|  |  |
| --- | --- |
| 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. 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. |
| 4.5 | 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:    * n -> 1a'c'bbb (This requires the string to start with "1a'c'")    * 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".    * The string doesn't begin with "1a'c'".    * 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. |
| 4.6 | S ::= 0 | 1S |
| 4.7 | 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 |
| 4.8 | [+-]?([[0]|[1-9][0-9]\*]) |
| 4.9 | |  |  |  |  | | --- | --- | --- | --- | | Category | Token | Regex | Terminal Symbol | | Keywords | move, on, return | \b(move) | on | | Identifiers | RandomWalk, MovingForward, Avoid, ShutDown | \b(RandomWalk) | MovingForward | | Operators | -> | -> | Operator | | Literals | 10, 1, -180, 180 | [+-]?(0) | [1-9][0-9]\*), ([+-]?(0) | | Punctuation | {, }, (, ) | [{}()] | Punctuation | |
| 4.10 | The combination of the optional operator (?) for outputClause and the Kleene star (\*) for transition allows for these two syntactic possibilities:   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→𝛼((𝛽𝛾)∣(𝛾𝛽))𝛿*T*→*α*((*βγ*)∣(*γβ*))*δ*  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.Top of FormBottom of Form |
| 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  |  '+'  / \  expr expr  | |  ID ID  | |  x y \* z |
| 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 |  |
| 4.20 |  |
| 4.21 |  |
| 4.22 |  |
| 4.23 |  |
| 4.24 |  |
| 4.25 |  |
| 4.26 |  |
| 4.27 |  |
| 4.28 |  |
| 4.29 |  |
| 4.30 |  |
| 4.31 |  |
| 4.32 |  |
| 4.33 |  |
| 4.34 |  |
| 4.35 |  |
| 4.36 |  |
| 4.37 |  |
| 4.38 |  |
| 4.39 |  |
| 4.40 |  |
| 4.41 |  |