COMPUTER SCIENCE & INFORMATION TECHNOLOGY

Compiler Design

Comprehensive Theory
with Solved Examples *and* Practice Questions

MADE EASY PUBLICATIONS

www.madeeasypublications.org

MADE EASY Publications Pvt. Ltd.

Corporate Office: 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016 | **Ph. :** 9021300500

Email : infomep@madeeasy.in | **Web :** www.madeeasypublications.org

Compiler Design

EDITIONS

First Edition : 2015
Second Edition : 2016
Third Edition : 2017
Fourth Edition : 2018
Fifth Edition : 2019
Sixth Edition : 2020
Seventh Edition : 2021
Eighth Edition : 2022
Ninth Edition : 2023
Tenth Edition : 2024
Eleventh Edition : 2025

Twelfth Edition : 2026

CONTENTS

Compiler Design

Compiler Design

GOAL OF THE SUBJECT

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space. In order to understand or construct the compiler one must be aware of its design principles. Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

INTRODUCTION

In this book we tried to keep the syllabus of Compiler Design around the GATE syllabus. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into six chapters as described below.

1. Introduction to Compiler: In this chapter we will introduce you to the Analysis-Synthesis model of compilation, various stages of Compilation (namely: lexical, syntax, semantic, intermediate code generation, code optimization and code optimization), grouping of various stages into analysis phase and synthesis phase, passes in compiler and finally we discuss the bootstrapping.
2. Lexical Analysis: In this chapter we will study Tokens, lexemes and their patterns and finally we discuss various ways of specifying tokens.
3. Syntax Analysis (Parser): In this chapter we introduce you the types of parsers, Top down parser: LL (1) parsing, Bottom up parsers: LR parser, SLR parser, CLR parser, LALR parser and Operator precedence parsing.
4. Syntax Directed Translation: In this chapter we will study about the Syntax directed definition, attributes (synthesized and inherited), Construction of Syntax trees, of S-Attributed Definitions, L-Attributed Definitions, Bottom up evaluation of inherited Attributes, dependency graph using SDT, SDT for intermediate code generation.
5. Intermediate Code Generation: In this chapter, we will study about the code generation for program and their various representations. We will also discuss basic blocks and flow graphs.
6. Run Time Environment: In this chapter we will study about the activation trees, control stacks, Storage organization, storage allocation strategies, scope of variables, Symbol table representations, parameter passing.

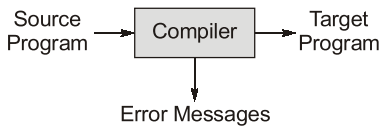
■■■■

CHAPTER

1 Introduction to Compiler

1.1 COMPILER

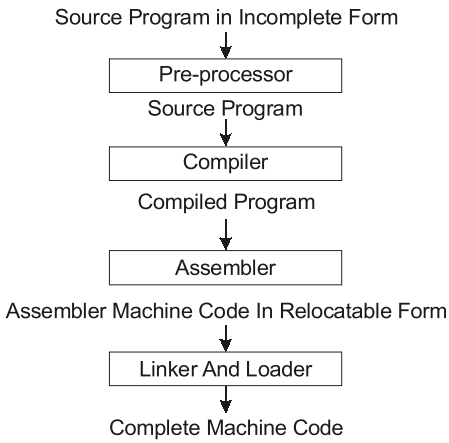
A compiler is a program that reads a program written in one language the source language-and translates into an equivalent program in another language-the target language. The translation process should also report the presence of errors in the source program.



1.1.1 Interpreter

An interpreter is a program that executes other programs. The interpreter and the underlying machine are combined into a virtual machine and simulate the execution of the input program on the virtual machine.

In general, the source language is any high level language and the object language is an assembly/machine language. We can show the general process of compilation by using following diagram.



Pre-processor is a program that processes its input data to produce output that is used as input to another program (compiler). The output is said to be a preprocessed form of the input data, which is generally used by compiler. Very basic work of pre-processor is to insert the files which are named in source program.

For *example*, let in source program, it is written #include "stdio.h". Pre-processor replace this file by its contents in the produced output.

The basic work of linker is to merge object codes (that has not even connected), produced by compiler, assembler, standard library functions, and operating system resources.

The codes generated by compiler, assembler, and linker are generally re-locatable by its nature, mean to say, the starting location of these codes are not determined, means, they can be anywhere in the computer memory. Thus, the basic task of loader is to find/calculate the exact address of these memory locations.

1.1.2 The Analysis-Synthesis Model of Compilation

There are two parts of compilation:

- **Analysis:** The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- **Synthesis:** The synthesis part constructs the desired target program from the intermediate representation.

1. Analysis of Source Program

In compiling, analysis consists of three phases:

1. **Linear/Lexical Analysis:** 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. The lexical analysis is performed by scanner. The working of the scanner is as follows:
   - The scanner reads characters from the source program.
   - The scanner groups the characters into lexemes (sequences of characters that "go together").
   - Each lexeme corresponds to a token; the scanner returns the next token (plus maybe some additional information) to the parser.
   - The scanner may also discover lexical errors (e.g., erroneous characters).

   The definitions of what is a lexeme, token, or bad character all depend on the source language.
2. **Syntax/Hierarchical 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.
3. **Semantic Analysis:** The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. An important component of semantic analysis is type checking. It may also annotate and/or change the abstract syntax tree (e.g., it might annotate each node that represents an expression with its type.

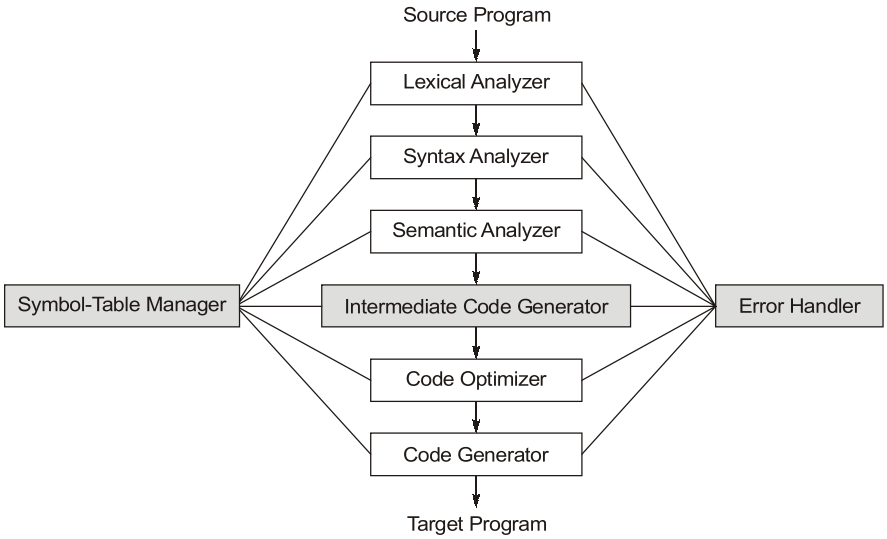
2. Synthesis of Source Program

In compiling, synthesis consists of three phases:

1. **Intermediate Code Generation:** After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation should have two important properties; it should be easy to produce, and easy to translate into the target program. The intermediate code can have a variety of forms.
2. **Code Optimization:** In the optimization phase, the compiler performs various transformations in order to improve the intermediate code. These transformations will result in faster-running machine code.
3. **Code Generation:** The final phase in the compilation process is the generation of target code. This process involves selecting memory locations for each variable used by the program. Then, each intermediate instruction is translated into a sequence of machine instructions that performs the same task.

1.2 COMPILER STAGES

In general the process of compiler is divided into following six stages/phases. Each phase interact with symbol table administrator and error handling.



The first three phases, forming the bulk of the analysis portion of a compiler and the last three phases, forming the bulk of the synthesis portion of a compiler.

1. Lexical Analysis

It is a program which categorizes character sequence into lexemes and produces a sequence of tokens as an output for parser. Lexeme is a sequence of characters from the input program. Token is just a representative for the bodies that form the text of the program.

LEXEMES —Pattern Rule which tells when a sequence of characters make a token→ TOKEN

For example **integer** is a lexeme, i, n, t, e, g, e, r is the rule, and integer is its token. Similarly **madeeasy** is a lexeme, any sequence of alphanumeric starting with a letter is rule, and finally **madeeasy** becomes a token.

Let us consider the following example.

While (a >= b) a = a – 2;

We can represent it in the form of lexemes and tokens as under.

Lexemes	Tokens	Lexemes	Tokens
while	WHILE	a	IDENTIFIER
(	LPAREN	=	ASSIGNMENT
a	IDENTIFIER	a	IDENTIFIER
>=	COMPARISON	-	ARITHMETIC
b	IDENTIFIER	2	INTEGER
)	RPAREN	;	SEMICOLON

Let us also consider the following piece of code.

```
/* C program by madeeasy*/
main()
{
printf ("madeeasy/n");
}
```

Let us suppose that the length of words of memory is 4 bytes, and then the starting form of the input code is as follows:

/	*		C		p	r	o	g	r	a	m		b	y		m	a	d	e	e	a	s	y		*	/
	m	a	i	n	(	)		{		p	r	i	n	t	f	(	“	m	a	d	e	e	a	s	y	/
n	”	)	;		}																					

Lexical analyzer reads the input character one by one. Then according to the regulation of lexical grammar of C language, the character stream is grouped into different token and so on, it becomes the stream of tokens. In C language, the tokens can be KEYWORDS, IDENTIFIERS, INTEGERS, REAL NUMBERS, a notation of SINGLE CHARACTER, COMMENTS, and CHARACTER STRINGS, etc. For example, the lexical analyzer starts its work with reading the first word of the input. It reads the “/”, it knows that this is not a letter, rather it is an operator. However, as expression with operator is missing, so it continues its reading to the second character to see if it is “*” or not. If it is not “*”, then it is wrong. Otherwise it knows that the combination of “/” and “*” forms the identification of the start of comment line, and all the character string before the identification of the end of comment line, i.e., the combination of “*” and “/” is a comment.

Example: The number of tokens in the FORTRAN statement `DO 10 I = 1, 25` is seven.

KEYWORD = ‘DO’; STATEMENT-LEBEL= ‘10’; IDENTIFIER= ‘I’; OPERATOR = ‘=’ ; CONSTANT = ‘1’; COMMA= ‘,’ and CONSTANT = ‘25’.

2. Syntax Analysis (Parsing)

It is a program that takes sequence of tokens as an input from lexical analysis phase and classify it as an expression with their expression-type. The expression is based on the rules of the source language. For example, consider the following piece of code written in source language.

$$x_1 = x_2 * x_3 + x_4$$

Lexical analyzer converts this code in following frame:

Identifier (x_1) assignment operator (=) identifier (x_2) binary operator (*) identifier (x_3) binary operator (+) identifier (x_4). Let the rules of expression in source language is:

1. CONSTANT and IDENTIFIERS are expressions; and
2. If A and B are two expressions then A BINARY-OPERATOR B is also an expression.

Based on above rules, syntax analyzer performs the following action.

A = IDENTIFIER (2) and B = IDENTIFIER (3) – Both are expressions because from rule (1) IDENTIFIERS are expressions. Its type is “expression”.

C = A BINARY-OPERATOR (1)B – It is an expression because from rule (2), if A and B are two expressions then A BINARY-OPERATOR B is also an expression. Its type is “expression”.

D = IDENTIFIER (4) – It is expression because from rule (1) IDENTIFIERS are expression. Its type is “expression”.

E = C BINARY-OPERATOR (2) D – It is an expression because from rule (2), if A and B are two expressions then A BINARY-OPERATOR B is also an expression. Its type is “expression”.

Finally, we get:

id ASSIGNMENT-OPERATOR EXPRESSION – It is an expression and its type is “assignment”.

The form of expressions can be best presented by a parse tree or a syntax tree.

NOTE: The main difference between lexical analyzer and syntax analyzer is that: lexical analyzer very rarely uses recursion while syntax analyzer almost always uses recursion.

3. Semantic Analysis

It takes input from syntax analysis phase generally in the form of parse table. It decides that whether the input has properly defined according to the rules of source language. It performs two actions: (1) type checking; and (2) type coercion based on type rules. In general, type rules are of two types: (1) expression-based type rules; and (2) assignment-based type rules.

Expression-based Types Rules		
Expression	**What semantic analyzer determine**	**Type of expression**
Constant	PROPER ACCORDING TO RULE	CONSTANT
Variable	PROPER ACCORDING TO RULE	VARIABLE
Operator	IF BINARY OR UNARY – PROPER ACCORDING TO RULE	OPERATOR

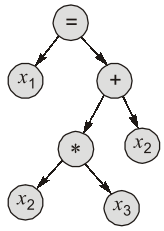
Expression-based Types Rules
Let A is a variable of TYPE(1) and B is properly defined expression of TYPE (2); and if we have TYPE(1) = TYPE(2), where TYPE(1) is an assignable type than A ASSIGN B is a properly defined assignment.

4. Intermediate Code Generation

Combination of lexical analysis, syntax analysis, and semantic analysis is called analysis phase of compiler. After analysis phase, the input source program is fragmented into a parse tree and a symbol table. Using information from these two, compilers starts the procedure of target/object code generation. Generally, code generation is done in two steps: (1) compiler generate code for general machine; and then (2) convert this general machine-based code to the code for particular machine by using a translator. A very general intermediate language is known as three-address code. Generation of code is recursive in nature, and for generating a code we use syntax tree; and therefore the procedure is also called syntax-directed translation.

For example consider the following piece of code: $x_1 = x_2 * x_3 + x_2$

Its equivalent syntax tree is as follows.

Syntax Tree	Equivalent Generated Code
	$t_3 = x_2$ $t_4 = x_3$ $t_1 = t_3 * t_4$ $t_2 = x_2$ $x_1 = t_1 + t_2$

where 't' denotes temporary variable.

5. Code Optimization

The code optimization is a process to improve the code so that the overall run time improves and space requirement reduces. There are lots of methods to improve the code.

6. Code Generation

It is the final stage of compiler process to generate the code for particular machine. Generally in this stage, compiler works on memory management and register allocation.

Summary

- Converting a program written in one language into an identical program written in another language is called compilation.
- Phases of compiler:
  1. **Lexical analysis:** Used to generate lexeme (Tokens)
  2. **Syntax analysis:** Used to generate syntax tree.
  3. **Semantics analysis:** Used to check syntax tree semantically correct or not.
  4. **Intermediate code generation:** Used to generate machine independent code.
  5. **Code optimization:** Improve the overall runtime and space of code.
  6. **Code generation:** Generate the code for particular machine.
- The first 3 phases of compiler are called front-end.
- The last 2 phases of compiler are called back-end.
- Pass is a module in which portions of one or more phases of compiler are combined when a compiler is implemented.
- Types of passes:
  1. One-pass compiler, 2. Two-pass compiler, 3. Multi-pass compiler
- In single pass compiler more memory is needed but time to complete compilation is less.
- In multipass compiler memory requirement is less but time to complete compilation is more.

Student's Assignments

Q.1 A C identifier consists of zero or more letters, digits, and underscores, in any order, but may not begin with a digit. Which pattern will match one?

(a) $[A - z]^+[0 - 9]^*$
(b) $[A - Za - z\_]^+[0 - 9]^+$
(c) $[a - zA - Z\,0 - 9\_]^+$
(d) None of the answers are correct

Q.2 Consider the following statements:

1. A compiler is a translator from one language to another.
2. The output of a compiler needs to be in a low-level language.

Which of the following statements are true?

(a) Only statement 1 is true
(b) Only statement 2 is true
(c) Both (1) and (2) are true
(d) Neither of them are true

Q.3 Consider the following statements:

1. There is no restriction on compiler that the input and output language must be different.
2. Interpreter takes code written in a particular programming language and executes it directly one statement (or expression, etc.) at a time.
3. Both interpreters and compilers could exists for a given language.

Which of the following statements are true?

(a) 1 and 2 (b) 1, 2 and 3
(c) 2 and 3 (d) Neither is true

Q.4 Consider the following statements:

1. Compiler produce executable binary object file whereas an interpreter produce code, both executable can run many times.
2. Before translation, compiler and interpreter reads all of the input file.

Which of the following statements are true?

(a) Only statement 1 is true
(b) Only statement 2 is true
(c) Both (1) and (2) are true
(d) Neither of them are true

Q.5 Consider the following statements:
1. C is a context-free language.
2. C++ is a context-free language.

Which of the following statements are true?
(a) Only statement 1 is true
(b) Only statement 2 is true
(c) Both (1) and (2) are true
(d) Neither of them are true

Q.6 Consider the following statements:
1. C++ is much harder to parse than C.
2. The backend of the compiler handles program parsing.

Which of the following statements are true?
(a) Only statement 1 is true
(b) Only statement 2 is true
(c) Both (1) and (2) are true
(d) Neither of them are true

Q.7 Consider the following statements:
1. Both compiler and interpreter perform syntactic and semantics analysis.
2. Both compiler and interpreter perform optimizations on the input code.
3. Both compilation and interpretation is an off-line process.

Which of the following statements are true?
(a) 1 and 2 only
(b) 1 and 3 only
(c) 2 and 3 only
(d) Neither of them are true

Q.8 Consider the following statements:
1. Abstract syntax tree made in syntax analysis phase contain cycle.
2. $30 = x * 3$ contains lexical error.

Which of the following statements are true?
(a) Only statement 1 is true
(b) Only statement 2 is true
(c) Both (1) and (2) are true
(d) Neither of them are true

Q.9 In a variable declaration, the variable has not previously been declared in the same step.
(a) Compiler handles it in Scanner stage
(b) Compiler handles it in Semantic stage
(c) Compiler handles it in parser stage
(d) None of these

Q.10 Consider the following statements:
S_1: A compiler performs code optimization; a preprocessor does not.
S_2: A compiler performs full syntactic and semantic analysis; a preprocessor does not.

Which of the following statements is correct?
(a) S_1 and S_2 are both true
(b) S_1 is true, S_2 is false
(c) S_1 is false, S_2 is true
(d) S_1 and S_2 are both false

Q.11 Consider the following issue:
I. Simplify the phases
II. Compiler efficiency is improved
III. Compiler works faster
IV. Compiler portability is enhanced.

Which is/are true in context of lexical analysis?
(a) I, II, III (b) I, III, IV
(c) I, II, IV (d) All of these

Q.12 The task of the lexical analysis phase is
(a) To parse the source program into the basic elements or token of the language.
(b) To build a literal table and an identifier table
(c) To build a uniform symbol table
(d) All of the above

Q.13 Assembly code data base is associated with
(a) Assembly language version of the program which is created by the code.
(b) A permanent table of decision rules in the form of patterns for matching with the uniform symbol table to discover syntactic structure.
(c) Consists of full or partial list or the token is as they appear in the program. Created by lexical analysis and used for syntax analysis and interpretation.
(d) A permanent table which lists all key words and special symbols of the language in symbolic form.

Q.14 The term "environment" in programming language semantics is said as
(a) Function that maps a name to value held there
(b) Function that maps a storage location to the value held there

Q.25 Consider the following statements:

S_1 : A lexeme is a sequence of characters in the source program that is matched by pattern for a token.

S_2 : The set of string described by a rule is called pattern associated with a token.

Which of the following is true?

(a) Only S_2
(b) Only S_1
(c) Both S_1 and S_2
(d) Neither S_1 nor S_2

Q.26 Which of the following is true?

(a) Symbol table is constructed during the analysis part of compiler i.e., (front end)
(b) Type checking is done during syntax analysis phase.
(c) Syntax directed definition with only synthesised attribute, always have a order of evaluation.
(d) Both (a) and (c)

Q.27 In a compiler the module that checks the token arrangement against the source code grammar is called _____.

(a) Lexical analyzer (b) Syntax analyzer
(c) Semantic analyzer (d) Code optimizer

Q.28 Match the following errors corresponding to their phase:

Group A

1. Unbalanced parenthesis
2. Appearance of illegal characters
3. Undeclared variables

Group B

A. Syntactic error
B. Semantic error
C. Lexical error

(a) 1 → A, 2 → C, 3 → B
(b) 1 → B, 2 → C, 3 → A
(c) 1 → A, 2 → B, 3 → C
(d) 1 → B, 2 → C, 3 → A

Q.29 Type checking is normally done during

(a) Lexical analysis
(b) Syntax analysis
(c) Semantic phase
(d) Code optimization

Q.30 Consider the following program:

```
1. main( )
2. {    int x = 10;
3.      it (x < 20;
4.      else
5.      y = 20;
6. }
```

When lexical analyzer scanning the above program, how many lexical errors can be produced?

Q.31 Match **Group-I** and **Group-II** and select the correct answer using the codes given below the lists:

Group-I

A. Token
B. Pattern
C. Lexeme

Group-II

1. Sequence of characters in the source program that matches the pattern of a token.
2. A pair consisting of a token name and an optional attribute value.
3. Description of the form that can be accepted.

Codes:

	A	B	C
(a)	1	2	3
(b)	2	3	1
(c)	3	1	2
(d)	1	3	2

Q.32 Consider the following statements:

I. The module that checks every character of the source text is called symbol table in a compiler.
II. Keywords of a language are recognized during lexical analysis.
III. Temporary variables are one of the contents of an activation record.

Which of the above statements is/are correct?

(a) I and III only (b) I only
(c) II and III only (d) I and II only

Answer Key:

1. (d)	**2.** (a)	**3.** (c)	**4.** (d)	**5.** (d)
6. (a)	**7.** (d)	**8.** (d)	**9.** (b)	**10.** (a)
11. (c)	**12.** (d)	**13.** (a)	**14.** (c)	**15.** (a, c)
16. (b)	**17.** (c)	**18.** (d)	**19.** (a)	**20.** (d)

21. (c) **22.** (24) **23.** (b) **24.** (b) **25.** (c)
26. (d) **27.** (b) **28.** (a) **29.** (c) **30.** (0)
31. (b) **32.** (c)

Student's Assignments Explanations

1. (d)

Since identifier may start with zero or more letters, digits and underscores, in any order but should not begin with digit.

Thus, $[a - zA - Z\_]^+ [a - zA - Z\_0 - 9]^*$

Hence, none of the option matches so option (d) is answer.

5. (d)

Both C and C++ are context sensitive langauge because context can make difference.

For example:

int A and int a are treated as two different variables.

9. (b)

Any declared variable is stored in a symbol table which is checked in the semantic phase of the compiler.

10. (a)

A preprocessor is a programs that processes its input data to produce output that is used as input to another program. While compiler performs full syntactic, semantic and code optimization etc.

12. (d)

Lexical analyzer performs the following tasks:
Reads the source program, scans the input characters, group them into lexemes and produces the token.
It also updates symbol table.
Hence option (d) is true.

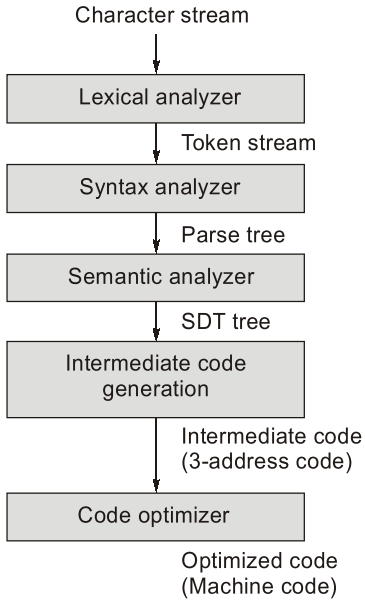
16. (b)

Push down automata is used in syntax analysis phase.

18. (d)

Symbol table is modified by both lexical and semantic phase of a compiler.

19. (a)



20. (d)

Analysis phase {lexical analysis, syntax analysis, semantic analysis} is followed by synthesis phase {intermediate code generation, code optimizer, machine code generation}.

21. (c)

Linking is done by a linker after compilation process.

Compiler can identify token, generates compilation error which can be lexical, syntax or semantic.

22. (24)

```
   1    2  3
main   (  )
4
{
      5   6  7  8  9
    int   a  ,  b  ;
    10 11 12 13 14 15 16
    a  =  5  +  8  +  ;
      17    18    19  20  21 22 23
    printf  (" % d"  ,  a  )  ;
    /* & b = 5; */
24
}
```

Total 24 tokens.