Grammar for DSL for fractals

Team 16

Lexical Considerations

In our programming language, several aspects govern how we write and understand code. These lexical considerations provide rules and guidelines for writing and interpreting the language's syntax and structure.

- Keywords and Identifiers
    - All keywords are lowercase and case-sensitive.
    - Keywords and identifiers must follow strict naming conventions. For example, "if" is a keyword, while "IF" is a variable name.
    - Reserved keywords include: create, if, for, and, bool, break, def, else, False, in, main, True, displayMoldebrot, as well as other names of the built-in functions which will be implemented.
- Comments
    - Comments start with the "#" symbol and terminate at the end of the line.
    - Comments are essential for documenting code and providing context to other developers.
- Whitespace
    - White space, including spaces, tabs, page and line-breaking characters, and comments, can appear between any lexical tokens.
    - Keywords and identifiers must be separated by white space or a token that is neither a keyword nor an identifier.
- String Literals
    - String literals are enclosed in double quotes (").
    - A character literal is enclosed in single quotes (').
    - String literals may contain printable ASCII characters, except for special characters like double quotes, single quotes, and backslashes. Special characters can be represented using escape sequences like ", ', \, \t, and \n.
- Numbers
- Numeric values are 32-bit signed integers, ranging from -2147483648 to 2147483647.

These foundational rules set the stage for understanding the language's syntax and structure. They ensure consistency and clarity in code writing and interpretation.

Operators

- Arithmetic Operators
    - Perform mathematical calculations such as addition (+), subtraction (-), multiplication (*), division (/), exponentiation (**), and modulo (%).
- Comparison Operators
    - Compare two values and return a Boolean result based on equality (==), inequality (!=), or order (> ,<, >=, <=).
- Assignment Operators
    - Assign values to variables and perform compound assignment operations (+=, -=, *=, /=, %=).
- Membership Operators
    - Check for the presence (in) or absence (not in) of a value within a sequence.
- Logical Operators
    - Perform logical operations on Boolean values, including AND (and), OR (or), and NOT (not).
- Identity Operators
    - Determine if two variables refer to the same object (is) or not (is not).

Special Characters

Various special characters like parentheses (), curly braces {}, square brackets [], comma (,), colon (:), period (.), equal sign (=), underscore (_), plus sign (+) and minus sign (-), asterisk (*), slash (/), percent (%), backslash (), exclamation mark (!), question mark (?), and hashtag sign (#) serve distinct purposes in defining syntax, delimiting code blocks, and indicating specific operations or conditions.

Escape Sequences

Escape sequences like \n (newline) and \t (tab) facilitate formatting and representation of special characters within string literals.

Delimiters

Delimiters like double quotes (") and single quotes (') are used to enclose string literals, while backslashes (\) are used as escape characters within them.

Case Sensitivity

Identifiers, function names, variable names, and string literals are all case-sensitive, ensuring precision and differentiation in code writing. Boolean values may also be case-sensitive, requiring exact specification as True or False.

Error Handling

Syntax errors are detected during code compilation or interpretation, while semantic errors and runtime errors may manifest during program execution. Exception handling allows graceful recovery from runtime errors without program termination.

Lexical Scope

Variables declared in the global scope are accessible throughout the entire program, while local variables are confined within specific functions or control statements. Enclosing scope allows access to variables declared in outer functions within inner functions. Scope is determined by the placement of variables and functions in the source code. Variables declared in an inner block have precedence over variables with the same name in an outer block. This ensures that variables are resolved based on their nearest enclosing scope. If a variable is declared with the same name as a variable in an outer scope, it "shadows" the outer variable. The inner variable takes precedence within its local scope, hiding the outer variable with the same name.

Access Modifiers

In the DSL for fractals, there are no explicit access modifiers like public or private. All variables and functions have either global or local scope based on their declaration.

Data Types

Numeric data types include integers and floats. Boolean data type represents truth values True or False. String data type represents sequences of characters. Collection data types include lists and dictionaries. Custom data types can be defined using classes for modeling complex data structures.

Types (detailed)

In the DSL for fractals, the selection of appropriate data types is critical to effectively represent and manipulate various elements related to fractal generation.

- Numeric Data Types
    - Integers and floats are fundamental for representing numerical values such as coordinates, dimensions, and mathematical calculations involved in fractal generation algorithms.
- Boolean Data Type
    - Booleans are essential for logical comparisons and control flow within fractal generation procedures. They determine conditions for iterative processes and branching logic.
- String Data Type
    - Strings play a crucial role in representing textual information related to fractals, such as file names, descriptions, or user input for customizing fractal parameters.
- Collection Data Types

    - **Lists:** Lists can store sequences of numerical values, such as coordinates or parameters for fractal algorithms. They provide flexibility in managing dynamic sets of data.
    - **Dictionaries:** Dictionaries are useful for mapping parameters to their corresponding values, facilitating organization and retrieval of fractal-related information, such as color palettes or transformation rules.
- Custom Data Types
    - Custom objects can be defined using classes to encapsulate complex data structures or entities specific to fractal representations. For example, a "Fractal" class could contain attributes and methods for rendering and manipulating fractal images.

Scope Rules (detailed):

Scope rules in a programming language define the visibility and accessibility of variables and functions within different parts of a program. In the DSL for fractals, the scope rules are defined as follows:

- Global Scope:
    - Variables and functions declared outside of any function or control statement have global scope.
    - They are accessible from anywhere within the program.
    - Global variables and functions can be accessed and modified from any part of the program.
- Local Scope:
    - Variables declared within a function or control statement have local scope.
    - They are accessible only within the block of code where they are defined.
    - Local variables cannot be accessed from outside their enclosing function or control statement.
- Enclosing Scope:
    - Inner functions have access to variables declared in their outer functions.
    - This allows inner functions to use and modify variables from their enclosing scope.
    - However, variables from the inner function are not accessible in the outer function.
- Lexical Scope:
    - Scope is determined by the placement of variables and functions in the source code.
    - Variables declared in an inner block have precedence over variables with the same name in an outer block.
    - This ensures that variables are resolved based on their nearest enclosing scope.
- Variable Shadowing:

        - If a variable is declared with the same name as a variable in an outer scope, it "shadows" the outer variable.
        - The inner variable takes precedence within its local scope, hiding the outer variable with the same name.
    - Access Modifiers:
        - In the DSL for fractals, there are no explicit access modifiers like public or private.
        - All variables and functions have either global or local scope based on their declaration.

Location (detailed):

Locations within the DSL for fractals play a crucial role in storing and manipulating data effectively, ensuring proper management and utilization of resources. The DSL encompasses various types of locations, including scalar variables and array elements, each serving specific purposes within the context of fractal generation and manipulation.

- Scalar Variables:

Scalar variables in the DSL refer to individual data storage units capable of holding single values. These variables are confined within specific scopes, such as within a function or block of code, ensuring encapsulation and preventing unintended interference with other parts of the program. For instance, scalar variables can be utilized to store essential data related to fractal generation, such as parameters defining the dimensions of the fractal image or the maximum number of iterations in an algorithm.

- Array Elements:

Array elements within the DSL represent collections of values organized in a structured manner. These elements facilitate the storage and manipulation of multiple data points related to fractals, allowing for efficient processing and analysis. Arrays can be utilized to store various aspects of fractal data, such as pixel values in an image or coordinates defining specific points within the fractal structure. By organizing data into arrays, the DSL enables systematic access and manipulation of fractal-related information, contributing to the overall effectiveness and efficiency of fractal generation algorithms.

- Data Types and Initialization:

Locations within the DSL may accommodate different data types, including numeric, boolean, string, and custom data types, each tailored to specific requirements of fractal manipulation. For instance, numeric data types such as integers and floats can be utilized to represent numerical values associated with fractal parameters, while boolean data types enable logical operations and condition checking within fractal algorithms. Additionally, string data types facilitate the storage of textual information, such as descriptions or labels associated with fractal images.

- Initialization and Default Values:

Upon declaration, each location within the DSL is initialized to a default value, ensuring consistent behavior and facilitating proper handling of uninitialized data. For instance, scalar variables representing fractal parameters may default to specific values, such as zero or empty strings, ensuring predictable behavior during fractal generation. By establishing default values for locations, the DSL promotes clarity and reliability in data management, minimizing potential errors and inconsistencies during fractal manipulation.

Assignment (detailed):

Assignment operations in our DSL for fractals adhere to specific rules and guidelines, ensuring precise manipulation of data within the program. Let's delve into the details:

- Assignment Semantics:
  - Assignment is permitted only for scalar values.
  - For integer and boolean types, value-copy semantics are used. This means that when assigning a value to a variable, the value resulting from the evaluation of the expression is copied into the location indicated by the variable.
- Increment and Decrement Operations:
  - The += assignment operator increments the value stored in the location by the value of the expression. This operation is valid only for both the location and the expression being of type integer.
  - Similarly, the -= assignment operator decrements the value stored in the location by the value of the expression. This operation is also valid only for integer types.
- Type Compatibility:
  - The location and the expression in an assignment must have the same type to ensure compatibility and prevent type errors.
  - For array types, both the location and the expression must refer to a single array element, which is also a scalar value.
- Assignment to Formal Parameters:
  - It is legal to assign to a formal parameter variable within a method body. Such assignments affect only the scope of the method and do not have implications outside of it.
- Scope and Lifetime:
  - Assignments to variables within a method body are confined to the scope of that method. Variables declared within the method have a limited lifetime and are accessible only within the method's execution context.

Assignment example:

```
# Assigning integer values
width = 800
height = 600
```

```
# Assigning boolean values
use_colors = True
show_axes = False

# Assigning string values
fractal_name = "Mandelbrot Set"
author_name = "Team 16"

# Assigning values to list
color_palette = ["blue", "green", "red"]

# Assigning values to dictionary
parameters = {"xmin": -2.0, "xmax": 2.0, "ymin": -1.5, "ymax": 1.5}

# Assigning values to custom objects
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# Creating an instance of Point and assigning values
start_point = Point(0, 0)
end_point = Point(1, 1)
```

Control Statements (detailed):

If Statement

The if statement in our DSL for fractals follows the conventional semantics. First, the expression (<expr>) is evaluated. If the result is True, the true arm of the statement is executed. Otherwise, the else arm, if it exists, is executed. This control structure allows conditional execution of code blocks based on the evaluation of boolean expressions.

For Statement

In the for statement, the <id> serves as the loop index variable. It shadows any variable of the same name declared in an outer scope, if one exists. The expression (<expr>) after it defines the range of possible values for the loop index. The loop body is executed if the current value of the index variable is less than the ending value. After each iteration of the loop body, the index variable is incremented by 1, and the new value is compared to the ending value to determine if another iteration should execute. This control structure facilitates iterative execution of code blocks, such as generating fractals across a specified range or iterating through a list of parameters for fractal rendering.

Control statements play a crucial role in determining the flow of execution within programs written in our DSL for fractals. The if statement enables conditional branching, allowing different code paths to be followed based on the evaluation of boolean expressions. This is particularly useful for implementing decision-making logic in fractal generation algorithms. The for statement provides a mechanism for repeated execution of code blocks, iterating over a range of values defined by the loop index expression. This is essential for tasks such as generating fractals with varying parameters or iterating through data structures to perform calculations or transformations.

By adhering to strict lexical considerations and syntax rules, our DSL ensures readability, maintainability, and predictability in code writing and interpretation. These control statements, along with other language constructs, empower users to express complex fractal algorithms concisely and effectively, unlocking creativity and exploration in fractal design and visualization.