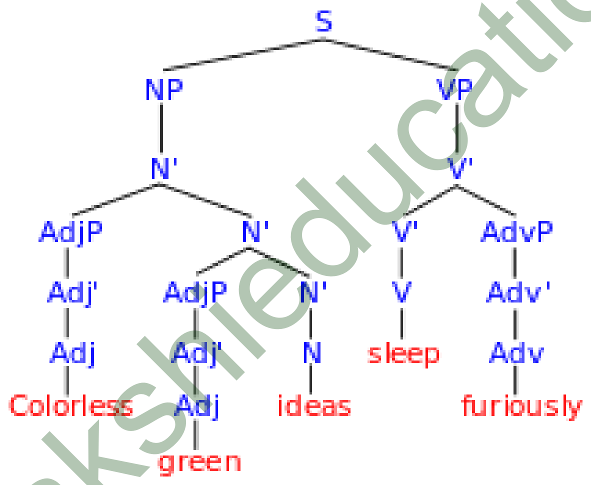
Intermediate forms of source Programs – abstract syntax tree

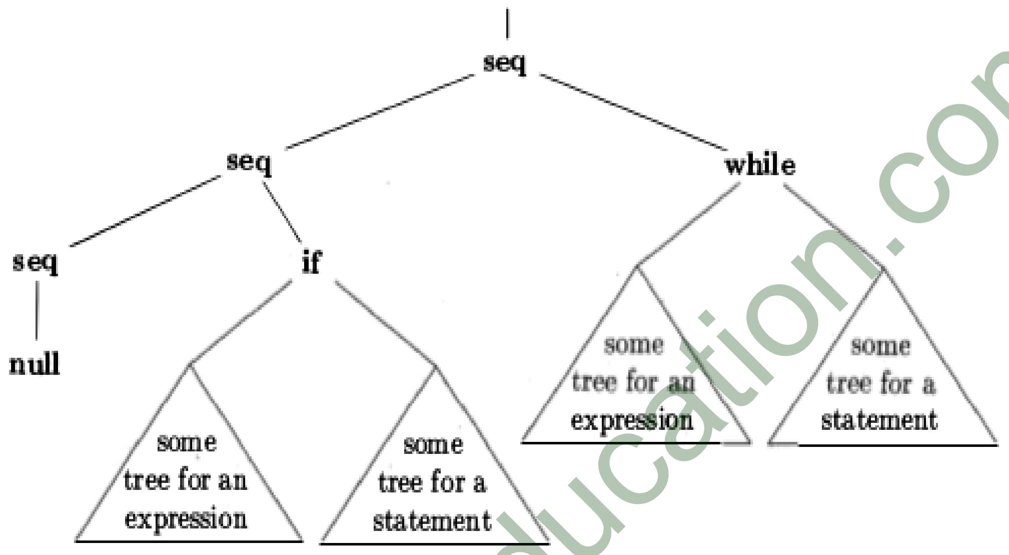
- In computer science, an **abstract syntax tree** (**AST**), or just **syntax tree**, is a tree representation of theabstract syntactic structure of source code written in a programming language.
- Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.
- This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees, which are often built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis. Please refer the below fig.



- ASTs represent the syntactic structure of the some code. The trees of programming constructs such as expressions, flow control statements, etc - grouped into operators (interior nodes) and operands (leaves). For example, the syntax tree for the expression i + 9 would have the operator + as root, the variable i as the operator's left child, and the number 9 as the right child.
- The difference here is that nonterminals and terminals don't play a role, as ASTs don't deal with grammars and string generation, but programming constructs, and thus they represent relationships between such constructs, and not the ways they are generated by a grammar.
- Note that the operators themselves are programming constructs in a given language, and don't have to be actual computational operators (like + is): for loops would also be treated in this way. For example, you could have a syntax tree such as for [expr, expr, expr,

stmnt] (represented inline), where for is an *operator*, and the elements inside the square brackets are its children (representing C's for syntax) - also composed out of operators etc.

- ASTs are usually generated by compilers in the syntax analysis (parsing) phase as well, and are used later for semantic analysis, intermediate representation, code generation, etc. Here's a graphical representation of an AST:


seq
seq
while
seq
if
null
some tree for an expression
some tree for a statement
some tree for an expression
some tree for a statement

- Parse trees tell us exactly how a string was parsed. Parse trees contain more information than we need ie., we only need the basic shape of the tree, not where every non-terminal is – Non-terminals are necessary for parsing, not for meaning.
- An Abstract Syntax Tree is a simplifed version of a parse tree – basically a parse tree without non-terminals.

Polish Notation and Three Address Code:

Polish Notation (PN):

- **Polish notation** (PN), also known as **normal Polish notation** (NPN), **Łukasiewicz notation**, **Warsaw notation**, **Polish prefix notation** or simply**prefix notation**, is a form of notation for logic, arithmetic, and algebra.
- Its distinguishing feature is that it places operators to the left of their operands. If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity.
- The Polishlogician Jan Łukasiewicz invented this notation in 1924 in order to simplify sentential logic.

- The term Polish notation is sometimes taken (as the opposite of infix notation) to also include Polish postfix notation, or reverse Polish notation (RPN), in which the operator is placed after the operands.
- When Polish notation is used as a syntax for mathematical expressions by programming language interpreters, it is readily parsed into abstract syntax trees and can, in fact, define a one-to-one representation for the same. Because of this, Lisp (see below) and related programming languages define their entire syntax in terms of prefix notation (and others use postfix notation).

Three Address Code (TAC):

- In computer science, **three-address code** (often abbreviated to TAC or 3AC) is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations.
- Each TAC instruction has at most three operands and is typically a combination of assignment and a binary operator. For example, `t1 := t2 + t3`. The name derives from the use of three operands in these statements even though instructions with fewer operands may occur.
- Since three-address code is used as an intermediate language within compilers, the operands will most likely not be concrete memory addresses or processor registers, but rather symbolic addresses that will be translated into actual addresses during register allocation.
- It is also not uncommon that operand names are numbered sequentially since three-address code is typically generated by the compiler. A refinement of three-address code is static single assignment form (SSA).
- In three-address code, this would be broken down into several separate instructions. These instructions translate more easily to assembly language. It is also easier to detect common sub-expressions for shortening the code. In the following example, one calculation is composed of several smaller ones:

Example:

# Calculate one solution to the [[quadratic equation]].

```
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
t1 := b * b
t2 := 4 * a
```

```
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

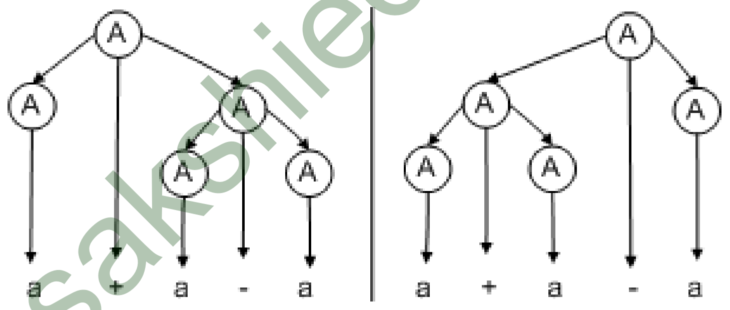
- Three-address code may have conditional and unconditional jumps and methods of accessing memory. It may also have methods of calling functions, or it may reduce these to jumps. In this way, three-address code may be useful in control-flow analysis. In the following C-like example, a loop stores the squares of the numbers between 0 and 9:

```
...
for (i = 0; i < 10; ++i) {
b[i] = i*i;
}
...
t1 := 0              ; initialize i
L1:  if t1 >= 10 goto L2    ; conditional jump
t2 := t1 * t1        ; square of i
t3 := t1 * 4         ; word-align address
t4 := b + t3         ; address to store i*i
*t4 := t2            ; store through pointer
t1 := t1 + 1         ; increase i
goto L1              ; repeat loop
L2:
```

Attribute Grammer:

- An **attribute grammar** is a formal way to define attributes for the productions of a formal grammar, associating these attributes to values. The evaluation occurs in the nodes of the abstract syntax tree, when the language is processed by some parser or compiler.

- The attributes are divided into two groups: synthesized attributes and inherited attributes.
    - The synthesized attributes are the result of the attribute evaluation rules, and may also use the values of the inherited attributes.
    - The inherited attributes are passed down from parent nodes.
    - In some approaches, synthesized attributes are used to pass semantic information up the parse tree, while inherited attributes help pass semantic information down and across it.
    - For instance, when constructing a language translation tool, such as a compiler, it may be used to assign semantic values to syntax constructions.
    - Also, it is possible to validate semantic checks associated with a grammar, representing the rules of a language not explicitly imparted by the syntax definition.
- Attribute grammars can also be used to translate the syntax tree directly into code for some specific machine, or into some intermediate language.
- One strength of attribute grammars is that they can transport information from anywhere in the abstract syntax tree to anywhere else, in a controlled and formal way. Please check the below image.



Syntax Directed Translation:

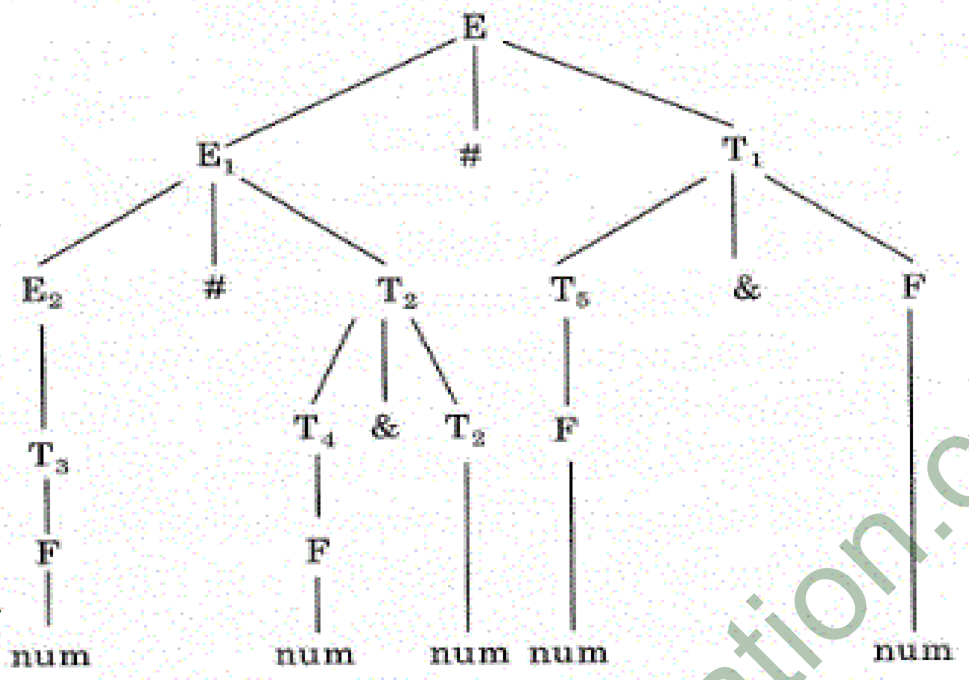
- Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser.
- A common method of **Syntax-directed translation** is translating a string into a sequence of actions by attaching one such action to each rule of a grammar. Thus, parsing a string of the grammar produces a sequence of rule applications. SDT provides a simple way to attach semantics to any such syntax.
- The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.

- By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
- We associate Attributes to the grammar symbols representing the language constructs. Values for attributes are computed by Semantic Rules associated with grammar productions.
- Evaluation of Semantic Rules may:
    - Generate Code
    - Insert information into the Symbol Table
    - Perform Semantic Check
    - Issue error messages etc.
- There are two notations for attaching semantic rules:

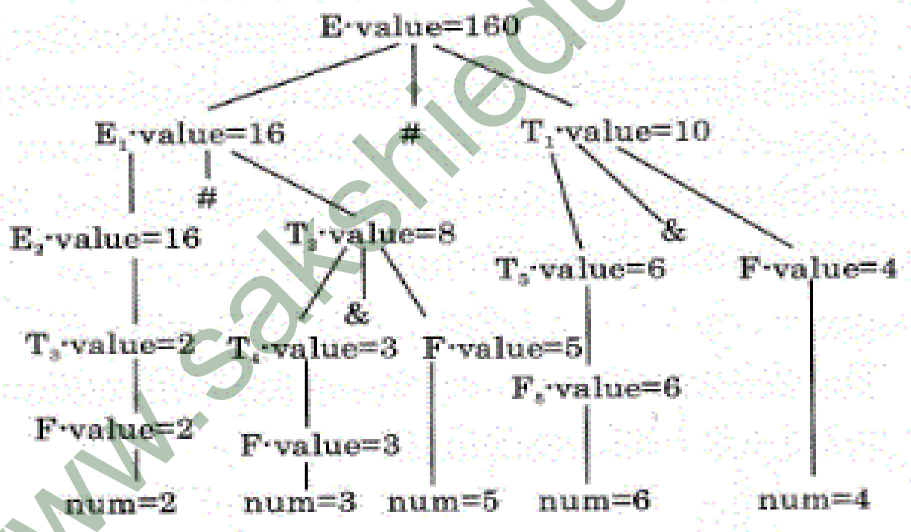
  1. Syntax Directed Definitions. High-level specification hiding many implementation details (also called Attribute Grammars).

  2. Translation Schemes. More implementation oriented: Indicate the order in which semantic rules are to be evaluated.
- The construction of parse tree is as show in the figure.

First we have to construct the parse tree.

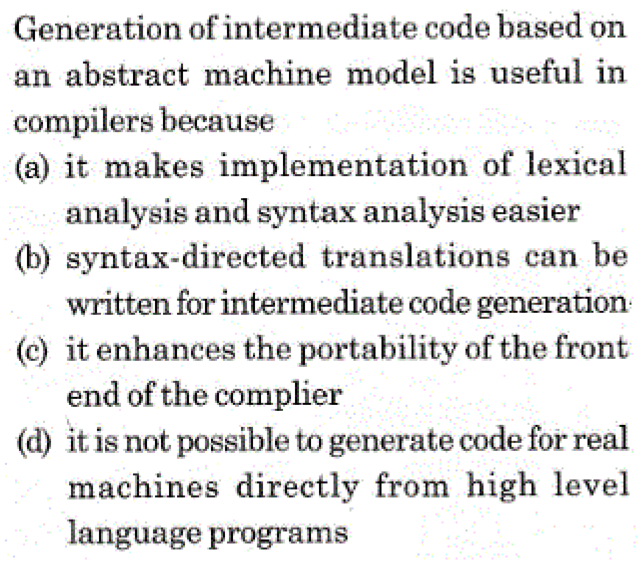


Then we construct the annonated parse tree or parse tree with value at the leaf node.



Previous GATE Questions:

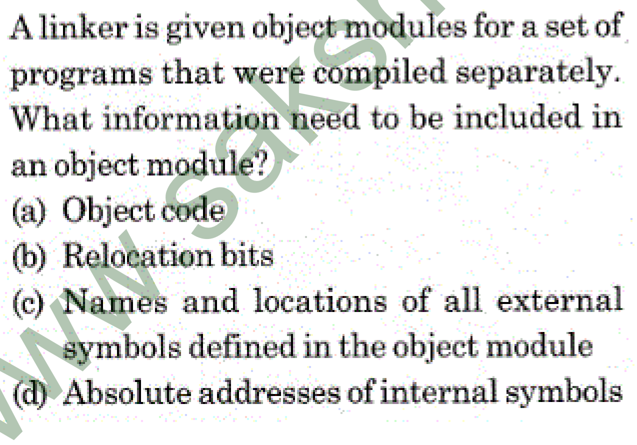
1. **Question**



Generation of intermediate code based on an abstract machine model is useful in compilers because
(a) it makes implementation of lexical analysis and syntax analysis easier
(b) syntax-directed translations can be written for intermediate code generation
(c) it enhances the portability of the front end of the complier
(d) it is not possible to generate code for real machines directly from high level language programs

- ○ (a)
- ○ (b)
- ○ (c)
- ○ (d)

2. **Question**



A linker is given object modules for a set of programs that were compiled separately. What information need to be included in an object module?
(a) Object code
(b) Relocation bits
(c) Names and locations of all external symbols defined in the object module
(d) Absolute addresses of internal symbols

- ○ (a)
- ○ (b)
- ○ (c)
- ○ (d)

3. **Question**

In the following grammar

X : : = X ⊕ Y/Y

Y : : = Z ⊙ Y/Z

Z : : = id

Which of the following is true?

(a) '⊕' is left associative while '⊙' is right associative

(b) Both '⊕' and '⊙' is left associative

(c) '⊕' is the right associative while '⊙' is left associative

(d) None of the above

- ○ (a)
- ○ (b)
- ○ (c)
- ○ (d)

4. Question

In a bottom-up evaluation of a syntax directed definition, inherited attributes can

(a) always be evaluated

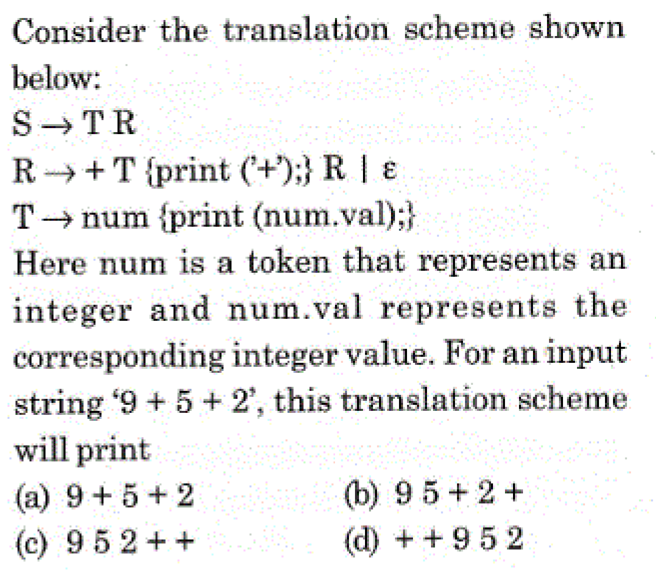
(b) be evaluated only if the definition is L-attributed

(c) be evaluated only if the definition has synthesized attributes

(d) never be evaluated

- ○ (a)
- ○ (b)
- ○ (c)
- ○ (d)

5. Question



Consider the translation scheme shown below:

$S \rightarrow T\ R$

$R \rightarrow +\ T$ {print ('+');} $R \mid \varepsilon$

$T \rightarrow$ num {print (num.val);}

Here num is a token that represents an integer and num.val represents the corresponding integer value. For an input string '9 + 5 + 2', this translation scheme will print

(a) 9 + 5 + 2 (b) 9 5 + 2 +

(c) 9 5 2 + + (d) + + 9 5 2

- (a)
- (b)
- (c)
- (d)

6. **Question**

Consider the syntax directed definition shown below:

$S \rightarrow id := E$ {gen (id.place = E.place;);}

$E \rightarrow E_1 + E_2$ {t = newtemp ( );

gen ($t = E_1$. place + E_2. place;);

E.place = t}

$E \rightarrow id$ {E.place = id.place;}

Here, gen is a function that generates the output code, and newtemp is a function that returns the name of a new temporary variable on every call. Assume that t_i's are the temporary variable names generated by newtemp.

For the statement 'X: = Y + Z', the 3-address code sequence generated by this definition is

(a) $X = Y + Z$

(b) $t_1 = Y + Z; X = t_1$

(c) $t_1 = Y; t_2 = t_1 + Z; X = t_2$

(d) $t_1 = Y; t_2 = Z; t_3 = t_1 + t_2; X = t_3$

- ○ (a)
- ○ (b)
- ○ (c)
- ○ (d)

7. **Question**

Consider the grammar rule $E \rightarrow E_1 - E_2$ for arithmetic expressions. The code generated is targeted to a CPU having a single user register. The subtraction operation requires the first operand to be in the register. If E_1 and E_2 do not have any common subexpression, in order to get the shortest possible code

(a) E_1 should be evaluated first

(b) E_2 should be evaluated first

(c) Evaluation of E_1 and E_2 should necessarily be interleaved

(d) Order of evaluation of E_1 and E_2 is of no consequence

- ○ (a)
- ○ (b)
- ○ (c)
- ○ (d)

8. Question

Consider the grammar with the following translation rules and E as the start symbol.

$E \rightarrow E_1 \# T$	{E.value = E_1.value * T.value}
$\mid$ T	{E.value = T.value}
$T \rightarrow T_1$ & F	{T.value = T_1.value + F.value}
$\mid$ F	{T.value = F.value}
$F \rightarrow$ num	{F.value = num.value}

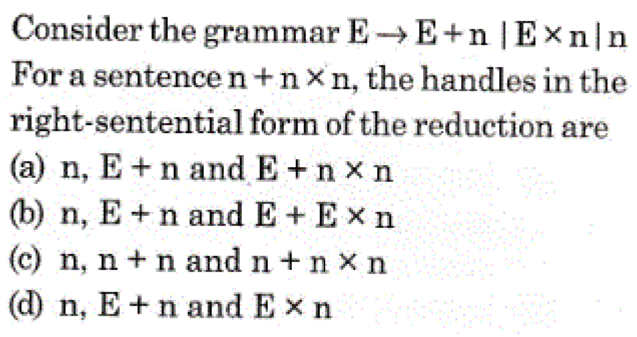
Compute E. value for the root of the parse tree for the expression: 2 # 3 & 5 # 6 & 4.

(a) 200 (b) 180

(c) 160 (d) 40

- ○ (a)
- ○ (b)
- ○ (c)
- ○ (d)

9. Question



Consider the grammar $E \to E + n \mid E \times n \mid n$

For a sentence $n + n \times n$, the handles in the right-sentential form of the reduction are

(a) n, $E + n$ and $E + n \times n$

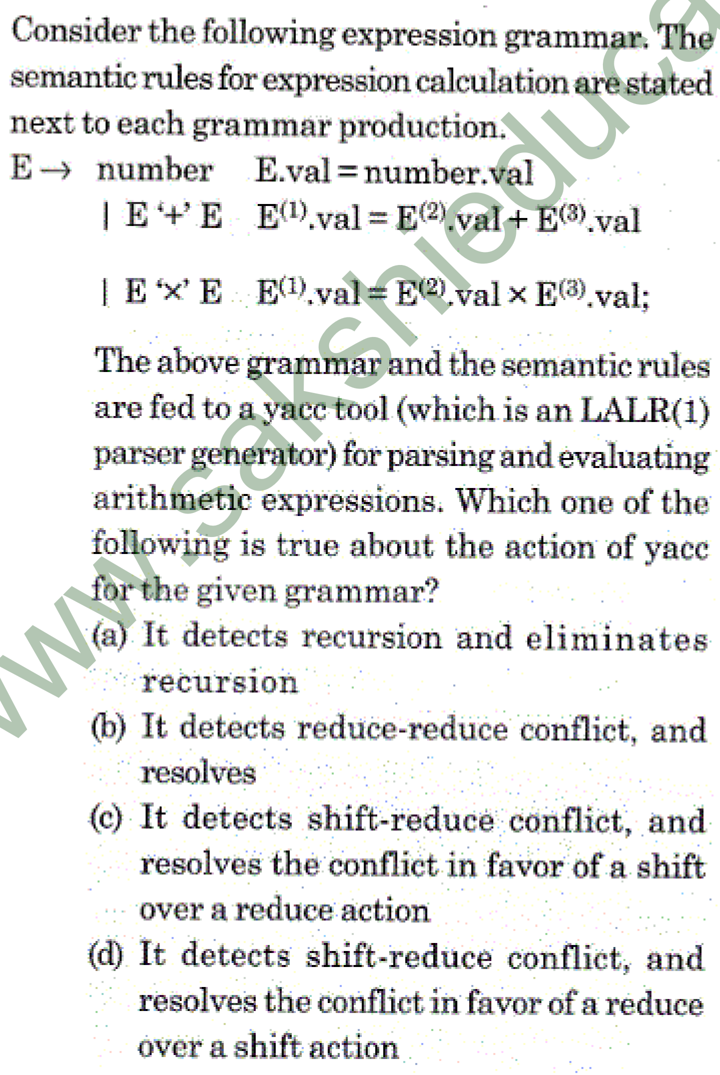
(b) n, $E + n$ and $E + E \times n$

(c) n, $n + n$ and $n + n \times n$

(d) n, $E + n$ and $E \times n$

- ○ (a)
- ○ (b)
- ○ (c)
- ○ (d)

10. Question



Consider the following expression grammar. The semantic rules for expression calculation are stated next to each grammar production.

$E \to$	number	$E.val = number.val$
	$\mid E$ '+' E	$E^{(1)}.val = E^{(2)}.val + E^{(3)}.val$
	$\mid E$ '×' E	$E^{(1)}.val = E^{(2)}.val \times E^{(3)}.val;$

The above grammar and the semantic rules are fed to a yacc tool (which is an LALR(1) parser generator) for parsing and evaluating arithmetic expressions. Which one of the following is true about the action of yacc for the given grammar?

(a) It detects recursion and eliminates recursion

(b) It detects reduce-reduce conflict, and resolves

(c) It detects shift-reduce conflict, and resolves the conflict in favor of a shift over a reduce action

(d) It detects shift-reduce conflict, and resolves the conflict in favor of a reduce over a shift action

- ○ (a)
- ○ (b)
- ○ (c)
- ○ (d)