Syllabus Content:

16.2 Translation software

Candidates should be able to:

- **Show understanding of how an interpreter can execute programs without producing a translated version.**
- **Show understanding of the various stages in the compilation of a program Including:**
  - **lexical analysis**
  - **syntax analysis**
  - **code generation**
  - **optimisation**
- **Show understanding of how the grammar of a language can be expressed using syntax diagrams or Backus-Naur Form (BNF) notation**
- **Show understanding of how Reverse Polish Notation (RPN) can be used to carry out the evaluation of expressions**

Translation Software:

Why translators, compilers, interpreters, and assemblers are needed

	Assembler	Compiler	Interpreter
Source program written in	assembly language	high-level language	high-level language
Machine dependent	yes	no	no
Object program generated	yes, stored on disk or in main memory	yes, stored on disk or in main memory	no, instructions are executed under the control of the interpreter
Each line of the source program generates	one machine code instruction, one to one translation	many machine code instructions, instruction explosion	many machine code instructions, instruction explosion

Assemblers:

An assembler is a program which converts the low level assembly programming language into machine code.

The assembler does this by converting the one word assembly instructions into an **opcode,** e.g. converting **AND** to **0010.**

It also allocates memory to variables, often resulting in an operand.

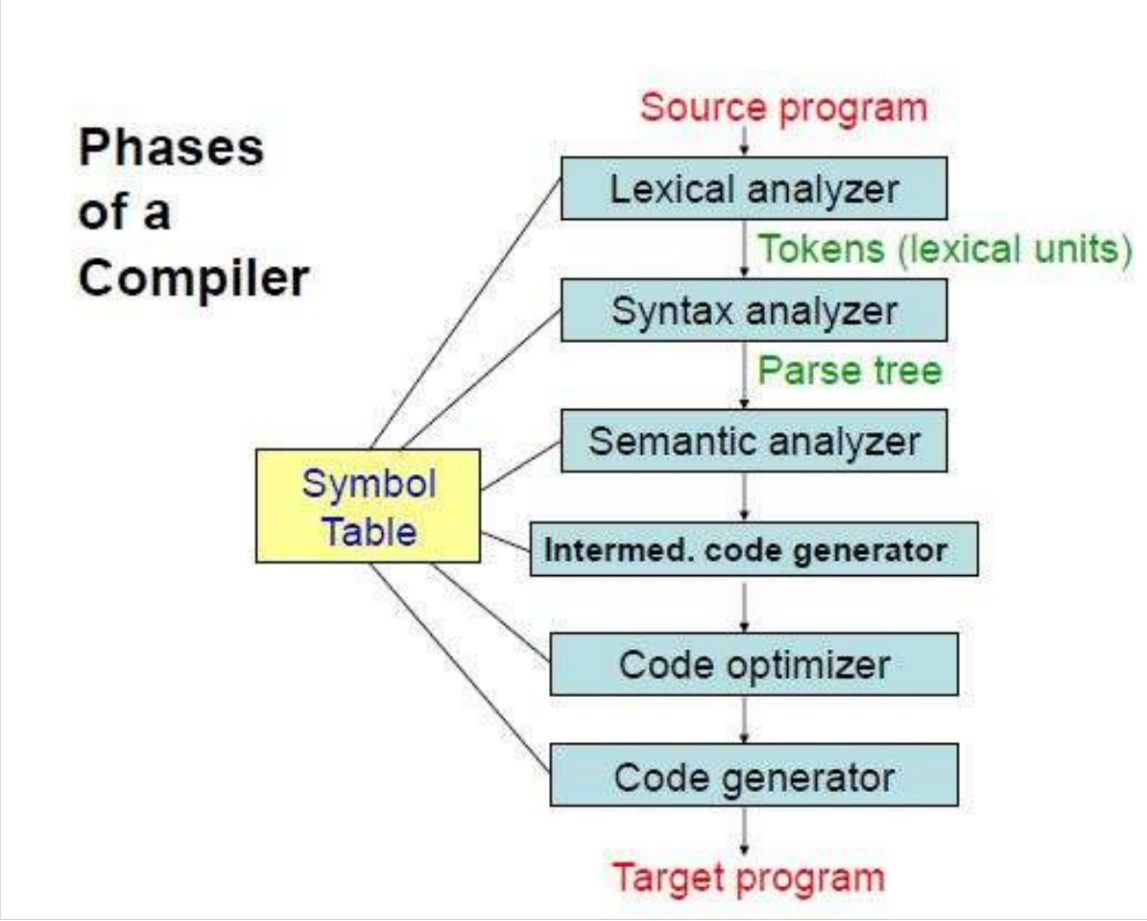
AND A	→	0010	0001
LOD B	→	0110	0010

Compiler software

A compiler is used when high level programming languages are converted into machine code, ready to be executed by the CPU.

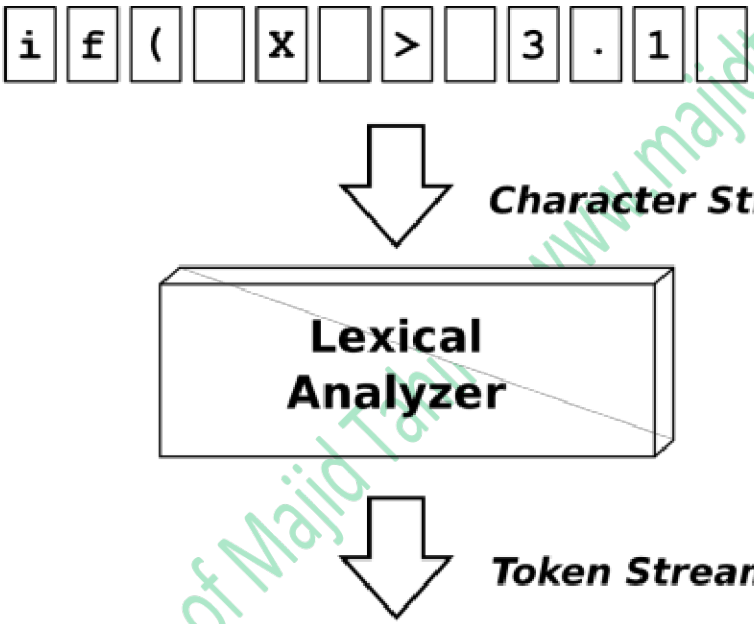
There are four main stages of compilation:

1. **Lexical analysis**
2. **Syntax analysis**
3. **Semantic analysis**
4. **Code generation**



Lexical analysis

- Comments and unneeded spaces are removed.
- Keywords, constants and identifiers are replaced by **'tokens'**.
- A symbol table is created which holds the addresses of variables, labels and subroutines.



KEYWORD	BRACKET	IDENTIFIER	OPERATOR	NUMBER
"if"	"("	"x"	">"	"3.1"

Symbol Table:

The **lexical analyse**r builds up a symbol table which contains all the identifiers found in the source code and - if applicable-their data type. **Constants** have their **value stored**. An **array** has its **lower and upper bound** recorded.

Most compilers must make **two passes** through the source code in order to complete the symbol table entries.

The first pass inserts each new variable or constant name.
A second pass through the source code can establish the actual memory address to be used for that identifier.

For a large program, the symbol table will contain hundreds of entries and it is important that fast access is possible to any identifier entry. This can be done by construction the symbol table as hash table with a hash key generated for each entry.

Consider the following program statement

```
// Calculate discount rate
IF Discount = True THEN
        Discount Rate = 5
        ELSE
        Discount Rate = 0
END IF
```

Lexical analyser produce from this statement using the keyword table in Table below

Keyword	Token
REPEAT	∇
UNTIL	∇∇
IF	□
THEN	□∇
ELSE	□∇∇
END IF	□∇∇∇
True	♦
False	♦♦

The source code is scanned:

- The comment statement is removed
- All 'whitespaces' is removed
- The two variable are added to the symbol table
- The output string contains pointers to each of these variables
- The keywords are looked up in the keyword table and replaced by their matching table

The lexical analysis produces the following output string and symbol table

□ Pointer to 01CD = ♦□∇^01CF=5□∇∇^01CF=0□∇∇∇

Note ' **^01CF means** ' pointer 01CF

Identifier	Data Type	Memory address
Discount	BOOLEAN	01CD
DiscountRate	INTEGER	01CF

Syntax analysis

- Tokens are checked to see if they match the spelling and grammar expected, using standard language definitions. This is done by parsing each token to determine if it uses the correct syntax for the programming language.
- If syntax errors are found, error messages are produced.

For example compiler will check the following statement and report that it is missing the closing bracket.

```
Console.WriteLine(number & "  WRONG ENTRY, Please Re-Enter again"
```

At the syntax analysis stage, statements which are incorrectly formed will be identified. These statements do not conform to the rules of the language.

When an error is found the compiler will continue – unlike an interpreter – and all error will be reported either on screen or in a listing file. The programmer must then return to the source code, make the changes and re-compile this code.

Semantic analysis

- Variables are checked to ensure that they have been properly declared and used.
- Variables are checked to ensure that they are of the correct data type, e.g. real values are not being assigned to integers.
- Operations are checked to ensure that they are legal for the type of variable being used, e.g. you would not try to store the result of a division operation as an integer.

Code generation & code optimisation

- Machine code is generated.
- Code optimisation may be employed to make it more efficient/faster/less resource intense.

Consider the following statement in High-level program

```
TriangleArea = Base * Height  /  2
```

Identifier	Data type	Memory address
Base	SINGLE	0F03
Height	SINGLE	0F04
TriangleArea	SINGLE	0F05

This code translation process then produces the following **corresponding machine code**. It is shown as **an assembly language instruction for readability**

```
LOAD    0F03
MULT    0F04
DIV     2
STORE   0F05
```

This code illustrates that any one high level language statement will be translated into several machine code instruction.

The code generation process will need to make references to:

- The information stored in symbol table
- Code contained in various program libraries, for example when the source code statement uses built in function of language.

Interpreter software:

Before high level programming languages can be run, code is converted by an interpreter, one line at a time, into machine code, which is then executed by the CPU.

The interpreter software will translate each statement in program in sequence and execute these program statements until an error is found. When the first error is found, the execution of program terminates and programmer will fix the error by correcting the error.

Syntax Diagram

Syntax diagrams were used extensively in early textbooks for high-level programming languages. The following example illustrates the diagrams needed to be defined the syntax allowed for an identifier.

Figure1 below defines digit character. The syntax diagram is always read from left to right and is designed to show here that a digit is always one of ten characters shown.

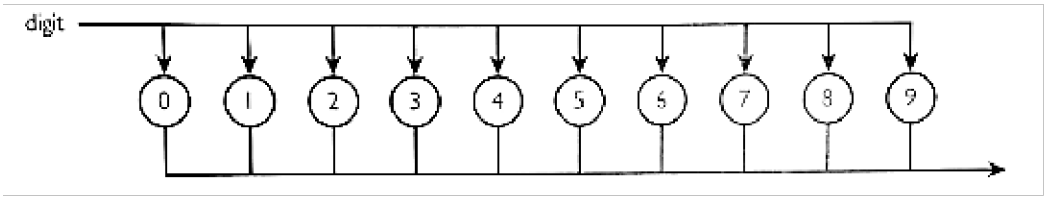


Figure 1 defines a digit

Following **figure 2** defines a **'letter'** We conclude that letter is always exactly one of the upper or lower case letters.

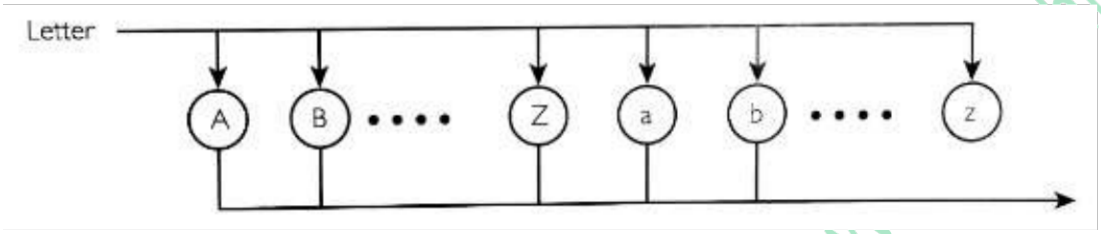


Figure 2 defines a **'letter'**

The previous two definitions can be used to define **'identifier'** as shown in **figure3.**

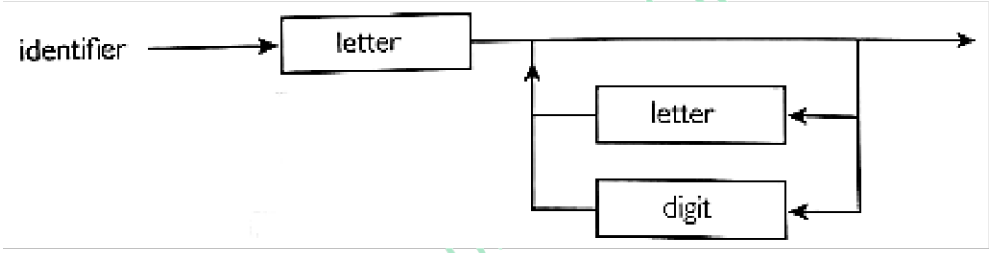


Figure3 defining and **'identifier'**

Figure3 explains why the identifier in **TABLE below** are **VALID or INVALID**

Suggested Identifier	VALID or INVALID?	Explanation
P	VALID	
MyObject	VALID	
8index	INVALID	Must start with at least one letter character
Count	VALID	
My_Object	INVALID	Contains an **underscore** letter which has not been defined
Loop5times	VALID	

TABLE identifier table

Backus – Naur Form (BNF) notation

Backus-Naur Form (BNF) is a special language, called a **'meta-language'** which is used to describe the syntax and composition of statements which make up a high-level programming language.

Backus-Naur Form (BNF) notation technique used to describe recursively the syntax of

- programming languages
- document formats
- communication protocols – etc.

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<binary digit> ::= 0 | 1

The first statement reads as **<digit>** is defined as **0 or 1 or 2 or … or 8 or 9.**

Since each of the terms on the right-hand side of each rule cannot be broken further 0,1,2, etc are called **'terminator symbols'**

A rule definition may be recursive. Consider the definition to describe identifier names for a programming language.

<letter> ::= A | B | C |… | Z | a | b | c | . . . | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<identifier> ::= <letter>|
<identifier><letter>|<identifier><digit>|

The third rule states that a valid identifier name can be any of:

- a single letter
- a letter followed by one or more letter or digit characters.

The process of analyzing whether or not a given identifier is valid (following the give rule) is called 'parsing' the expression.

Consider the identifier name **She1** is Valid?

Figure4 below demonstrates the expression is valid identifier name.

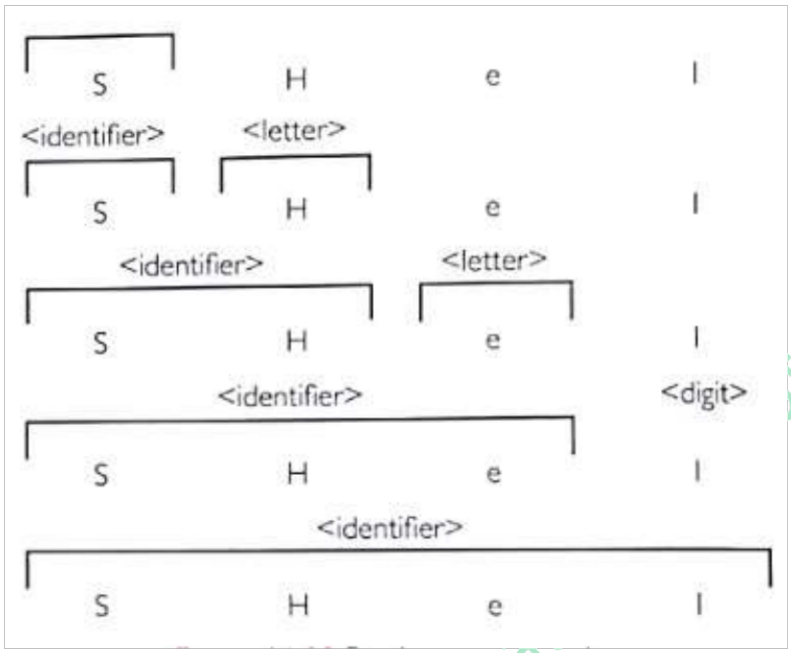


Figure 4 'Parsing an expression'

Consider the identifier name as **1he** is valid?

Figure 5 demonstrates that the expression is not a valid identifier name.

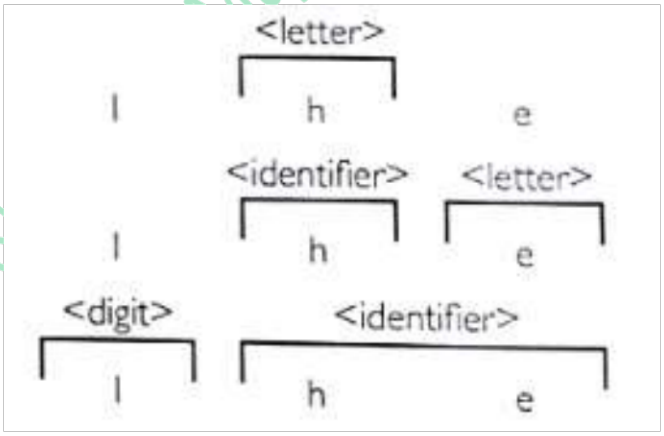


Figure 5 'Parsing an expression'

Reverse Polish notation (RPN)

Infix notation

When we were taught the fundamentals of math, we wore expressions using infix notation, for example:

Area = Length X Width

The operator – the multiplication sign - is positioned between the two operands (Length and Width). The meaning of this expression is clear. However some expression require the use of brackets to convey the order in which the component parts must be evaluated:

Z = (x + y) / 5

The brackets are needed here to make it clear the the sum of x and y must be worked out first.

Reverse Polish (or postfix) notation:

In reverse polish notation, the operand for the expression is written following the two operands

Area = Length Width X

Z = x y + 5 /

Reverse Polish notation has the major advantage over infix in that the meaning for any expression is clear without the use of brackets.

Study carefully the examples in **Table7** below

Infix expression	Postfix expression
(8 - 4) x 5	8 4 - 5 x
(3p + 5) / (p - z)	3 p x 5 + p z - /
2 x 7 - 8 / 4	2 7 x 8 4 / -
7^4 - 9 / 3	7 4 ^ 9 3 / -

Table 7 Infix and Reverse polish notation.

Translators

A translator changes (translates) a program written in one language into an equivalent program written in a different language. For example, a program written using the PASCAL programming language may be translated into a program written in the C programming languages by a translator.

The types of error that may occur in programming code such as: syntax, run time/ execution, logical, linking, rounding, truncation

Syntax	An error that occurs when a command does not follow the expected syntax of the language, e.g. when a keyword is incorrectly spelt	• **Incorrect:** IF A ADN B Then • **Correct:** IF A AND B Then
Runtime/ execution	An error that only occurs when the program is running and is difficult to foresee before a program is compiled and run	• Program requests more memory when none is available, so the program *crashes*
Logical	An error that causes a program to output an incorrect answer (that does not necessarily crash the program)	• An algorithm that calculates a person's age from their date of birth, but ends up giving negative numbers
Linking	Aan error that occurs when a programmer calls a function within a program and the correct library has not been linked to that program	• When the square root function is used and the library that calculates the square root has not been linked to the program
Rounding	Rounding is when a number is approximated to nearest whole number/tenth/hundredth, etc.	• 34.5 rounded to nearest whole number is 35, an error of +0.5
Truncation	Truncating is when a number is approximated to a whole number/tenth/hundredth, etc. nearer zero	• 34.9 truncated to whole number is 34, an error of -0.9

References:

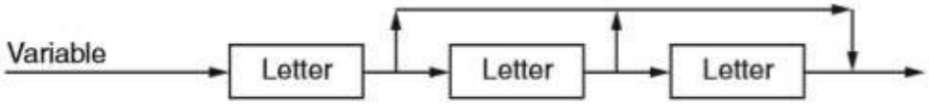
- **Computer Science revision guide by Toni piper**
- **GCE Computer Science by Geraint D. Jones and Mark D. Thomas**
- **Past Papers**

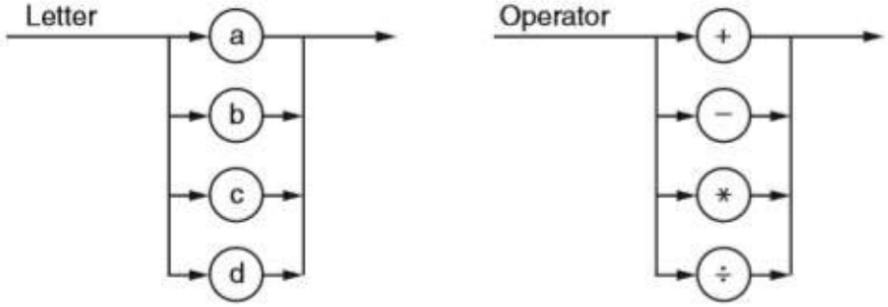
Past Paper Question:

9608/31/M/J/15
Q1.

1 The following syntax diagrams, for a particular programming language, show the syntax of:

- an assignment statement
- a variable
- a letter
- an operator





(a) The following assignment statements are invalid. Give the reason in each case.

(i) a = b + c
Reason
.....
.....[1]
(ii) a = b – 2;
Reason
.....
.....[1]
(iii) a = dd * cce;
Reason

.....
.....[1]

(b) Write the Backus-Naur Form (BNF) for the syntax diagrams shown on the opposite page.

<assignmentstatement> ::=

.....

<variable> ::=

.....

<letter> ::=

.....

<operator> ::=

.....[6]

(c) Rewrite the BNF rule for a variable so that it can be any number of letters.

<variable> ::=

.....[2]

(d) Programmers working for a software development company use both interpreters and compilers.

(i) The programmers prefer to debug their programs using an interpreter.

Give one possible reason why.

.....

.....[1]

(ii) The company sells compiled versions of its programs.

Give a reason why this helps to protect the security of the source code.

.....

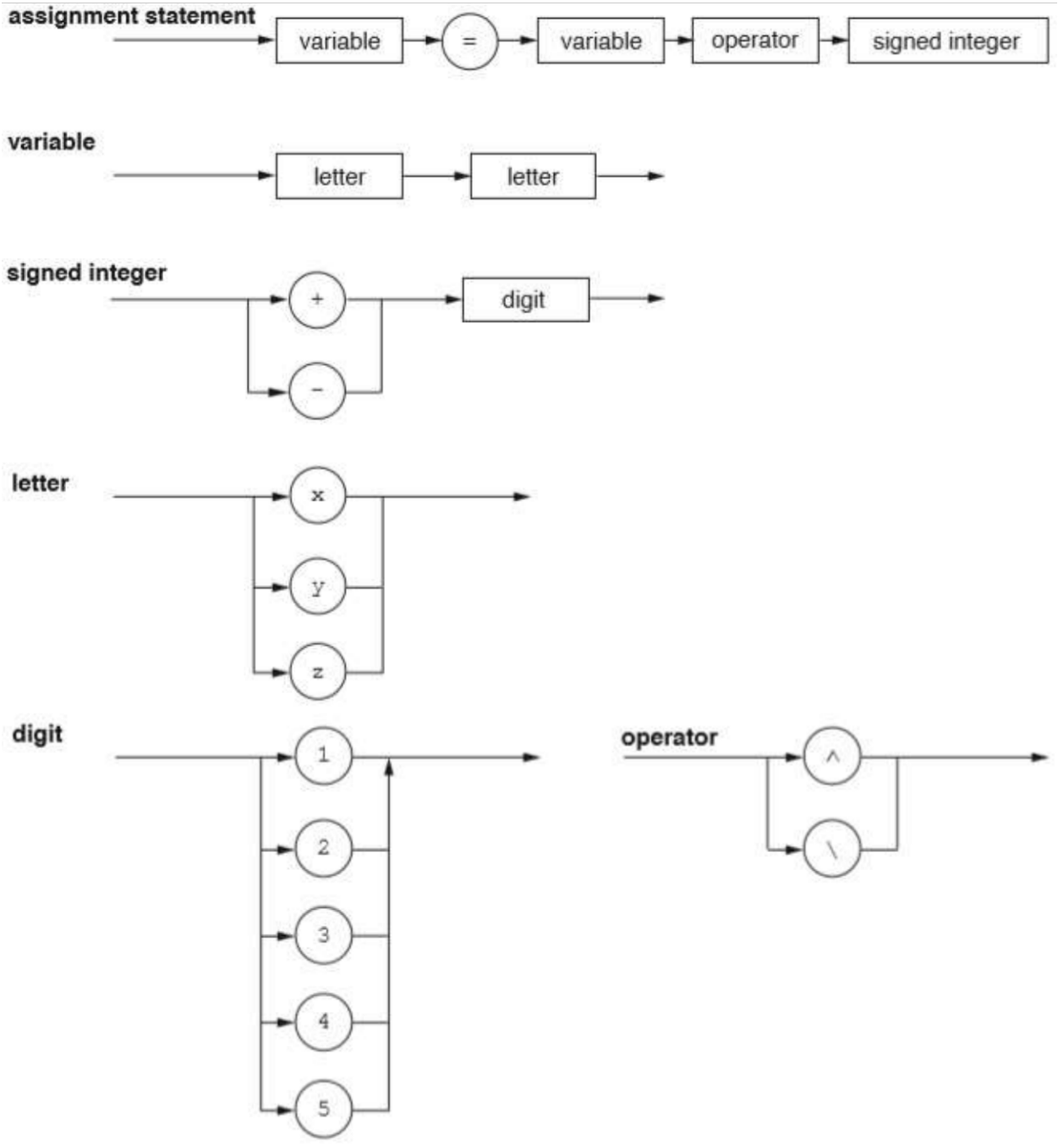
.....[1]

08/31/M/J/18

Q5. The following syntax diagrams show the syntax of:

- an assignment statement
- a variable
- a signed integer
- a letter

- a digit
- an operator



(a) The following assignment statements are invalid. Give the reason in each case.

(i) xy = xy ^ c4

Reason

.....

.....[1]

(ii) zy = zy \ 10

Reason

.....

.....[1]

(iii) yy := xz ^ − 6

Reason

.....

.....[1]

(b) Complete the Backus-Naur Form (BNF) for the syntax diagrams on the opposite page.

<assignment statement> ::=

.....

<variable> ::=

.....

<signed integer> ::=

.....

<operator> ::=

.....[4]

(c) Rewrite the BNF rule for a variable so that it can be any number of letters.

<variable> ::=

.....[2]

Answers:

1 (a) (i)	';' missing	1
(ii)	'2' is not a variable	1
(iii)	'e' is not a valid letter	1
(b)	<assignment statement> ::= <variable> = <variable><operator><variable>; <variable> ::= <letter>|<letter><letter>|<letter><letter><letter> <letter> ::= a|b|c|d <operator> :: =+|-|*|÷	2 2 1 1
(c)	<letter> | <letter><variable> // <letter> | <variable><letter>	2

(d) (i)	debugging is faster / easier
	// can debug incomplete code
	// better diagnostics
		1

(ii)	compiler produces executable version – not readable / no need for source code // difficult to reverse-engineer	1

9608/31/M/J/18
Q5

Question	Answer	Marks
5(a)(i)	c4 is not a signed integer	1
5(a)(ii)	10 is not a valid signed integer // 0 is not a valid digit/signed integer // only one digit allowed	1
5(a)(iii)	wrong assignment operator // should be = not := // 6 is not a valid digit/signed integer	1
5(b)	1 mark per bullet assignment • <variable>=<variable><operator><signed integer> variable • <letter><letter> signed integer • +<digit> | -<digit> operator • ^ | \ <assignment statement> ::= <variable> = <variable><operator><signed integer> <variable> ::= <letter><letter> <signed integer> ::= +<digit> | -<digit> <operator> ::= ^ | \	4
5(c)	1 mark per bullet • <letter>| • <letter><variable> For example: <letter>|<letter><variable> <letter>|<variable><letter>	2