Steps for adopting InterGPS Formal Language for our DSL:

1. Finalize the Vocabulary and Grammar
2. Build the Corresponding Lean Library
3. Implement the DSL-to-Lean Translator

Step 1: Finalize the Vocabulary and Grammar :-

- Expand the Predicate List (Vocabulary)
- Define the Syntax (Grammar and rules)

Expanding the Predicate list in categories :-

1. Core Geometric Objects: (fundamental types)
   - 1.1. `Point (A)`: Defines a point, usually referenced by a letter or name.
   - 1.2. `Line (A B)`: Defines an infinite line passing through two points.
   - 1.3. `Segment (A B)`: Defines the finite line segment between two points. (added)
   - 1.4. `Ray (A B)`: Defines a ray starting at the first point and passing through the second. (added)
   - 1.5. `Angle (A B C)`: Defines an angle using three points, with the middle point as the vertex.
   - 1.6. `Triangle (A B C)`: Defines a triangle using its three vertices.
   - 1.7. `Quadrilateral (A B C D)`: Defines a four-sided polygon using its four vertices in order.
   - 1.8. `Parallelogram (A B C D)`: Defines a parallelogram.
   - 1.9. `Square (A B C D)`: Defines a square.
   - 1.10. `Rectangle (A B C D)`: Defines a rectangle.
   - 1.11. `Rhombus (A B C D)`: Defines a rhombus.
   - 1.12. `Trapezoid (A B C D)`: Defines a trapezoid.
   - 1.13. `Kite (A B C D)`: Defines a kite.
   - 1.14. `Polygon (A B C ...)`: Defines a polygon with an ordered list of vertices.
   - 1.15. `Pentagon (A B C D E)`: Defines a five-sided polygon.
   - 1.16. `Hexagon (A B C D E F)`: Defines a six-sided polygon.
   - 1.17. `Heptagon (A B C D E F G)`: Defines a seven-sided polygon.
   - 1.18. `Octagon (A B C D E F G H)`: Defines an eight-sided polygon.
   - 1.19. `Circle (O r)`: Defines a circle, typically with a center point and a radius.
   - 1.20. `Arc (A B)`: Defines an arc, usually with two endpoints and often a center point.
   - 1.21. `Sector (O A B)`: Defines a sector of a circle (a "slice").
   - 1.22. `Shape ($)`: A generic predicate for an undefined or abstract shape.

2. Properties of Objects: (characteristic of a single object)
   - 2.1. `RightAngle (Angle)`: Asserts that a given angle is a right angle (90 degrees).
   - 2.2. `Right (Triangle)`: Asserts that a given triangle is a right triangle.
   - 2.3. `Isosceles (Triangle)`: Asserts that a given triangle is isosceles.

   2.4. `Equilateral (Triangle)`: Asserts that a given triangle is equilateral (all sides are equal).
   2.5. `Regular (Polygon)`: Asserts that a given polygon is regular (all sides and angles are equal).
   2.6. `Red (Shape)`: Identifies a shape as being colored red (for parsing specific problems).
   2.7. `Blue (Shape)`: Identifies a shape as being colored blue.
   2.8. `Green (Shape)`: Identifies a shape as being colored green.
   2.9. `Shaded (Shape)`: Identifies a shape or region as being shaded.

3. Relations Between Objects:
   3.1. `PointLiesOnLine (P) (Line)`: Asserts that a point lies on a given line.
   3.2. `PointLiesOnCircle (P) (Circle)`: Asserts that a point lies on the circumference of a given circle.
   3.3. `Collinear (A B C)`: Asserts that three or more points lie on the same line. (added)
   3.4. `Between (B) (Segment)`: Asserts that a point lies on a segment between its two endpoints. (added)
   3.5. `Parallel (Line1) (Line2)`: Asserts that two lines are parallel.
   3.6. `Perpendicular (Line1) (Line2)`: Asserts that two lines are perpendicular.
   3.7. `IntersectAt (Line1) (Line2) (Point)`: States that two or more lines intersect at a given point.
   3.8. `BisectsAngle (Line) (Angle)`: Asserts that a line or ray divides an angle into two equal angles.
   3.9. `Congruent (Shape1) (Shape2)`: Asserts that two shapes (e.g., triangles, segments) are congruent.
   3.10. `CongruentAngle (Angle1) (Angle2)`: Asserts that two angles are congruent (have the same measure). (added)
   3.11. `Similar (Shape1) (Shape2)`: Asserts that two shapes (e.g., triangles) are similar.
   3.12. `Tangent (Line) (Circle)`: Asserts that a line is tangent to a circle.
   3.13. `Secant (Line) (Circle)`: Asserts that a line is a secant to a circle (intersects it at two points).
   3.14. `CircumscribedTo (Shape1) (Shape2)`: Asserts that one shape is circumscribed around another.
   3.15. `InscribedIn (Shape1) (Shape2)`: Asserts that one shape is inscribed within another.
   3.16. `IsMidpointOf (Point) (Segment)`: Asserts that a point is the midpoint of a segment.
   3.17. `IsCentroidOf (Point) (Tringle)`: Asserts that a point is the centroid of a triangle.

3.18. `IsIncenterOf (Point) (Tringle)`: Asserts that a point is the incenter of a triangle.
3.19. `IsRadiusOf (Segment) (Circle)`: Asserts that a segment is the radius of a circle.
3.20. `IsDiameterOf (Segment) (Circle)`: Asserts that a segment is the diameter of a circle.
3.21. `IsMidsegmentOf (Segment) (Shape)`: Asserts that a segment is the midsegment of a shape (like a triangle).
3.22. `IsChordOf (Segment) (Circle)`: Asserts that a segment is a chord of a circle.
3.23. `IsSideOf (Segment) (Polygon)`: Asserts that a segment is a side of a polygon.
3.24. `IsHypotenuseOf (Segment) (Triangle)`: Asserts that a segment is the hypotenuse of a right triangle.
3.25. `IsPerpendicularBisectorOf (Line) (Segment)`: Asserts that a line is the perpendicular bisector of a segment.
3.26. `IsAltitudeOf (Segment) (Triangle)`: Asserts that a segment is an altitude of a shape.
3.27. `IsMedianOf (Segment) (Tringle)`: Asserts that a segment is a median of a shape.
3.28. `IsBaseOf (Segment) (Triangle)`: Asserts that a segment is a base of a triangle.
3.29. `IsDiagonalOf (Segment) (Polygon)`: Asserts that a segment is a diagonal of a polygon.
3.30. `IsLegOf (Segment) (Triangle)`: Asserts that a segment is a leg of a right triangle.

4. Measurements and Attributes: (functions that return a numerical value or a property)
    4.1. `AreaOf (Shape)`: Returns the numerical area of a shape.
    4.2. `PerimeterOf (Shape)`: Returns the numerical perimeter of a shape.
    4.3. `RadiusOf (Circle)`: Returns the numerical radius of a circle or arc.
    4.4. `DiameterOf (Circle)`: Returns the numerical diameter of a circle.
    4.5. `CircumferenceOf (Circle)`: Returns the numerical circumference of a circle.
    4.6. `AltitudeOf (Shape)`: Returns the altitude segment of a given shape.
    4.7. `HypotenuseOf (Triangle)`: Returns the hypotenuse segment of a right triangle.
    4.8. `SideOf (Polygon)`: Returns a specific side of a polygon.
    4.9. `WidthOf (Shape)`: Returns the numerical width of a shape.
    4.10. `HeightOf (Shape)`: Returns the numerical height of a shape.
    4.11. `LegOf (Shape)`: Returns a leg segment of a shape.
    4.12. `BaseOf (Shape)`: Returns the base segment of a shape.

4.13. `MedianOf (Shape)`: Returns the median segment of a shape.
4.14. `IntersectionOf (Object1) (Object2)`: Returns the point of intersection of two or more objects.
4.15. `MeasureOf (Property)`: Returns the numerical measure of a property (like an angle or arc).
4.16. `LengthOf (Segment)`: Returns the numerical length of a segment.
4.17. `ScaleFactorOf (Shape1) (Shape2)`: Returns the scale factor between two similar shapes.

5. Numerical and Logical Operators: (connect geometric properties to arithmetic and logic)
   5.1. `SinOf (value)`: Returns the sine of a given angle/value.
   5.2. `CosOf (value)`: Returns the cosine of a given angle/value.
   5.3. `TanOf (value)`: Returns the tangent of a given angle/value.
   5.4. `CotOf (value)`: Returns the cotangent of a given angle/value.
   5.5. `HalfOf (value)`: Returns half of a given value.
   5.6. `SquareOf (value)`: Returns the square of a given value.
   5.7. `SqrtOf (value)`: Returns the square root of a given value.
   5.8. `RatioOf (val1) (val2)`: Returns the ratio of two values.
   5.9. `SumOf (val1) (val2)`: Returns the sum of a list of values.
   5.10. `AverageOf (val1) (val2)`: Returns the average of a list of values.
   5.11. `Add (val1) (val2)`: Represents the addition operation.
   5.12. `Mul (val1) (val2)`: Represents the multiplication operation.
   5.13. `Sub (val1) (val2)`: Represents the subtraction operation.
   5.14. `Div (val1) (val2)`: Represents the division operation.
   5.15. `Pow (base) (exponent)`: Represents raising a value to a power.
   5.16. `Equals (val1) (val2)`: Asserts that two values are equal.
6. Goal Predicates:
   6.1. `Find (measurement)`: Specifies that the goal is to calculate a numerical value.
   6.2. `Prove (proposition)`: Specifies that the goal is to prove a logical proposition or relation. (added)
   6.3. `UseTheorem (TheoremName)`: Specifies that a particular theorem should be used (a meta-command for a solver).

Grammar for the language:

1. **Syntax**: We will use **S-expressions**, where every literal is enclosed in parentheses `()`.
2. **Structure**: The first element inside the parentheses is always the **Predicate**, followed by its **Arguments**. `(Predicate Argument1 Argument2 ...)`
3. **Argument Types**: An argument can be one of three things:
   a. **Symbol**: A name for an object or variable (e.g., `A`, `l1`, `my_triangle`, `x`).
   b. **Number**: A numerical constant (e.g., `90`, `14.5`).

c. **Literal**: A nested S-expression, allowing you to build complex statements.

For example let's represent the statement "The length of segment AB is equal to the sum of the lengths of segments AC and CB."

```
(Equals
        (LengthOf (Segment A B))
        (SumOf
                (LengthOf (Segment A C))
                (LengthOf (Segment C B))
        )
)
```

- The top-level predicate is `Equals`.
- Its first argument is a nested literal: `(LengthOf (Segment A B))`.
- Its second argument is also a nested literal: `(SumOf ...)`. This shows how the grammar allows for composition.

CFG for the Geometry DSL: (context free grammer)

```
<literal> ::= ( <predicate> <arguments> )
<arguments> ::= <argument> <arguments> | ε
<argument> ::= <symbol> | <number> | <literal>
<predicate> ::= 'Point' | 'Line' | 'Segment' | 'Equals' | ... (any of the 97 defined predicates)
<symbol> ::= {a-z | A-Z | _}{a-z | A-Z | 0-9 | _}* (cannot start with a number)
<number> ::= [0-9]+ | [0-9]+.[0-9]+ (An integer or float token)
```

The parse tree for the above example is as follows:
(add the picture of the parse tree for this CFG)

Step 2: Build the Corresponding Lean Library

- The goal of this step is to create the formal Lean code that mirrors the vocabulary of the DSL.
- We define all predicates as either Structure, Relation or Measurement.
- We should leverage **`mathlib`**, Lean's mathematical library, which already has a vast collection of definitions for Euclidean geometry.
- Predicates from “Core Geometric Objects” are defined in Structure.
- Predicates from “Properties of Objects” and “Relations Between Objects” are defined in Relations.
- Predicates from “Measurements and Attributes” are defined in Measurements.

- Predicates from “Numerical and Logical Operators” does not goes in any of these files as our DSL-to-Lean translator will map them directly to Lean's built-in operators like `=`, `+`, and `*`, which are already available from **`mathlib`**.
- “Goal Predicates” are **meta-instructions** for our DSL-to-Lean translator.
- Some predicates from “Properties of Objects” such as `Blue (Shape), Green (Shape)` and `Shaded (Shape)`are also **meta-instructions** for our DSL-to-Lean translator.
- For Example:
    - When the translator sees `(Prove (Congruent ...))` in the DSL, it uses that information to structure the final `theorem` statement in the output `lean` file; it doesn't look for a `def Prove ...` in our library.

Step 3: Implement the DSL-to-Lean Translator

- We write a program that automatically converts your Domain-Specific Language (DSL) into valid Lean code.
- Which takes a script written in your geometry DSL and outputs a perfectly formatted `.lean` theorem file.
- There are 2 parts of this program:
    a. The Parser
    b. The Code Generator

A. The Parser:
    a. The parser's job is to read the DSL text file (written in the CFG format we defined) and understand its structure.
    b. It will convert the raw text into a tree-like data structure that your program can easily work with.
B. The Code Generator
    a. The code generator (or transpiler) walks through the tree structure created by the parser and writes the equivalent Lean code for each part.
    b. It needs a set of rules to map DSL predicates to Lean definitions.
    c. It takes an input file `problem1.dsl` and outputs `Problem1.lean` file.

EDIT:

Parser Design and Implementation

1. **Parser Design Philosophy: AST over CST**
    - The parser is designed to build an **Abstract Syntax Tree (AST)**, which is a more direct and efficient data structure for code generation than a full **Concrete Syntax Tree (CST)** or parse tree.
    - A CST is a literal representation of the source text, analogous to a detailed sentence diagram that includes all syntactic elements.
    - An AST is a more abstract representation that captures the essential semantic structure of the code, akin to a concise outline. This is the standard approach in compiler and interpreter design.
2. **Implementation via a Two-Stage Process**
    - The parser is implemented in two main stages:
        - First, a preliminary parse of the S-expression syntax into a nested list structure.
        - Second, a transformation of this structure into the formal AST.
3. **Stage 1: Syntactic Parsing with `sexpdata`**
    - We use the Python library **`sexpdata`** to handle the initial parsing of our S-expression-based Domain-Specific Language (DSL).
    - This library efficiently converts a DSL string into a nested data structure of Python lists, symbols, and primitive values.
        - For example, the DSL expression

          ```
          (Equals (LengthOf (Segment A B)) 5)
          ```

          is parsed into the list:

      ```
      [Symbol('Equals'), [Symbol('LengthOf'), [Symbol('Segment'),
      Symbol('A'), Symbol('B')]], 5]
      ```

    - This intermediate list serves as a simplified parse tree, providing a direct precursor to the formal AST.
4. **Stage 2: AST Construction**
    - A set of custom Python `dataclasses` are defined to represent the nodes of our formal AST. The primary node classes are:
        - **`SymbolNode`**: Represents identifiers (e.g., predicate and point names).
        - **`NumberNode`**: Represents numerical values.
        - **`PredicateNode`**: Represents a predicate and its associated arguments.
    - A transformer function, **`build_ast`**, recursively traverses the nested list generated by `sexpdata`.

    - This function maps each element—be it a symbol, a number, or a sublist—to its corresponding AST node class, producing the final, complete AST. This AST serves as the direct input for the code generation phase.
5. The attached folder contains the complete code for the parser: AI4MATH

Example output:

```
--- Parsing DSL String ---



   (

       (Triangle A B C)

       (Equals (LengthOf (Segment A B)) 5)

       (Prove (Isosceles (Triangle A B C)))

   )




--- Generated Parse Tree (as nested lists) ---

[[Symbol('Triangle'), Symbol('A'), Symbol('B'), Symbol('C')],

 [Symbol('Equals'),

  [Symbol('LengthOf'), [Symbol('Segment'), Symbol('A'), Symbol('B')]],

  5],

 [Symbol('Prove'),

  [Symbol('Isosceles'),

   [Symbol('Triangle'), Symbol('A'), Symbol('B'), Symbol('C')]]]]
```

```
--- Generated Abstract Syntax Tree (AST) ---

PredicateNode(name=PredicateNode(name=SymbolNode(name='Triangle'),
                                args=[SymbolNode(name='A'),
                                      SymbolNode(name='B'),
                                      SymbolNode(name='C')]),
              args=[PredicateNode(name=SymbolNode(name='Equals'),
                                  args=[PredicateNode(name=SymbolNode(name='LengthOf'),
                                                      args=[PredicateNode(name=SymbolNode(name='Segment'),
                                                                          args=[SymbolNode(name='A'),
                                                                                SymbolNode(name='B')])]),
                                        NumberNode(value=5)]),
                    PredicateNode(name=SymbolNode(name='Prove'),
                                  args=[PredicateNode(name=SymbolNode(name='Isosceles'),
                                                      args=[PredicateNode(name=SymbolNode(name='Triangle'),
                                                                          args=[SymbolNode(name='A'),
                                                                                SymbolNode(name='B'),
                                                                                SymbolNode(name='C')])])])])
```