COMPILER DESIGN

For
COMPUTER SCIENCE

COMPILER DESIGN

SYLLABUS

Lexical analysis, parsing, syntax-directed translation. Runtime environments. Intermediate code generation.

ANALYSIS OF GATE PAPERS

Exam Year	1 Mark Ques.	2 Mark Ques.	Total
2003	3	5	13
2004	3	1	5
2005	1	4	9
2006	1	5	11
2007	1	5	11
2008	2	1	4
2009	1	-	1
2010	2	1	4
2011	1	1	3
2012	-	2	4
2013	1	3	7
2014 Set-1	1	1	3
2014 Set-2	2	2	6
2014 Set-3	2	1	4
2015 Set-1	1	2	5
2015 Set-2	2	1	4
2015 Set-3	1	1	3
2016 Set-1	1	2	5
2016 Set-2	1	1	3
2017 Set-1	0	1	2
2017 Set-2	2	2	6

CONTENTS

Topics **Page No**

1. INTRODUCTION TO COMPILERS

1.1 Language processing system 01
1.2 Phases of compiler 02
1.3 Analysis of the program 03
1.4 Lexical analysis 07
1.5 Input Buffering 11

2. PARSING

2.1 Syntax analysis 12
2.2 The Role of the parser 12
2.3 Context free grammars 13
2.4 Derivations 19
2.5 Top down parsing 20
2.6 LL(1) grammar 23
2.7 LR parser 37
2.8 SLR parsing 39
2.9 LALR parsing 45
2.10 Error handling 47

3. SYNTAX DIRECTED TRANSLATION

3.1 Introduction 51
3.2 Synthesized attributes 54
3.3 Syntax trees 56
3.4 Translation schemes 61
3.5 Top-down translation 65
3.6 Run time environment 69

4. INTERMEDIATE CODE GENERATION

4.1 Introduction 74
4.2 Three address code 75

5. GATE QUESTIONS 79

6. ASSIGNMENT QUESTIONS 104

I.C.E. Inspiring Creativity & Endeavour IES | GATE | PSU

1 INTRODUCTION TO COMPILERS

1.1 Language Processing System:

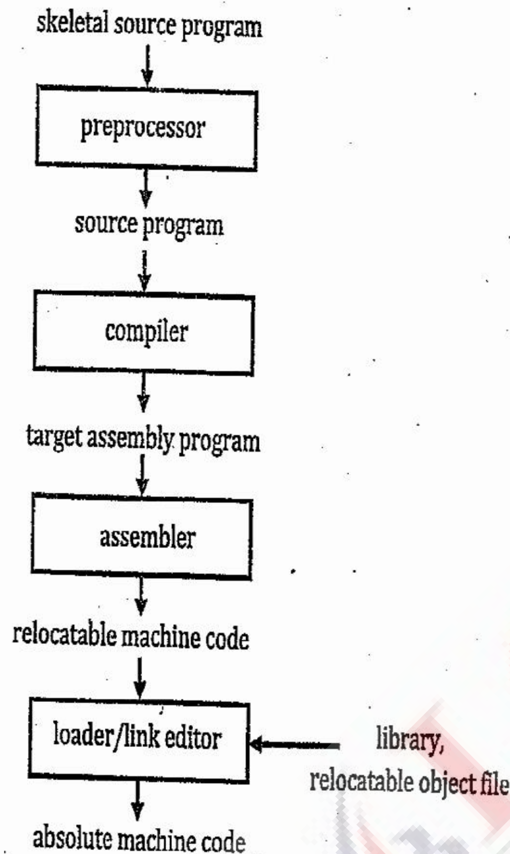


Fig. 1.1 Language Processing System

Preprocessor: A preprocessor produce input to compilers. They may perform the following functions.

1. **Macro processing:**
   A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. **File inclusion:**
   A preprocessor may include header files into the program text.
3. **Rational preprocessor:**
   These preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. **Language Extensions:**
   These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro.

Compilers:

- A compiler is a program that reads a program written in one language the source language and translates it into an equivalent program in another language the target language.
- Compilers are sometimes classified as single pass multi-pass, load and go, debugging or optimizing depending on how they have been constructed or on what function they are supposed to perform.

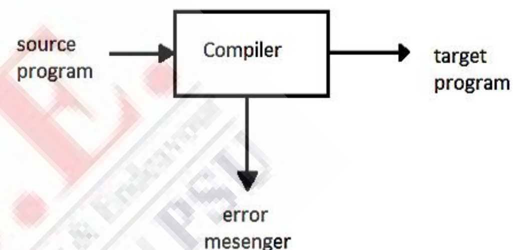


Fig. 1.2 a compiler

Assembler:

- As it is difficult to write machine language instructions, programmers begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. The assemblers were written to automate the translation of assembly language into machine language.

Interpreter:

- An interpreter is a program that appears to execute a source program as if it were machine language.
- Languages such as BASIC, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.
  1. Lexical analysis

© Copyright Reserved by Inspiring Creativity & Endeavour Gate Institute. No part of this material should be copied or reproduced without permission.

2. Syntax analysis
3. Semantic analysis
4. Direct Execution

Advantages:

- Modification of user program can be easily made and implemented as execution proceeds.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent

Disadvantages:

- The execution of the program is slower.
- Memory consumption is more.

1.2 Phases of compiler

- A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. There are two phases of compilation.
  - ❖ Analysis (Machine Independent/ Language Dependent)
  - ❖ Synthesis (Machine Dependent/ Language independent)
- Compilation process is partitioned into no-of-sub processes called 'phases'.
- In practice some of the phases grouped together and the intermediate representation between the grouped phases need not explicitly constructed. A common division into phases is described below.
- The **front end** includes all analysis phase and the intermediate code generator with a certain amount of code optimization as well.
- The back end includes the code optimization phase and the final code generation phase.

The front end includes the source program and produces intermediate code which the back end synthesizes the target program from the intermediate code.

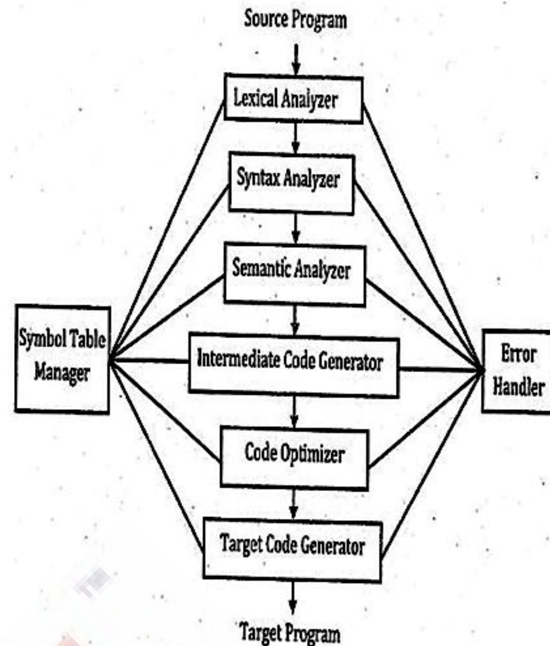


Fig 1.3 Phases of a compiler

1.2.1 Lexical analysis:

- This is the initial part of reading and analyzing the program text.
- The text is read and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

1.2.2 Syntax analysis:

The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc. are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

1.2.3 Semantic Analysis :

Syntax analyzer will just create parse tree. Semantic Analyzer will check actual meaning of the statement parsed in parse tree. Semantic analysis can compare information in one part of a parse tree to that in another part (e.g., compare reference to variable agrees with its

© Copyright Reserved by Inspiring Creativity & Endeavour Gate Institute. No part of this material should be copied or reproduced without permission.

declaration, or that parameters to a function call match the function definition).

1.2.4 Intermediate code generation :

The output of the syntax analyzer is some representation of a parse tree. the intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

1.2.5 Code optimization:

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

1.2.6 Code generation:

The last phase of translation is code generation. A number of optimizations to reduce the length of machine language program are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

1.3 Analysis of the Source Program

- Linear or Lexical analysis, in which 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.
- Hierarchical or Syntax analysis, in which characters or tokens are grouped hierarchically into nested collections with collective meanings.
- Semantic analysis, in which certain checks are performed to ensure that the components of a program fit together meaningfully.

1.3.1 The Analysis-Synthesis Model of Compilation

There are two parts of compilation:

1) **Analysis** : It breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
2) **Synthesis :** It constructs the desired target program from the intermediate representation.

- Out of the two parts, synthesis requires the most specialized techniques.
- During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called as tree.

Example, a syntax tree for an assignment statement is shown in Fig. 1.4

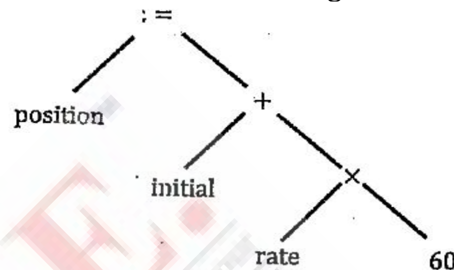


Fig. 1.4 Syntax tree for position: = initial + rate × 60.

1.3.2 The Context of a Compiler

- In addition to a compiler, several other programs may be required to create an executable target program.
- A source program may be divided into modules stored in separate files.
- The task of collecting the source program is sometimes entrusted to a distinct program, called as Preprocessor. The preprocessor may also expend shorthand's called macros, into source language statements.

1.3.3 Symbol-Table Management

- A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly,
- Symbol table is a Data Structure in a Compiler used for Managing information about variables & their attributes,

© Copyright Reserved by Inspiring Creativity & Endeavour Gate Institute. No part of this material should be copied or reproduced without permission.

1.3.4 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.

- The lexical phase can detect errors where the character remaining in the input do not form any token in the language.
- Where token stream violates the rule that violates structure rules of language determined by syntax analysis phase.
- The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors where the token stream violates the structure rules (syntax) of the Language are determined by the syntax analysis phase.

1.3.5 Retargeting

It is a process of creating more & more compilers for the same source language, but for different machines.

1.3.6 Assemblers

Some compiler produces assembly code that's passes to an assembler for further processing. Other compilers perform the job of the assembler, producing relocatable machine code that can be passed directly to loader/link editors.

Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory addresses. A typical sequence of assembly instructions might be

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

This code moves the contents of the address a into register1, then adds the constant 2 to it treating the content of register1 as a fixed point number and finally stores the result in the location named by b thus it computes b=a+2

1.3.7 Two-Pass Assembly

The simplest form of assembler makes two passes over the input, where a pass consists of reading an input file once. In the first pass, all the identifiers that denote storage locations are found and stored in a symbol table.

In the second pass, the assembler scans the input again. This time, it translates each operation code into the sequence of bits representing that operation in machine language, and it translates each identifier representing a location into the address given for that identifier in the symbol table. The output of the second pass is usually relocatable machine code, meaning that it can be loaded starting at any location L in memory; i.e., if L is added to all addresses in the code, then all references will be correct. Thus, the output of the assembler must distinguish those portions of instructions that refer to addresses that can be relocated.

1.3.8 Preprocessor

Preprocessor produce input to compiler.
They may perform following function.

- **Macro processing :** A Preprocessor may allow a user to define macro that are shorthand's for longer constructs.
- **File inclusion :** A pre-processor may include header file into the program text. For example : C processor causes the context of the file <global.h> to replace the statement # include <global> when it processes a file, containing this statement.
- **Rational pre-processor :** These processor augment older languages with

© Copyright Reserved by Inspiring Creativity & Endeavour Gate Institute. No part of this material should be copied or reproduced without permission.

more modern flow of control and data structure facilities for example, such a pre-processor might provide user with built in macros for construct like while statement or if-statement , where none exists in the programming language itself.

- **Language Extensions :** These processors attempt to add capabilities to the language by what amounts to built in macros

1.3.9 Phases vs. Passes:

Scanning of complete text from Left hand side to Right side is called as "Pass" For translations, only use passes i.e. implemented more 2 passes.
- Scan
- Translate
Multi pass - requires less space, but it is slower.
Single pass - It is faster, but requires more space.

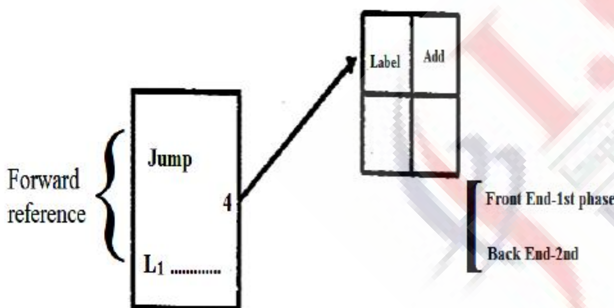


Fig. 1.4

1.3.10 The Analysis Phases

As translation progresses, the compiler's internal representation of the source program changes . We illustrate these representations by considering the translation of the statement.
position:= initial + rate * 60 -------------(1.1)

1.3.11 Lexical Analyzer

The lexical analysis phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword (if, -while, etc.), a punctuation character, or a multi-character operator like : = The character sequence forming a token is called the lexeme for the token.
Certain tokens will be augmented by a "lexical value". For example, when an identifier like rate is found, the lexical analyzer not only generates a token, say id, bat also enters the lexeme rate into the symbol table, if it is not already there. The lexical value associated with this occurrence of id points to the symbol-table entry for rate.
In this section, we shall use id_1, id_2, and id_3 for position, initial, and rate, respectively, emphasize that the internal representation of an identifier is different from the character sequence forming the identifier.
The representation of assignment statement (1.1) after lexical analysis is therefore suggested by:
$Id_1 = id_2 + id_3 * 60$ -------------(1.2)

1.3.12 Syntax Analysis Phase

i) The syntax Analysis Phase: The syntax analysis phase imposes a hierarchical structure on the token stream, shown below

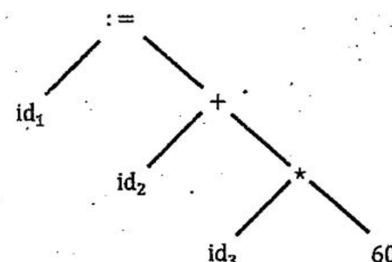


Fig. 1.5

ii) 'The Semantic Analysis Phase: Daring the semantic analysis, it is considered that in our example all identifiers have been declared to be real and that 60 by itself is assumed to be an integer. 'Type checking of syntax tree reveals that * is applied to a real rate and an integer, 60. The general approach is to convert the integer into a real. This has been achieved by creating an integer into a real

© Copyright Reserved by Inspiring Creativity & Endeavour Gate Institute. No part of this material should be copied or reproduced without permission.

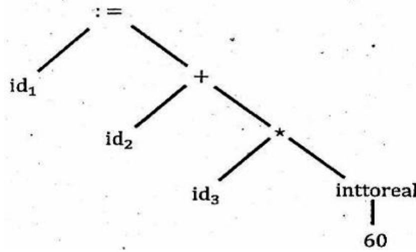


Fig. 1.6

1.3.13 The Synthesis Phases

i) **Intermediate Code Generation (or) ICG**

- 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

Symbol Table

1	position	…….
2	Initial	…….
3	rate	……..
4		

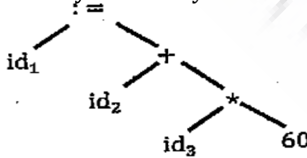
Position: initial + rate*60

↓

Lexical analyzer

$Id_1{:=}id_2{+}id_3{*}60$

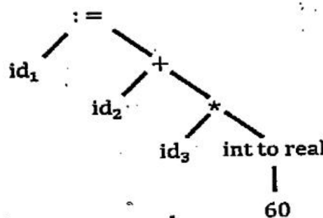
Syntax analyzer



↓

Semantic analyzer

↓



↓

Intermediate code generator

↓

Temp 1 : int to real(60)

Temp 2: id3*temp1

Temp 3 = id3 + temp2

↓

Code optimizer

Temp 1: = id3*60.0

Id1: = id2 + temp1

Code optimizer

↓

MOVF id3, R2

MULF #60.0, R2

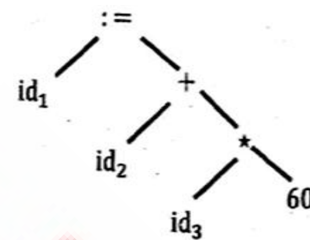
MOVF id2, R1

ADDF R2, R1

MOVF R1, id1

Fig.1.7

(a)



(b)

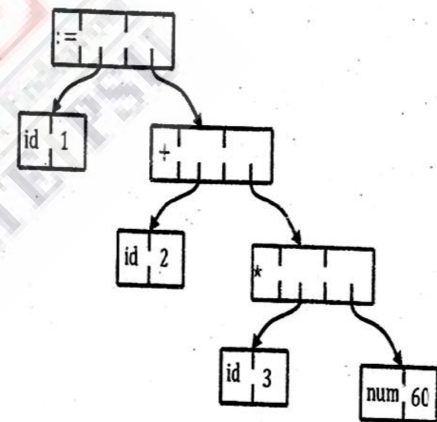


Fig 1.8 Data structure in (b) is for the tree in (a)

ii) **Code Optimization :-**

- This phase attempts to improve the intermediate code , so that faster running 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

© Copyright Reserved by Inspiring Creativity & Endeavour Gate Institute. No part of this material should be copied or reproduced without permission.

GATE QUESTIONS

Topics	Page No

1. LEXICAL ANALYSIS 80

2. PARSING TECHNIQUES 83

3. SYNTAX DIRECTED TRANSLATION 94

4. CODE GENERATION AND OPTIMIZATION 99

1 LEXICAL ANALYSIS

Q.1 The number of tokens in the following C statements
Print f("i=%d, & i = % x", i & i); is
a) 3 b) 26
c) 10 d) 21
[GATE- 2000]

Q.2 Which of the following is not an advantages of using shared, dynamically linked libraries as opposed to using statically linked libraries?
a) Smaller sizes of executable files
b) Lesser overall page fault rate in the system
c) Faster program startup
d) Existing programs need not be relinked to take advantage of newer versions
[GATE -2003]

Q.3 Consider a program P that consists of two source modules M_1 and M_2 contained in two different files. If M_1 contains a reference to a function defined in M_2, the reference will be resolved at
a) edit time b) Compile time
c) link time d) load time
[GATE- 2004]

Q.4 Which of the following statements are true?
1. There exist parsing algorithms for some programming languages whose complexities are less than $\theta(n^3)$.
2. A programming language which allows recursion can be implemented with static storage allocation.
3. No L-attributed definition can be evaluated in the framework of bottom-up parsing.
4. Code improving transformations can be performed at both source Language & intermediate code level

a) 1 and 2 b) 1 and 4
c) 3 and 4 d) 1, 3 and 4
[GATE -2009]

Q.5 Which data structure in a compiler is used for managing information about variables and their attributes?
a) Abstract syntax-tree
b) Symbol table
c) Semantic stack
d) Parser table
[GATE -2010]

Q.6 In a compiler, keyboards of a language are recognized during
a) Parsing of the program
b) The code generation
c) The lexical analysis of the program
d) Data flow analysis
[GATE -2011]

Q.7 Match the following:
List-I
(P) Lexical analysis
(Q) Top down parsing
(R) Semantic analysis
(S) Runtime environments
List-II
(i) Leftmost derivation
(ii) Type checking
(iii) Regular expressions
(iv) Activation records
a) P ↔ i, Q ↔ ii, R ↔ iv, S ↔ iii
b) P ↔ iii, Q ↔ i, R ↔ ii, S ↔ iv
c) P ↔ ii, Q ↔ iii, R ↔ i, S ↔ iv
d) P ↔ iv, Q ↔ i, R ↔ ii, S ↔ iii
[GATE -2016]

Q.8 Match the following according to input from the left column to the

© Copyright Reserved by Inspiring Creativity & Endeavour Gate Institute. No part of this material should be copied or reproduced without permission.

EXPLANATIONS

Q.1 (c)

A token is strictly a string of characters that the compiler replaces with another string of characters. Token are defined as a sort of search and replace. Tokens can be used for common phrases, page elements, handy shortcuts, etc. Tokens are defined with either the define or replace tags and can use single or paired tag syntax.

Examples

<define%bgcolor%#ccffcc>
<define (c) ©>
<define [email] joe@cool.com>,
<:replace [mailto] [email]:>

Q.2 (d)

Let's take a look at the advantages of the dynamic linking.

Advantages of Dynamic Linking

1. A stub is a piece of code which is used to stand in for functionality of some other program. This stub is included in the image or each binary routine reference is dynamic linking.
2. without dynamic linking, a library when replaced with a new version then all the programs that has reference to the library will be relinked to gain access. Dynamic linking avoids all such conditions.

Q.3 (c)

To link the modules, firstly the modules need to be individually compiled.

Once, the modules are compiled, linking is done. The linking is done at the link time.

Q.4 (b)

Statement 1 True: Parsing algorithms exists for some of the programming languages whose. Complexities are even less than θ(n^3).

Statement 2 False: A programming language allowing recursion cannot be implemented with static storage allocation.

Statement 3 False: L-attributed definition can be evaluated in framework of bottom up parsing.

Statement 4 True: Code improvement can be done at both source language level and intermediate language level.

Q.5 (b)

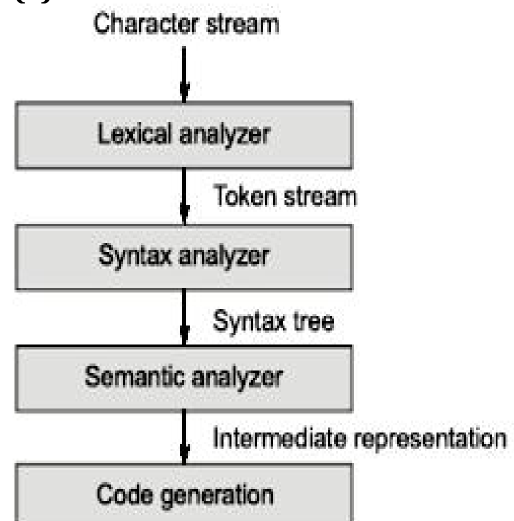
Example Symbol table

Symbol table is a data structure in a compiler. It was devised for the purpose that it can be used to manage information about variable and their attributes.

Q.6 (c)

The lexical analysis of program.

Q.8 (c)



© Copyright Reserved by Inspiring Creativity & Endeavour Gate Institute. No part of this material should be copied or reproduced without permission.

ASSIGNMENT QUESTIONS

Q.1 Cross compiler is a compiler
a) Which is written in a language that is different from the source language
b) That generates object code for its host machine.
c) Which is written in a language that is a same as the source language
d) That runs on one machine but produces object code for another machine

Q.2 Incremental-compilers is a compiler
a) Which is written in a language that is different from the source language
b) That generates object code for its host machine.
c) Which is written in a language that is a same as the source language
d) That allows a modified portion of a program to be recompiled?

Q.3 For which of the following reasons, an interpreter is preferred to a compiler?
a) It takes less time to execute.
b) It is much helpful in the initial stages of program development.
c) Debugging can faster and easier.
d) It needs less computer resources

Q.4 For which of the following reasons, a compiler is preferable to an interpreter?
a) It can generate stand-alone programs that often take less time for execution.
b) It is much helpful in the initial stages of program development.
c) Debugging can be faster and easier.
d) If one changes a statement, only that statement needs recompilation.

Q.5 The cost of developing a compiler is proportional to the
a) Complexity of the source language
b) Complexity of the architecture of the target machine
c) Flexibility of the available instruction set
d) None of the above

Q.6 An ideal compiler should
a) Be smaller in size
b) Take less time for compilation
c) Be written in a high level language
d) Produce object code that is smaller in size and executes faster

Q.7 An optimizing compiler
a) Is optimized to occupy less space
b) Is optimized to take less time for execution
c) Optimizes the code
d) None of the above

Q.8 In a compiler, grouping of characters into tokens is done by the
a) Scanner
b) Parser
c) Code generator
d) code optimizer

Q.9 whether a given pattern constitutes a token or not
a) Depend on the source language
b) Depends on the target language
c) Depends on the compiler
d) None of the above comments true

© Copyright Reserved by Inspiring Creativity & Endeavour Gate Institute. No part of this material should be copied or reproduced without permission.

EXPLANATIONS

Q.25 (a)
Consider the string a+a*a. it can be derived as
E→E+E→E+E*E→a+E*E→a+a*E→a+a*a
Or
E→E*E→E+E*E→a+E*E→a+a*E→a+a*a
Since we know a string that can be derived in more than one way, the given grammar is ambiguous.

Q.27 (a)
abc can be derived as follows.
S → Abc → abc using {Ab → ab}
As we see, any production from the start state has to end in c. So aab is impossible. Option (c) and (d) also not possible.

Q.28 (d)
Generates some of the strings that can be derived from the start state and verify that they fall into the category covered by option (d).

Q.47 (c)
If memory space is not the constraint, then by increasing the number of bins to K, the access time can be reduced by a factor of K. so, average number of item in a bin will decrease as the number of bins increases. In the case of list, access time will be proportional to n , the number of items, but we will be using as much memory space as is abQ.utely necessary. In the case of search tree implementation, the access time will be logarithmic.

Q.69 (b)
Any shift-reduce parser typically word by shifting entries onto the stack. If a handle is found on the top of the stack, it is popped and replaced by the corresponding left hand side of the production. If ultimately we have only the starting non-terminal on the stack, when there are no more tokens to be scanned, the parsing will be successful. So, it is bottom-up.

Q.94 (a)
The right most derivation of the string xxxxyzz is,
S→xxW→xxSz→xxxxWz→xxxxSzz →xxxxyzz.
A shift reduce parser, performs the right-most derivation in reverse. So, first it reduces the Y to s, by the production S → y. As a consequences of this, 2 is immediately printed. Next, Sz is reduce to W, by the production W → Sz. So, 3 will be printed. Proceeding this way, we get the output string 23131.

Q.95 (c)
It is because, it is equivalent to recognizing wcw, where the first w is the declaration and the second is its use. wcw is not a CFG.

© Copyright Reserved by Inspiring Creativity & Endeavour Gate Institute. No part of this material should be copied or reproduced without permission.