Question	Answer
4.1.a	Abstract Syntax: Change the initial attribute to point to "S1" instead of "S0". Concrete Graphical Syntax: Move the circle representing "S1" to the top position where the initial state usually resides in graphical state machine representations. Concrete Textual Syntax: Change the line initial SO to initial S1.
4.1.b	Abstract Syntax: Add a new Transition element within the leavingTransitions list of state "S1". This new element would specify the target state as "S0", input as "reset", and output as "initialized". Concrete Graphical Syntax: Add a new arrow leaving state "S1" and pointing to state "S0". Label this arrow with "reset / initialized". Concrete Textual Syntax: Add a new line under the existing transitions leaving state "S1" in the format on input "reset" output "initialized" and go to S0.
4.1.c	Abstract Syntax: Change all occurrences of "S0" to "S2" within the states section. This would modify the state name attribute and references to it in the transitions. Concrete Graphical Syntax: Rename the circle labeled "S0" to "S2" Concrete Textual Syntax: Change all occurrences of "S0" to "S2" throughout the textual representation. This would include the line initial SO and the state name within transition descriptions.
4.2	Lexical Structure Keywords: The fragment includes keywords like message, enum, required, optional, and repeated. These keywords have special meanings within the language and cannot be used as identifiers for other purposes. Identifiers: Identifiers are used to name messages, fields (including phone number types like MOBILE and WORK), and other language constructs. In the fragment, examples of identifiers are Person, PhType, PhoneNo, and name. Identifiers must start with a letter and can contain letters, numbers, and underscores. Literals: The fragment shows string literals enclosed in double quotes ("). For example, "MOBILE" and "WORK" are literals defining the enumeration values for phone types. Comments: The fragment does not include comments, but Google Protocol Buffers supports comments using the // syntax for single-line comments.

	Syntactic Structure Messages: A message definition starts with the message keyword followed by an identifier naming the message type. The message body is enclosed in curly braces {} and contains field declarations. For instance, the fragment defines a message named Person that contains fields for a person's name and phone numbers. Enums: An enum definition starts with the enum keyword, followed by an identifier naming the enumeration type, and then the enumeration body enclosed in curly braces {}. The enumeration body lists the possible values for the enumeration, each with an identifier and an optional integer constant. In the fragment, the PhType enumeration defines phone number types MOBILE and WORK. Fields: A field declaration specifies the name, data type, and other properties of a field within a message. The fragment shows two types of field declarations: required and optional. Required fields must be present in every message of that type. Optional fields may be omitted. A required field declaration starts with the data type, followed by the field name, an equal sign (=), and the field number (a positive integer that uniquely identifies the field within the message). For example: required string name = 1. An optional field declaration is similar to a required field declaration but uses the optional keyword instead of required. Additionally, optional fields can have a default value specified using the default keyword. For example: optional PhType type = 2 [default = MOBILE]. Data Types: The fragment shows two data types: string and PhType (referring to the previously defined enumeration). Protocol Buffers supports various other data types for different purposes. Repetitions: The repeated keyword can be used before the field data type to indicate that a message can have multiple instances of that field. For example: repeated PhoneNo phone = 2.
4.3	[[0 | 1[01]*]]
4.4	4 new rules are needed. 1. expr -> (expr) : This rule allows an expression to be enclosed in parentheses.

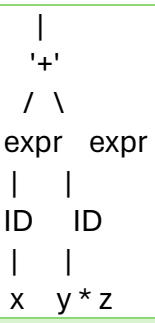
<table>
<tr><td></td><td>2. expr -> term : This rule maintains the ability to have a single term as an expression, even if parentheses are introduced.
3. term -> (expr) : Similar to rule 1, this allows a term to be enclosed in parentheses.
4. term -> factor : This rule preserves the ability to have a single factor as a term, even with parentheses.</td></tr>
<tr><td>4.5</td><td>The word "acxxxac" does NOT belong to the language generated by the given grammar. Here's why:

1. Start Symbol (n): The start symbol n has two possible derivations:
o n -> 1a'c'bbb (This requires the string to start with "1a'c'")
o n -> 3bbb'a'c'b (This requires the string to end with "b'a'c'")
2. String Analysis: Neither of the start symbol's derivations aligns with "acxxxac".
o The string doesn't begin with "1a'c'".
o The string doesn't end with "b'a'c'".
3. Missing "x": The grammar's rules for b always involve "x" characters. The string "acxxxac" lacks these "x" characters within the required "b" productions.

Therefore, there's no derivation path from the start symbol n that can generate the string "acxxxac" using the given grammar.</td></tr>
<tr><td>4.6</td><td>S ::= 0 | 1S</td></tr>
<tr><td>4.7</td><td>Expr ::= Term | Expr <Or> Term
Term ::= Factor | Term <And> Factor
Factor ::= Id | Not Factor
Id ::= <identifier> ; any valid variable identifier
Not ::= <"~" | "NOT"> ; negation operator
And ::= <"&" | "AND"> ; conjunction operator
Or ::= <"|" | "OR"> ; disjunction operator
<identifier> ::= [a-zA-Z_][a-zA-Z0-9_]* ; defines a valid identifier pattern</td></tr>
<tr><td>4.8</td><td>[+-]?([[0]|[1-9][0-9]*])</td></tr>
<tr><td>4.9</td><td>
<table>
<tr><td>Category</td><td>Token</td><td>Regex</td><td>Terminal Symbol</td></tr>
<tr><td>Keywords</td><td>move, on, return</td><td>\b(move)</td><td>on</td></tr>
<tr><td>Identifiers</td><td>RandomWalk, MovingForward, Avoid, ShutDown</td><td>\b(RandomWalk)</td><td>MovingForward</td></tr>
<tr><td>Operators</td><td>-></td><td>-></td><td>Operator</td></tr>
<tr><td>Literals</td><td>10, 1, -180, 180</td><td>[+-]?(0)</td><td>[1-9][0-9]*), ([+-]?(0)</td></tr>
<tr><td>Punctuation</td><td>{, }, (,)</td><td>[{}()]</td><td>Punctuation</td></tr>
</table>
</td></tr>
<tr><td>4.10</td><td>The combination of the optional operator (?) for outputClause and the Kleene star (*) for transition allows for these two syntactic possibilities:</td></tr>
</table>

	1. An empty transition list implicitly defines the end state. 2. A list of transitions where the last transition has an empty output defines the end state. This flexibility provides alternative ways to express the same logic while building the state machine description.
4.11	To eliminate the unordered composition operator & from the production T → α (β & γ) δ and achieve the same language generation using only standard EBNF operators, you can express all possible orderings of the elements \beta and \gamma using the alternative operator |. Here's the transformation: $T \rightarrow \alpha((\beta\gamma)\|(\gamma\beta))\delta T \rightarrow \alpha((\beta\gamma)\|(\gamma\beta))\delta$ This expression means that T starts with \alpha, followed by either \beta then \gamma or \gamma then \beta, and concludes with \delta. This setup replaces the & operator by explicitly enumerating the possible orders of elements, achieving the same results using standard EBNF constructs.
4.12	def STRING: Rule1[String] = rule { WS.? ~ capture (!("\\" | "\"") (ANY | "\\" ("n" | "t" | "\""))* ~ WS.? }
4.13	css ::= style_sheet+ style_sheet ::= CDO CD | selector block SEMI | declaration SEMI selector ::= element | element descendant element | element CHILD element element ::= IDENT descendant ::= WS+ '>' WS+ CHILD ::= WS+ '>' WS+ block ::= '{' declaration* '}' declaration ::= property COLON value SEMI property ::= IDENT value ::= STRING | IDENT | NUMBER | color | URI | HEXCODE | percentage | dimension SEMI ::= ';' CDO ::= '/*' CD ::= '*/' WS ::= ' ' | '\t' | '\n'
4.14	Model -> 'model' Feature Feature -> 'feature' ID Subfeatures Groups Subfeatures -> Feature | Feature Subfeatures | ε Groups -> Group | Group Groups | ε Group -> OrGroup | XorGroup OrGroup -> 'or' GroupContent

	XorGroup -> 'xor' GroupContent GroupContent -> Feature | Feature GroupContent
4.15	Original Grammar and Examples Grammar: 1. poly → 1 poly sign var '^' num | ε 2. var → 3 'x' | 'y' 3. sign → 2 '+' | '-' 4. num → 4 '0' | '1' | '2' Examples: • +x^1 -y^2 • -x^0 +x^2 Regular Expression for Original Grammar Regular Expression: ^([+-][xy]\^[\d])+ This matches strings where each term starts with a sign (+ or -), followed by x or y, a caret (^), and a digit (0 to 2). Modified Grammar Modification: • var → 'x' | 'y' | '(' poly ')' This allows nested polynomials within terms. Examples of New Polynomials: • +x^1 -(+x^1 -y^2)^2 • +(+x^2)^0 -y^1 Regular Expression for Modified Grammar It's not feasible to define a regular expression for the modified grammar. The inclusion of nested polynomials introduces recursion, which regular expressions cannot handle, as they are not capable of matching nested or recursive patterns.
4.16	1. Start with the start symbol: • expr 2. Apply the rule expr → expr '+' expr to introduce the addition operator: • expr + expr 3. Replace the leftmost expr (from step 2) with ID to introduce x: • ID + expr • After substitution: x + expr 4. Replace the remaining expr (from step 3) with ID to introduce y * z (considered as identifier yz): • x + ID • After substitution: x + y * z Parse Tree: expr

	
4.17	Test Objectives: • Verify that the parser can correctly identify valid CSS styles according to the defined grammar. • Ensure the parser handles different combinations of elements, attributes, and values within the specified scope. • Identify any syntax errors or ambiguities in the grammar. Test Case Selection: • Valid Styles: ○ Single element with single attribute (e.g., p { color: black; }) ○ Multiple elements with single attribute each (e.g., p { color: red; } div { background-color: white; }) ○ Single element with multiple attributes (e.g., p { color: black; background-color: white; }) ○ Empty stylesheet (empty css rule) • Invalid Styles: ○ Missing element selector (e.g., { color: black; }) ○ Missing curly braces (p color: black;) ○ Incorrect attribute syntax (e.g., p {color: black}) ○ Invalid attribute name (e.g., p { unknown: black; }) ○ Invalid color value (e.g., p { color: blue; }) ○ Unclosed element block (e.g., p { color: black) • Edge Cases: ○ Consecutive element selectors (e.g., pp { color: black; }) ○ Empty attribute value (e.g., p { color: ; }) Scope of Testing: This testing focuses on the core functionality of parsing valid and invalid styles based on the grammar's rules. It doesn't encompass advanced features like selectors with nesting, pseudo-classes, or media queries. Stopping Criteria: Testing can stop when a significant number of test cases, covering various scenarios, have been executed without encountering errors. Additionally, if no new types of errors are discovered after a certain number of test cases, testing can be concluded.

	Example Test Cases: Valid: • p { color: red; } • div, span { background-color: white; } • h1 { font-size: 20px; color: black; } • {} (empty stylesheet) Invalid: • { color: black; } (missing element selector) • p color: black; (missing curly braces) • p {color: black} (incorrect attribute syntax) • p { unknown: black; } (invalid attribute name) • p { color: blue; } (invalid color value) • p { color: black (unclosed element block) • pp { color: black; } (consecutive element selector) • p { color: ; } (empty attribute value)
4.19	A lexer generator, such as Flex, operates by reading a specification containing a set of named regular expressions that define tokens. It then generates code that efficiently tokenizes a stream of symbols (characters, typically) into a list of tokens. Here's how the process typically works: 1. Input Specification: The input to a lexer generator is a file containing a series of regular expressions paired with token names. These regular expressions describe patterns that match different token types within the input text. 2. Building the Lexer: • The lexer generator constructs a finite automaton (either deterministic DFA or nondeterministic NFA) from these regular expressions. • This automaton is designed to recognize the longest possible prefix of the input stream that matches any of the specified regular expressions. 3. Code Generation: • The lexer generator then converts this finite automaton into source code (e.g., in C or Java). This source code is capable of executing the state transitions of the automaton as it reads through an input stream. • As it reads the input, it identifies substrings that match the patterns defined by the regular expressions and converts these substrings into tokens, often adding them to a list along with metadata like the token type and its position in the input. 4. Tokens in the Input:

	• The tokens for the lexer generator itself are the textual representations of the regular expressions and any associated actions or token names in the specification.
4.20	grammar ::= { production } production ::= nonterminal "->" expression ";" expression ::= term { "|" term } term ::= factor { factor } factor ::= nonterminal | terminal nonterminal ::= identifier terminal ::= "'" character "'" | '"' character '"'
4.21	grammar ::= { production } production ::= nonterminal "->" expression ";" expression ::= term { "|" term } term ::= factor { factor } factor ::= nonterminal | terminal | optional | iteration | grouping optional ::= "[" expression "]" iteration ::= "{" expression "}" grouping ::= "(" expression ")" nonterminal ::= identifier terminal ::= "'" character "'" | '"' character '"' In essence, this basic EBNF grammar shows the core idea behind an Xtext grammar for EBNF. Xtext grammars typically deal with more complex features like explanations, references between code parts, and defining data types in more detail. But fundamentally, both grammars use similar concepts like production rules, basic building blocks (terminals and non-terminals), and EBNF symbols to represent grammar structure. The key difference is often in the specific way they write things down and the extra features Xtext offers to work with the Eclipse and Java environment. In short, both this EBNF grammar and the Xtext grammar use rules and operators to define how a language is written. This highlights a strong similarity in how they describe the structure of a language.
4.22.a	(10)* This expression describes the language of all strings consisting of **zero or more repetitions** of the pattern "10".
4.22.b	1(0|1)* This expression describes the language of all strings that start with "1" followed by **zero or more occurrences** of either "0" or "1".
4.23.c	((0(1|2)3))+

	This expression describes the language of all strings with a length that is a multiple of 3, consisting of a repeated pattern of "0" followed by either "1" or "2", occurring three times each.
4.23.a	'c0ffee.0730' Does NOT belong – The string starts with 'c', which is not a digit (0-9) and doesn't match the first part of the expression.
4.23.b	'0' Belongs – The string consists of a single '0', which matches the first part of the expression.
4.23.c	'1' Belongs – The string consists of a single '1', which matches the first part of the expression (digits 0-9)
4.23.d	'0830.c0ffee' Belongs: • '0830' matches the second part (['0'-'9']+) as it consists of digits. • '.' matches the literal dot. • 'c0ffee' doesn't match the allowed characters (['0'-'9'a'-'f']) as it contains letters beyond 'f'. However, the expression allows zero occurrences of this part (*), so the string is still valid.
4.23.e	'09ea67.' Does NOT belong: • '09ea67' matches the second part (['0'-'9']+) as it starts with digits. • '.' matches the literal dot. • '67' matches the allowed characters (['0'-'9'a'-'f']). • However, the expression requires the digit/letter sequence to occur zero or more times *after* the dot. Since '09ea' appears before the dot, it violates the pattern.
4.24	(a{2})*b | (a+)*c
4.25	[_a-zA-Z][_a-zA-Z0-9]*
4.26	[a-zA-Z][_a-zA-Z0-9]* | _[a-zA-Z0-9][_a-zA-Z0-9]*
4.27	[1-9][0-9]{0,2}\.[0-9]{1,3} | 0\.[0-9]{1,3}
4.28	0x[0-9A-Fa-f]{1,8}
4.29.a	String starting with zero that matches: 0.123 (single zero allowed before the decimal)
4.29.b	String ending with zero that matches: .789 or 5. (single zero allowed after the decimal, or a single digit)
4.29.c	(?<!0)[0-9]+\.(?!0)[0-9]+ | 0\.[0-9]+ | ^[0-9]+$

4.30	\[[1-9][0-9]*\.\.[1-9][0-9]*\]
4.31	^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$
4.32	This grammar describes sentences about reading or writing actions performed by specific people
4.33	This EBNF grammar describes a simple expression format.
4.34.a	Yes, the string "(a, b, c)" belongs to the language generated by this grammar.
4.34.b	s -> 1 '(' t ')' (start with a list) t -> 2 ID ',' t (first identifier followed by comma) ID -> "a" (identifier "a") t -> 3 ε (empty list after first identifier)
4.34.c	\(([^,)]+(?:,\s*[^,)]+)*)?\)
4.35	For this task, a combination of a context-free grammar and a regular expression is the most suitable approach. Context-Free Grammar: Can define the overall structure of the list with separators. Regular Expression: Can capture the complex pattern of hexadecimal groups within each number.
4.36	a(b(1|ε))*c(b(1|ε))*
4.37	S → ε | L S | R S // S is the start symbol L → (L S) | { L S } | [L S] // L represents left parentheses R →) | } |] // R represents right parentheses
4.38	String = 'a'+ | ('a'+ 'b') (String)? 'c'
4.39	String 1: "abab" (Derivation Length: 3) 1. s → (2) g s (Start with "g" followed by another sequence) 2. s → (4) a b s ("g" replaced with "ab" followed by another sequence) 3. s → (4) a b ε (The final sequence is empty) String 2: "dad" (Derivation Length: 2) 1. s → (3) (s)' s (Start with "(" followed by another sequence and ")") 2. s → (5) d (The sequence inside parentheses is replaced with "d") These derivations demonstrate how the grammar can generate strings of different lengths using different production rules.
4.40	The grammar is left-recursive because the start symbol's first production rule (start → '(' parameterList ')') starts with the same non-terminal (parameterList) as the leftmost symbol in the second production rule of parameterList (parameterList → parameterList ',' parameter). This can lead to infinite loops during parsing. Eliminating Left Recursion:

	parameterList → ε | parameterList ',' parameter // Base case (empty list or list with separator) start → '(' parameterList ')' // Use the modified parameterList definition
4.41.a	Not left-recursive.
4.41.b	Left-recursive.
4.41.c	Left-recursive.
4.42.a	stmtList → ε | stmtList ';' stmt // Base case (empty list or list with separator) stmt → 2 '{' stmtList '}' | 3 'print' | 'skip' // Use the modified stmtList definition
4.42.b	qualified-name → ID | qualified-name '.' ID // Base case (single ID or qualified name with dot)

4.43	formula = clause (AND clause)* // Formula is conjunction of clauses (optional repetitions)
	clause = atom (OR atom)* // Clause is disjunction of atoms (optional repetitions)
	atom = literal | NOT literal // Atom is either a literal or its negation
	literal = identifier // Literal is a variable identifier

4.44	formula = clause (AND clause)* // Formula is conjunction of clauses (optional repetitions)
	clause = atom (OR atom)* // Clause is disjunction of atoms (optional repetitions)
	atom = literal | NOT literal // Atom is either a literal or its negation
	literal = identifier // Literal is a variable identifier

4.45	message = word (SLASH word)* // Message is a sequence of words separated by slashes word = tone+ // Word is one or more tones tone = DASH | TRIPLE_DASH | SPACE // Tone can be a dash, triple dash, or space DASH = "-" // Literal dash character TRIPLE_DASH = "---" // Literal triple dash characters SPACE = " " // Literal space character SLASH = "/" // Literal slash character
4.46	Variants of Finite-State Machine Syntax: 1. Nested Definitions: Allow nesting of state definitions within the simpleFSM definition. 2. Optional State Naming: Make naming of states optional. 3. No let Definitions: Remove the let keyword and directly define states within the FSM definition.

	EBNF Grammar (Abstract): fsm = "simpleFSM" name? "{" states transitions "}" name = identifier states = "states" "{" state ("," state)* "}" state = name? "{" transitions "}" transitions = "transitions" "{" transition ("," transition)* "}" transition = from_state "->" to_state (action)? from_state = state_ref to_state = state_ref action = identifier state_ref = name | integer // Allow referencing states by name or index
4.47	cardinality ::= SINGLE_CHAR | RANGE SINGLE_CHAR ::= "?" | "+" | "*" RANGE ::= "[" INT ".." INT "]" INT ::= digit+ digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
4.51	inputClause → 'on' 'input' ID | 'on' ID Reordering Effect: Yes, reordering the operands of the alternative in this PEG production does affect the language it accepts. Explanation: • Original Production: ○ Matches "on", "input", and an ID in that specific order. ○ This wouldn't allow "on" followed by any ID other than "input". • Reordered Production: ○ Matches "on" followed by either "input" and an ID or just an ID. ○ This allows for more flexibility, accepting any ID after "on". Key Difference between PEGs and CFGs: • PEGs: Deterministic left-to-right processing. Once a match is found, there's no backtracking. • CFGs: Non-deterministic. The parser can explore different alternatives to find a match. Reflection Point:

	In CFGs, reordering alternatives doesn't change the generated language because the parser can try both options to find a match. PEGs, due to their left-to-right nature, only attempt the first matching alternative.
4.52	ID → !(isUppercase) IDSuffix
4.53	We'll use statement coverage as the metric to ensure each production rule in the grammar is exercised by at least one test case. This provides a basic level of confidence that the grammar behaves as expected for different syntactic structures. Positive Test Cases: 1. a: This tests the simplest case with a single identifier as an expression (factor → ID). 2. (a + b): This tests the combination of terms with an operator (expr → term (+ term), term → factor). 3. a * b * c: This tests consecutive multiplications within a term (term → factor (*) factor (*) factor). 4. (a + b) * c: This tests nesting of expressions within parentheses (factor → (expr)). Negative Test Cases: 1. +a: This tests a missing operand before the operator (term → operator term). 2. a+: This tests a missing operand after the operator (term → term operator). 3. (a): This tests an unmatched opening parenthesis (factor → (expr)). 4. a(b): This tests an unmatched closing parenthesis (factor → (expr)). 5. 123: This tests an invalid identifier (only letters allowed) (factor → ID). Explanation: The positive test cases cover each production rule: • Single identifier (factor → ID). • Term with an operator (term → factor operator term). • Consecutive multiplications (term → factor (*) factor (*) factor). • Nested expressions (factor → (expr)). The negative test cases target common syntax errors: • Missing operands around operators. • Unmatched parentheses. • Invalid identifiers (numbers in this case). Statement coverage ensures each production rule is included in at least one test case. While it doesn't guarantee complete functionality, it provides a good starting point for testing the grammar.
4.54	model → "model" name "{" elements "}"

	name → ID elements → ε | element ("," element)* element → "class" class_name "{" attributes references generalizations "}" class_name → ID attributes → ε | "attributes" "{" attribute ("," attribute)* "}" attribute → type name type → "int" | "string" | "boolean" // Extend for more data types references → ε | "references" "{" reference ("," reference)* "}" reference → type name "to" class_name generalizations → ε | "generalizations" "{" class_name ("," class_name)* "}"
4.55	document → element (element)* element → OPEN_TAG name attributes? content? CLOSE_TAG OPEN_TAG → "<" CLOSE_TAG → ">" name → ID attributes → ε | attribute (SPACE attribute)* attribute → name EQUALS value EQUALS → "=" value → STRING_LITERAL | !(OPEN_TAG) any_char+ // Not a starting tag content → ε | any_char+ (element content)* // Text or nested elements any_char → . // Matches any character except newline STRING_LITERAL → '"' .*? '"'
4.56	(ID -> ID)+