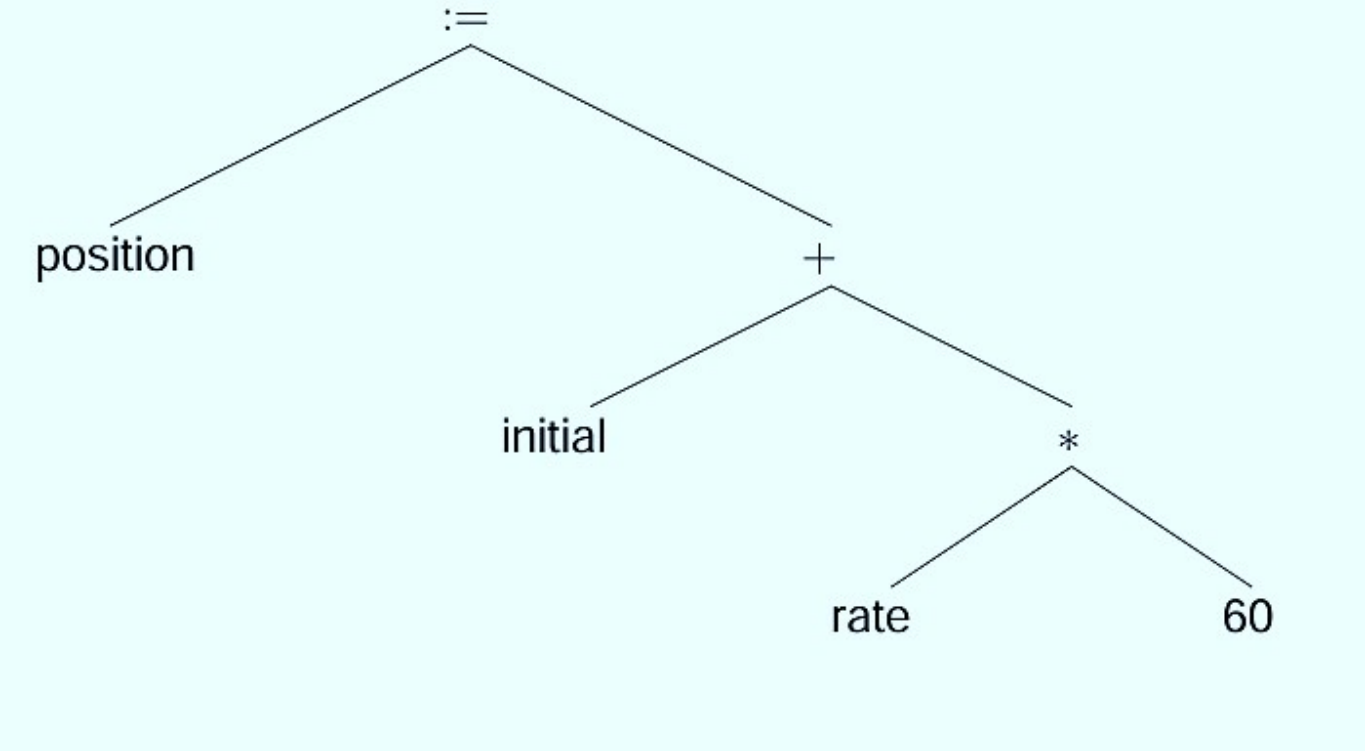
Syntax Analysis: Parser

- The parser receives the tokens from the lexical analyzer and checks if they arrive in the correct order
- It involves grouping the tokens into grammatical phrases that are used by the compiler to produce output.
- In general, the syntax of a programming language is described by a context free grammar.
- The job of the parser is to recover the hierarchical structure of the program from the stream of tokens received from the lexical analyzer.
- The implementation of the compiler depends on output of the parser : **usually a tree**
- **Parse tree**: describes the syntactic structure of the input
- **Syntax tree**: a compressed representation of the parse tree in which the operators appear as the interior node and the operands of an operator are the children of the node

Parse tree


assignement
statement
identifier
:=
expression
position
expression
+
expression
identifier
expression
*
expression
initial
identifier
number
rate
60

Syntax tree



Semantic Analysis

- Checks the source program for semantic errors and gathers type information for the subsequent code generation phrase
- Uses the syntax analysis phase to identify the operators and operands of the expressions and statements
- In a context free grammar, it is not possible to represent a rule such as: All identifier should be declared before being used. The verification of this rule worries the semantic analysis.
- Important component of semantic analysis is **type checking**:

  The compiler checks that each operator has operands that are permitted by the source language specification

  Example: arithmetic operator applied to an integer and a real

- First phase that deals with the meanings of programming language concepts

Intermediate Code Generator

- **Helps to relocate** the code from one process to another
- Simple language supported by most of the modern processors
- Powerful enough to express programming language constructs
- Two important properties of the intermediate representation:
  - easy to produce
  - easy to translate into the target program

- Example: three-address code

```
temp1  :=  inttoreal(60)
temp2  :=  id3 * temp1
temp3  :=  id2 + temp2
id1    :=  temp3
```

Code Optimization

- Attempts to improve the intermediate code, so that faster-running machine code will result
- Example: The intermediate code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

can be performed by using two instructions:

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

- In optimizing compilers, a significant fraction of the time of the compiler is spent on this phase.
- However, there are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much

Code Generation

- Final phase of the compiler
- Generation of target code (in general, machine code or assembly code)
- Intermediate instructions are translated into a sequence of machine instructions

Translation of the intermediate code

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

into some machine code:

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

- **where:** the first and the second operands of each instruction specify a source and a destination, respectively
- The **F** in each instruction tells us that the instruction deals with floating-points numbers
- The **#**indicates that 60.0 is to be treated as a constant
- Registers 1 and 2 are used

Symbol Table Management

- A compiler needs to *record information concerning the objects found in the source program.*
- A **Symbol Table** is a data structure containing a record for each identifier, with fields for the attributes of the identifiers

  Note: Attributes provide information about identifiers: storage allocation, type, scope, names, etc
- Symbol Table allows the compiler to find a record for each identifier quickly and to store or retrieve data from that record quickly
- When an identifier is detected by the lexical analyzer, it is entered in the symbol table.
- Generally created by lexical analyzer and syntax analyzer
- Good data structures needed to minimize searching time
- The data structure may be flat or hierarchical

Error handling and Recovery

- 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 difficult task due to the following 2 reasons:
  - An error can hide another one
  - An error can provoke a lot of other errors (all of them solved by correcting the first one)
- Two criteria can be followed:
  - Stop when the first error is found (usually not very helpful, but can be used by an interactive compiler)
  - Find all the errors
- An important **criteria for judging the quality of compiler**
  - For a semantic error, compiler can proceed
  - For syntax error, parser enters into an erroneous state
- Recovery is essential to provide a bunch of errors to the user, so that all of them may be corrected together instead of one-by-one.

Phases and passes

- In logical terms a compiler is thought of as consisting of stages and phases
- Physically it is made up of passes
- The compiler has one pass for each time the source code, or a representation of it, is read
- *Many compilers have just a single pass so that the complete compilation process is performed while the code is read once*
- The various phases described will therefore be executed in parallel
- Earlier compilers had a large number of passes, typically due to the limited memory space available
- *Modern compilers are single pass since memory space is not usually a problem*

Use of tools

- The 2 main types of tools used in compiler production are:

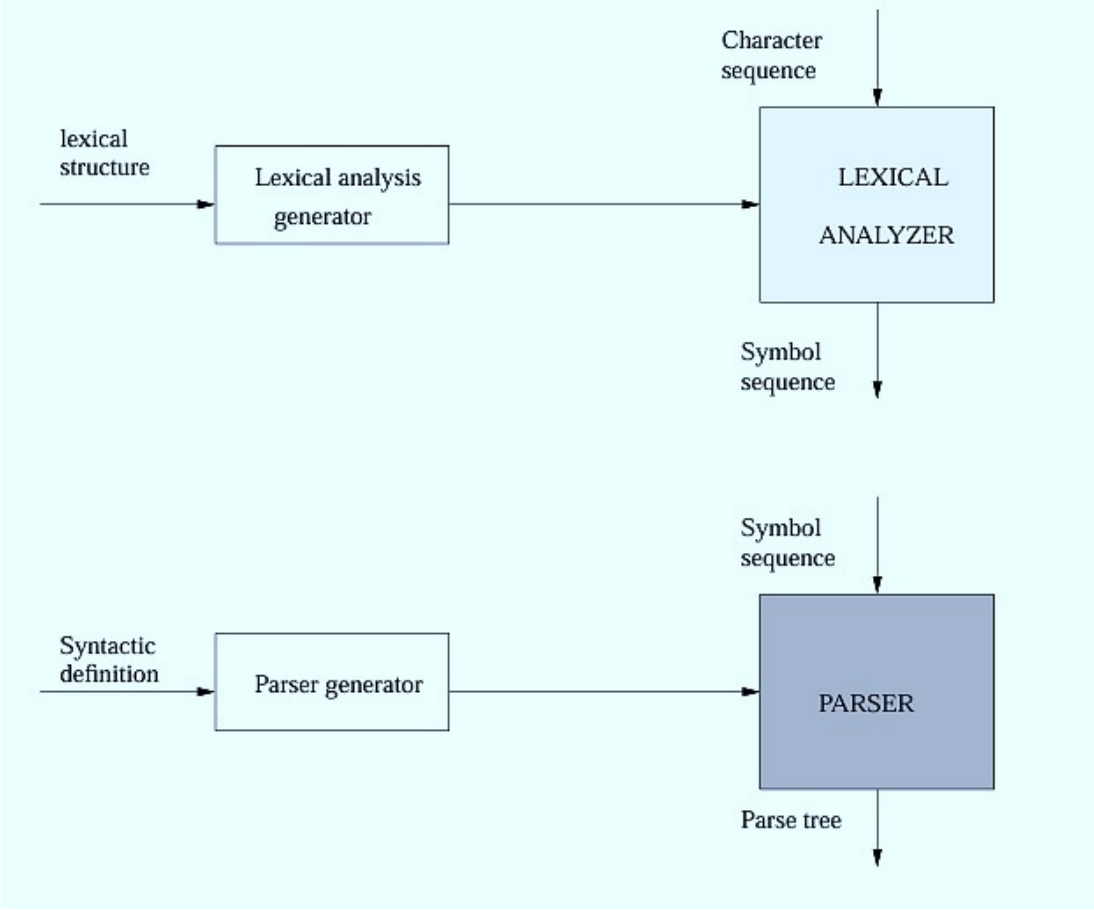
1. a lexical analyzer generator

- Takes as input the lexical structure of a language, which defines how its tokens are made up from characters
- Produces as output a lexical analyzer (a program in C for example) for the language
- Unix lexical analyzer Lex

2. a symbol analyzer generator

- Takes as input the syntactical definition of a language
- Produces as output a syntax analyzer (a program in C for example) for the language
- The most widely know is the Unix-based YACC (Yet Another Compiler-Compiler), used in conjunction with Lex. Bison: public domain version

Lexical analyzer generator and parser generator



Code Linking and Loading (Post Compilation)

- **Linker** adds in code from **libraries** (like standard I/O) and combines all object files.

- **Loader** loads the compiled program into **main memory** and starts execution.

Summary Table

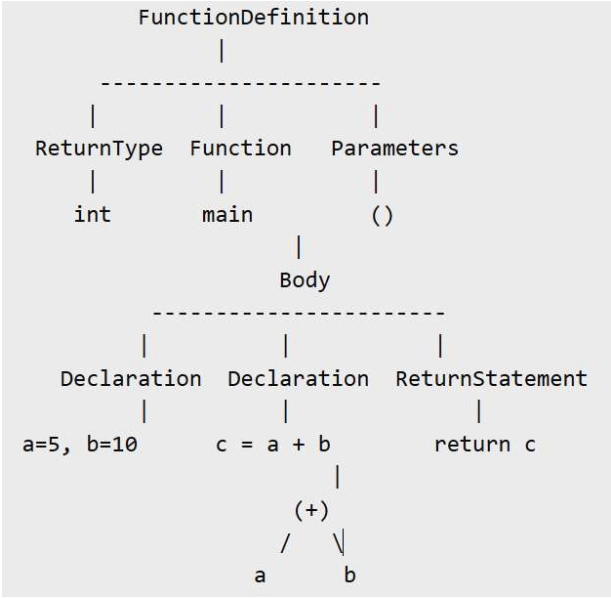
Phase	Input	Output	Purpose
Lexical Analysis	Source code	Tokens	Tokenization
Syntax Analysis	Tokens	Parse Tree	Structure check
Semantic Analysis	Parse Tree	Annotated Tree	Type, scope validation
IR Generation	Annotated Tree	Intermediate Code	Language-independent representation
Code Optimization	Intermediate Code	Optimized Code	Improve performance
Code Generation	Optimized Code	Assembly/Machine Code	Target CPU code
Linking & Loading	Object Code + Libraries	Executable	Final run-ready program

Example Program (in C):

```
int main() {
    int a = 5, b = 10;
    int c = a + b;
    return c;
}
```

[INT] [IDENTIFIER: main] [(] [)] [{]
[INT] [IDENTIFIER: a] [=] [CONSTANT: 5] [,] [IDENTIFIER: b] [=]
[CONSTANT: 10] [;]
[INT] [IDENTIFIER: c] [=] [IDENTIFIER: a] [+] [IDENTIFIER: b] [;]
[RETURN] [IDENTIFIER: c] [;]
[}]

Lexical Analysis



Syntax Tree

```
int a = 5;      // a: int
int b = 10;     // b: int
int c = a + b;  // c: int
return c;       // c is int, matches function return type
```

Semantic Analysis

```
t1 = 5
t2 = 10
t3 = t1 + t2
return t3
```

Intermediate Code Representation

```
t3 = 15
return t3
```

Code Optimization

```
MOV R1, #5
MOV R2, #10
ADD R3, R1, R2
MOV R0, R3
RET
```

Target Code Generation