Teoria kompilacji i kompilatory 2024/2025

Raport implementacji języka

MapS

Kinga Kowal

Paweł Knot

Jarosław Klima

Spis treści

Wprowadzenie

Celem projektu było zaprojektowanie i zaimplementowanie języka programowania o nazwie **MapS** oraz stworzenie dla niego interpretera.

Głównym przeznaczeniem języka **MapS** jest tworzenie i generowanie map. Ponadto projekt spełnia następujące wymagania:

- Możliwość definiowana zmiennych
- Zasięgi obowiązywania zmiennych (scope)
- Typy numeryczne, logiczne, ciągi znaków
- Operacje arytmetyczne oraz logiczne na zmiennych
- Złożone typy danych (np. Lista)
- Wbudowane specjalistyczne typy danych (np. Land, River)
- Możliwość definiowania i korzystania z własnych funkcji/procedur
- Instrukcje warunkowe oraz pętle
- Pełne treści komunikaty o błędach

Implementacja

Do implementacji analizatora składniowego wykorzystaliśmy narzędzie **ANTLR**, dzięki któremu mogliśmy wygenerować parser, lexer, listener oraz visitor dla naszej gramatyki.

Gramatyka

Gramatyka dla języka **MapS** została napisana w pliku z rozszerzeniem .g4. (Załącznik: MapS.g4)

Interpreter

Interpreter języka MapS jest dwuprzebiegowy, każdy przebieg ma swoje specyficzne zadanie.

Pierwszy przebieg:

Zadaniem pierwszego przebiegu jest analiza leksykalna, składniowa orz semantyczna. Kod źródłowy jest czytany znak po znaku i dzielony na tokeny. Tokeny są następnie grupowane w struktury zgodne z gramatyką języka. Na tym etapie program sygnalizuje błędy wynikające z nieprawidłowych deklaracji lub z użycia niedozwolonych poleceń.

Drugi przebieg:

Podczas drugiego przebiegu program z użyciem visitora przechodzi przez zbudowane drzewo w pierwszym przebiegu. Dla każdego węzła drzewa program wykonuje odpowiednią akcję. Program w drugim przebiegu wykorzystuje Tablicę Symboli (zał. InterpreterMemory.py) do poprawnego zarzadzania zmiennymi oraz zasięgami (scope).

Tablica Symboli

W celu realizacji tablicy symboli w interpreterze języka programowania, została utworzona klasa InterpreterMemory (zał. InterpreterMemory.py), której zadaniem jest zarządzanie przestrzeniami nazw oraz przechowywanie identyfikatorów zmiennych wraz z ich wartościami.

Kluczowe elementy implementacji:

Struktura przestrzeni nazw (scope):

Tablica symboli obsługuje zagnieżdżone przestrzenie nazw, co umożliwia m.in. obsługę funkcji i bloków. Do zarządzania nimi służą metody:

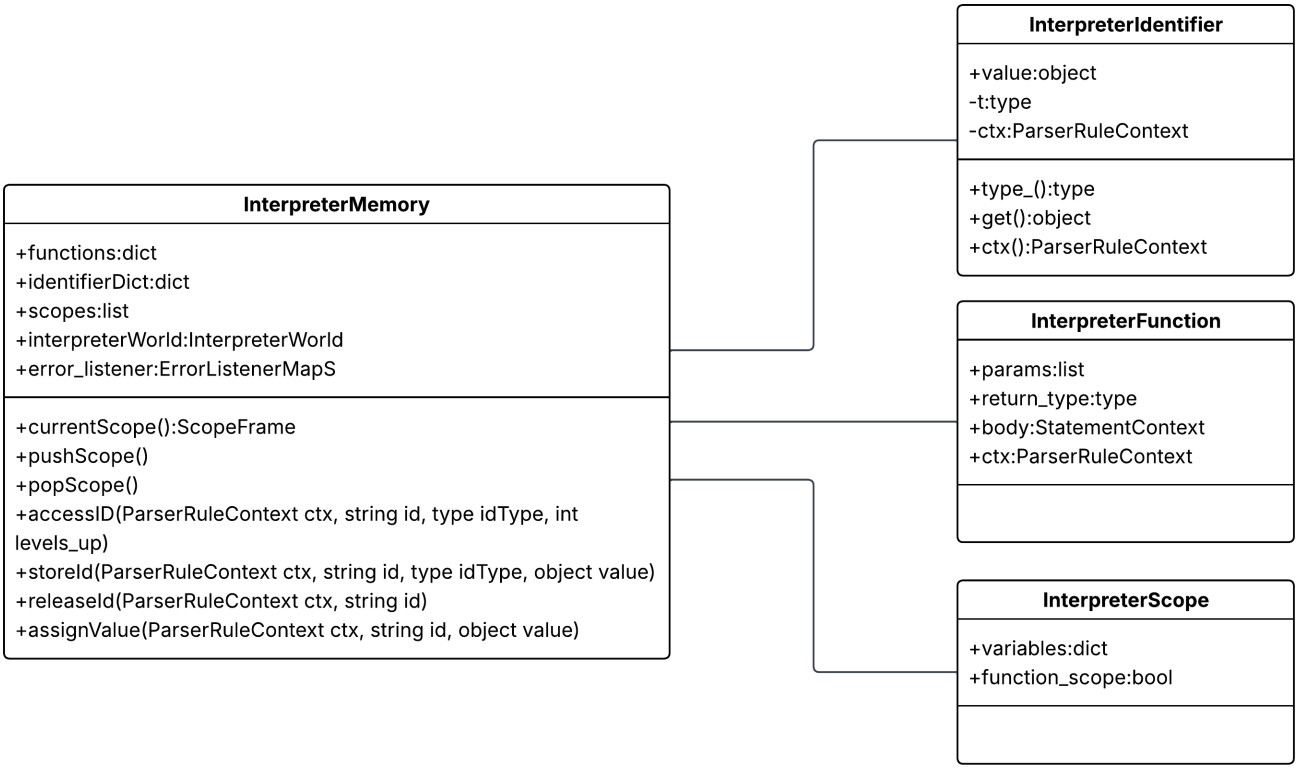
- **pushScope**(is_function_scope=False): tworzy nową przestrzeń nazw; parametr pozwala wyróżnić zakres funkcji.
- **popScope**(): usuwa bieżącą przestrzeń nazw.
- **currentScope**(): zwraca bieżący scope.

Dostęp do zmiennych:

Zarządzanie zmiennych zostało zaimplementowane przy pomocy słownika i specjalnych obiektów InterpreterIdentifier (zał. InterpreterContainers.py)

- **accessId**(ctx, identifier, idType=None, levels_up=0): służy do odczytu wartości zmiennej. Argument levels_up umożliwia przeszukiwanie nadrzędnych scope, a idType może określać typ oczekiwanej zmiennej.
- **storeId**(ctx, identifier, value, idType=None): zapisuje zmienną w bieżącej przestrzeni nazw.
- **releaseId**(ctx, identifier): usuwa zmienną z aktualnego scope.
- **assignValue**(ctx, identifier, value): przypisuje nową wartość zmiennej, przy założeniu, że została już wcześniej zadeklarowana.

Wizualizacja struktury tablicy symboli (UML)



Rekordy aktywacyjne

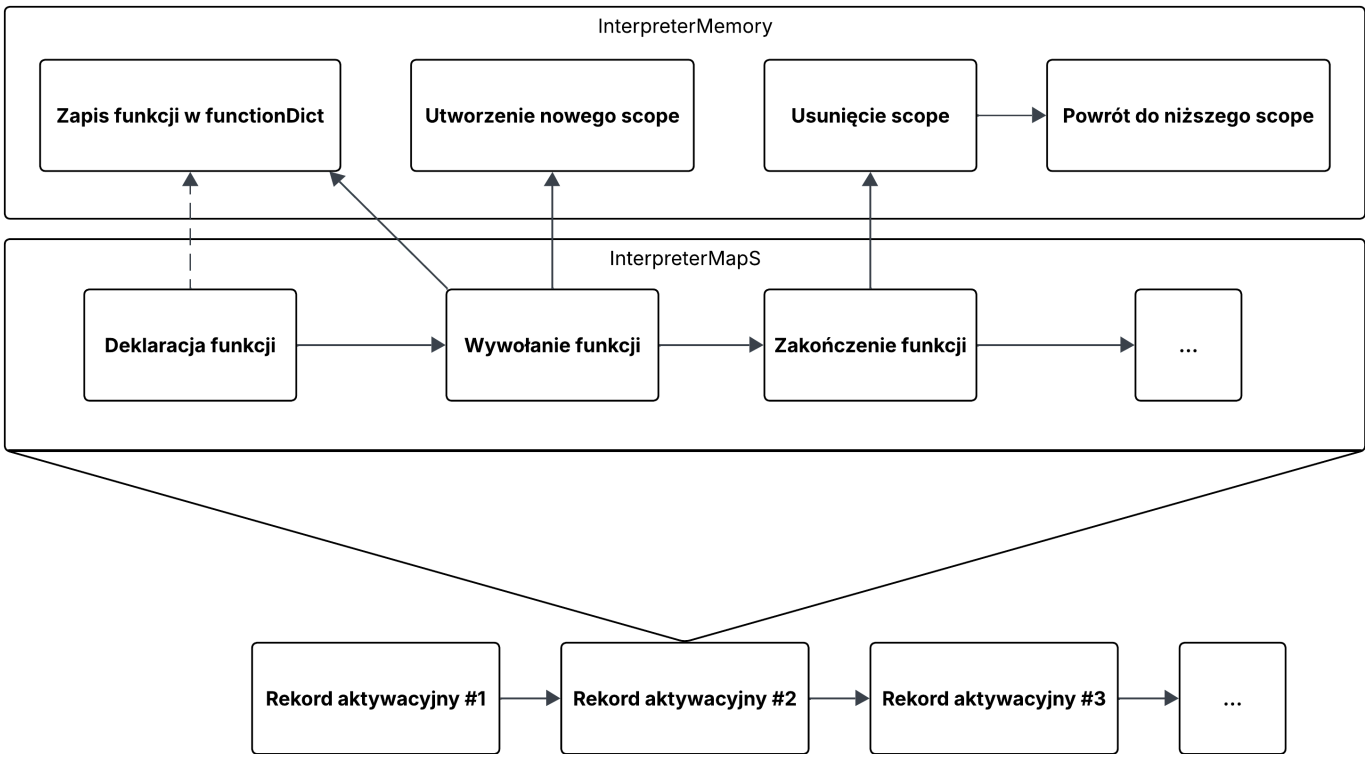
W prezentowanej implementacji, rekordy aktywacyjne są odwzorowane za pomocą struktur zakresów (scope) z użyciem klasy ScopeFrame (zał. InterpreterContainers.py). Rekord aktywacyjny to obszar pamięci przydzielany w trakcie wywoływania funkcji w celu przechowania informacji o parametrach i dostępnych zmiennych z poziomu rekordu. Implementacja struktury rekordów aktywacyjnych przy pomocy stosu, umożliwia wykonanie operacji w poprawnej kolejności.

Procedura wykonania rekordu aktywacyjnego jest następująca:

- utworzenie nowego scope (**pushScope**(is_function_scope=True)),
- wprowadzenie zmiennych odpowiadających parametrom funkcji do nowego zakresu,
- wykonanie ciała funkcji (body zawarty w obiekcie InterpreterFunction),
- usunięcie scope po zakończeniu funkcji (**popScope**()).
- powrót do scope niższego

Wizualizacja procesu wykonywania rekordów aktywacyjnych

Uproszczony schemat obrazujący proces wykonania rekordów aktywacyjnych sekwencyjnie:



Kod źródłowy

Zdecydowaliśmy się na implementację interpretera w języku programowania *Python*, w tym celu utworzyliśmy następujące klasy odpowiedzialne za realizację poszczególnych zadań w procesie interpretowania programów napisanych przez użytkowników:

- InterpreterMapS (Załącznik: InterpreterMapS.py)
- InterpreterMemory (Załącznik: InterpreterMemory.py)
- InterpreterIdentifier (Załącznik: InterpreterContainers.py)
- InterpreterList (Załącznik: InterpreterContainers.py)
- InterpreterLand (Załącznik: InterpreterContainers.py)
- InterpreterLake (Załącznik: InterpreterContainers.py)
- InterpreterRiver (Załącznik: InterpreterContainers.py)
- InterpreterPerimeter (Załącznik: InterpreterContainers.py)
- InterpreterHeight (Załącznik: InterpreterContainers.py)
- InterpreterPoint (Załącznik: InterpreterContainers.py)
- InterpreterFunction (Załącznik: InterpreterContainers.py)
- InterpreterWorld (Załącznik: InterpreterContainers.py)
- World (Załącznik: World.py)
- Land (Załącznik: Land.py)
- Lake (Załącznik: Lake.py)
- River(Załącznik: River.py)
- Perimeter (Załącznik: Perimeter.g4)
- ErrorListenerMapS(Załącznik: ErrorListenerMapS.g4)

Podsumowanie

Język **MapS** został zaimplementowany i po przeprowadzeniu testów wykazał że spełnia założone wymagania projektu.

MapS jest gotowym językiem do użytku publicznego, do raportu dołączamy również dokument "Dokumentacja dla użytkownika końcowego", który obrazuje zakres możliwości zaprojektowanego języka. Treść dokumentu zawiera również fragmenty kodów języka generujące przykładowe mapy.

Załączniki

Maps.g4

```
grammar MapS;

program: progStatement* EOF;

progStatement
    : functionDeclaration
    | statement
    ;

statement
    : variableDeclaration
    | ifStatement
    | blockStatement
    | loopStatement
    | assignment
    | expression ';'
    | returnStatement
    | printStatement
    ;

printStatement
    : 'print(' expression ')' ';'
    ;

returnStatement
    : 'return' expression ';'
    | 'return' ';'
    ;

variableDeclaration
    : primitiveVariableDeclaration
    | listVariableDeclaration
    | pointVariableDeclaration
    | heightVariableDeclaration
    | landVariableDeclaration
    | lakeVariableDeclaration
    | riverVariableDeclaration
    ;

primitiveVariableDeclaration
    : ('int' | 'double' | 'bool' | 'string') IDENTIFIER 'is' expression ';'
    ;

listVariableDeclaration
    : 'List<' type '>' IDENTIFIER 'is' listExpression ';'
    ;
```

```
pointVariableDeclaration
    : 'Point' IDENTIFIER 'is' pointExpression ';'
    ;

heightVariableDeclaration
    : 'Height' IDENTIFIER 'is' heightExpression ';'
    ;

landVariableDeclaration
    : 'Land' IDENTIFIER ('is' pointExpression)? 'with' perimeterDeclaration ','
heightDeclaration ';'
    | 'Land' IDENTIFIER 'is' expression ';'
    ;

perimeterDeclaration
    : 'perimeter is' shape
    ;

shape
    : 'Circle(' expression ')'
    | 'Square(' expression ',' expression ')'
    | 'RandomLand(' expression ',' expression ')'
    | listExpression
    ;

heightDeclaration
    : 'height is' ( functionCall | listExpression)
    ;

lakeVariableDeclaration
    : 'Lake' IDENTIFIER ('is' pointExpression)? 'with' perimeterDeclaration ';'
    ;

riverVariableDeclaration
    : 'River' IDENTIFIER 'is' pointExpression ';'
    ;

functionDeclaration
    : 'function' IDENTIFIER '(' parameters? ')' ':' type '{' statement* '}'
    | 'function' IDENTIFIER '(' parameters? ')' '{' statement* '}'
    ;

parameters
    : type IDENTIFIER (',' type IDENTIFIER)*
    ;

ifStatement
    : 'if' '(' expression ')' 'do' '{' statement* '}' ( 'eif' '(' expression ')'
'do' blockStatement )* ( 'else do' blockStatement )?
    ;

blockStatement
    : '{' statement* '}'
```

```
    ;

loopStatement
    : 'repeat' 'with' IDENTIFIER expression '{' statement* '}'
# RepeatFixedLoop
    | 'repeat' 'with' IDENTIFIER 'from' expression 'to' expression '{' statement*
'}'   # RepeatRangeLoop
    | 'while' '(' expression ')' 'do' '{' statement* '}'
# WhileLoop
    ;

expression
    : '(' expression ')'                  # parenExpr
    | '(' type ')' expression            # castExpr
    |'-' expression                      #unaryMinusExpr
    | expression '^' ('^'|expression)   # powExpr
    | expression '?' ('?'|expression)   # sqrtExpr
    | expression ('*' | '/') expression   # mulDivExpr
    | expression ('+' | '-') expression   # addSubExpr
    | expression ('>' | '<' | '>=' | '<=' | '=' | '!=') expression # compareExpr
    | NOT expression  # notExpr
    | expression AND expression        # andExpr
    | expression OR expression         # orExpr
    | ('sin'|'cos'|'tg'|'ctg')'('expression')' #trygExpr
    | functionCall                          # funcCallExpr
    | ('parent' '::')+ IDENTIFIER                          # scopeAccessExpr
    | INT                                   # intExpr
    | DOUBLE                                # doubleExpr
    | STRING                                # stringExpr
    | BOOLEAN                               # boolExpr
    | IDENTIFIER                            # varExpr
    | pointAccess                           # pointAccessExpr
    | listAccess                            # listAccessExpr
    ;

functionCall
    : IDENTIFIER '(' (expression (',' expression)*)? ')'
    ;

pointAccess
    : IDENTIFIER ('.x' | '.y')
    ;

listAccess
    : IDENTIFIER '[' expression ']'
    ;

pointExpression
    : '(' expression ',' expression ')'
    | IDENTIFIER
    ;

heightExpression
    : '(' pointExpression ',' expression ',' expression ')'
```

```
    |  IDENTIFIER
    ;

listExpression
    : '[' (listElementExpression (',' listElementExpression)*)? ']'
    | IDENTIFIER
    ;

listElementExpression
    : expression | pointExpression | heightExpression
    ;

type
    : 'int' | 'double' | 'bool' | 'string' | 'List<' type '>' | 'Point' | 'Height'
    ;

assignment
    : variableAssignment
    | pointFieldAssignment
    | listAssignment
    ;

variableAssignment
    : IDENTIFIER 'is' expression ';'
    ;

pointFieldAssignment
    : IDENTIFIER ('.x' | '.y') 'is' expression ';'
    ;

listAssignment
    : IDENTIFIER '.add(' expression ')' ';'             #ListAdd
    | IDENTIFIER '[' expression ']' 'is' expression ';' #ListUpdate
    ;

AND : 'and';
OR  : 'or';
NOT : 'not';
INT: [0-9]+;
DOUBLE: [0-9]+('.'[0-9]+)?;
STRING: '"' ~('"')* '"';
BOOLEAN: 'true' | 'false';
IDENTIFIER: [a-zA-Z_][a-zA-Z_0-9]*;
WHITESPACE: [ \t\r\n]+ -> skip;
LINE_COMMENT : '//' ~[\r\n]* -> skip ;
COMMENT: '/*' .*? '*/' -> skip ;
```

InterpreterMapS.py

```
from antlr4 import *
from MapSLexer import MapSLexer
```

```
from MapSParser import MapSParser
from MapSVisitor import MapSVisitor
from InterpreterContainers import *
from InterpreterMemory import *
from World import *
from ErrorListenerMapS import ErrorListenerMapS
import sys
import math

class MapInterpreter(MapSVisitor):
    def __init__(self, errorListener_: ErrorListenerMapS):
        self.memory = InterpreterMemory(errorListener_)
        self.errorListener = errorListener_
        self.in_function = 0


    def visitListVariableDeclaration(self,
ctx:MapSParser.ListVariableDeclarationContext):
        identifier = ctx.IDENTIFIER().getText()
        idType = self.visit(ctx.type_())
        elements = self.visit(ctx.listExpression())
        if type(idType) is tuple:
            self.errorListener.interpreterError(f"Can't create a list of lists",
ctx)
            return
        for element in elements:
            if type(element) is not idType:
                self.errorListener.interpreterError(f"Can't add
{type(element).__name__} to list of {idType.__name__}", ctx)
                return
        result = InterpreterList(idType, elements)
        self.memory.storeId(ctx, identifier, result, InterpreterList)
        return result

    def visitType(self, ctx:MapSParser.TypeContext):
        if ctx.getChildCount() == 3 and ctx.getChild(0).getText() == 'List<':
            inner_type = self.visit(ctx.getChild(1))
            return (list, inner_type)
        else:
            type_name = ctx.getChild(0).getText()
            match type_name:
                case 'int':
                    return int
                case 'double':
                    return float
                case 'bool':
                    return bool
                case 'string':
                    return str
                case 'Point':
                    return InterpreterPoint
                case 'Height':
                    return InterpreterHeight
                case _:
```

```
                    return None

    def visitListExpression(self, ctx:MapSParser.ListExpressionContext):
        result = []
        identifier = ctx.IDENTIFIER()
        if identifier is None:
            ctxList = ctx.listElementExpression()
            for ctxElement in ctxList:
                element = self.visit(ctxElement)
                result.append(element)
            return result
        else:
            return self.memory.accessId(ctx, identifier.getText())



    def visitHeightDeclaration(self, ctx:MapSParser.HeightDeclarationContext):
        funcCall = ctx.functionCall()
        listExpression = ctx.listExpression()
        if funcCall is not None:
            name = funcCall.IDENTIFIER().getText()
            if name not in self.memory.functions:
                self.errorListener.interpreterError(f"Function '{name}' not
defined", funcCall)
                return
            func = self.memory.functions.get(name)
            if func.return_type not in (int, float):
                self.errorListener.interpreterError(f"Height function '{name}'
must return int or double.", funcCall)
                return

            params = func.params
            if len(params) == 2 and params[0][0] in (int, float) and params[1][0]
in (int, float):
                return self.make_height_function(ctx, name, False)
            elif len(params) == 1 and params[0][0] is InterpreterPoint:
                return self.make_height_function(ctx, name, True)
            else:
                self.errorListener.interpreterError(f"Height function '{name}'
must take two numeric parameters or a Point.", funcCall)

        elif listExpression is not None:
            listHeight = self.visit(listExpression)
            if type(listHeight) is not InterpreterList or listHeight.innerType is
not InterpreterHeight:
                self.errorListener.interpreterError(f"Land height has to be a list
of Height", ctx)
                return
            return listHeight
        return None


    def make_height_function(self, ctx, function_name: str, argPoint: bool):
        def height_func(x, y):
```

```
            return self.callFunctionByName(ctx, function_name, [x, y])
        def height_func_point(x, y):
            return self.callFunctionByName(ctx, function_name,
[InterpreterPoint(float(x),float(y))])
        if argPoint:
            return height_func_point
        return height_func

    def visitLandVariableDeclaration(self,
ctx:MapSParser.LandVariableDeclarationContext):
        identifier = ctx.IDENTIFIER().getText()

        land = None
        displacement = None
        perimeter = None
        height = None
        perimeterFunc = None
        heightFunc = None

        pointExpression = ctx.pointExpression()
        if pointExpression is not None:
            displacement = self.visit(pointExpression)
        expression = ctx.expression()
        if expression is None:
            p = self.visit(ctx.perimeterDeclaration())
            h = self.visit(ctx.heightDeclaration())
            if (type(p)==InterpreterList and p.innerType==InterpreterPoint):
                perimeter = p.get()
            else:
                perimeterFunc = p
            if(type(h)==InterpreterList and h.innerType==InterpreterHeight):
                height = h.get()
            else:
                heightFunc = h
            land = InterpreterLand(displacement, perimeter, height, perimeterFunc,
heightFunc)
        else:
            if type(expression) == InterpreterLand:
                land = expression
        self.memory.storeId(ctx, identifier, land, InterpreterLand)
        self.memory.world().addLand(land)
        return land


    def visitShape(self, ctx:MapSParser.ShapeContext):
        #print("visitShape")
        listExpression = ctx.listExpression()
        if listExpression is None:
            expressions = ctx.expression()
            funcArg = self.visit(expressions[0])
            if type(funcArg) is not int:
                self.errorListener.interpreterError("Perimeter function argument
has to be int", ctx)
            funcName = ctx.getChild(0).getText()
```

```
            if "Circle" in funcName:
                circle = InterpreterCircle(funcArg)
                per = Perimeter.from_intcircle(circle)
                intpoints = per.to_intpoints()
                intlist = InterpreterList(InterpreterPoint,intpoints)
                print(type(intpoints))
                return intlist
            elif "Square" in funcName:
                funcArg2 = self.visit(expressions[1])
                if type(funcArg2) is not int:
                    self.errorListener.interpreterError("Perimeter function
argument has to be int", ctx)
                square = InterpreterSquare(funcArg,funcArg2)
                per = Perimeter.from_intsquare(square)
                intpoints = per.to_intpoints()
                intlist = InterpreterList(InterpreterPoint,intpoints)
                return intlist
            elif "RandomLand" in funcName:
                funcArg2 = self.visit(expressions[1])
                if type(funcArg2) not in (int, float):
                    self.errorListener.interpreterError("Second RandomLand
argument has to be int or double", ctx)
                per = Perimeter.from_random_land(funcArg,funcArg2)
                intpoints = per.to_intpoints()
                print(intpoints)
                for x in intpoints:
                    print(f'[{x.x},{x.y}]')
                intlist = InterpreterList(InterpreterPoint,intpoints)
                return intlist
            else:
                return None

    def visitHeightExpression(self, ctx:MapSParser.HeightExpressionContext):
        result = None
        identifier = ctx.IDENTIFIER()
        if identifier is None:
            point = self.visit(ctx.pointExpression())
            z = None
            steep = None
            ctxList = ctx.expression()
            if len(ctxList)==2:
                z = self.visit(ctxList[0])
                steep = self.visit(ctxList[1])
                if type(z) not in (int, float) or type(steep) not in (int, float):
                    self.errorListener.interpreterError(f"Invalid Height
declaration: expected int or float", ctx)
            return InterpreterHeight(point, z, steep)
        else:
            return self.memory.accessId(ctx, identifier.getText(),
InterpreterHeight)


    def visitPointExpression(self, ctx:MapSParser.PointExpressionContext):
        result = None
```

```
        identifier = ctx.IDENTIFIER()
        if identifier is None:
            ctxList = ctx.expression()
            x = self.visit(ctxList[0])
            y = self.visit(ctxList[1])
            if (type(x) is float or type(x) is int) and (type(y) is float or
type(y) is int):
                result = InterpreterPoint(float(x),float(y))
            else:
                self.errorListener.interpreterError(f"Invalid Point coordinates:
expected int or float", ctx)
            return result
        else:
            return self.memory.accessId(ctx, identifier.getText(),
InterpreterPoint)



    def visitPrimitiveVariableDeclaration(self,
ctx:MapSParser.PrimitiveVariableDeclarationContext):
        type_name = ctx.getChild(0).getText()
        match type_name:
            case 'int':
                type_name = int
            case 'double':
                type_name = float
            case 'bool':
                type_name = bool
            case 'string':
                type_name = str
        identifier = ctx.IDENTIFIER().getText()
        exp = self.visit(ctx.expression())
        self.memory.storeId(ctx, identifier, exp, type_name)
        return identifier

    def visitIntExpr(self, ctx:MapSParser.IntExprContext):
        return int(ctx.INT().getText())

    def visitDoubleExpr(self, ctx:MapSParser.DoubleExprContext):
        return float(ctx.DOUBLE().getText())
        return self.visitChildren(ctx)

    def visitStringExpr(self, ctx:MapSParser.StringExprContext):
        value = ctx.STRING().getText()
        return str(value[1:-1])

    def visitBoolExpr(self, ctx:MapSParser.BoolExprContext):
        value = ctx.BOOLEAN().getText()
        if value == 'true':
            return True
        elif value == 'false':
            return False
        return bool(ctx.BOOLEAN().getText())
```

```
    def visitVarExpr(self, ctx:MapSParser.VarExprContext):
        identifier = ctx.IDENTIFIER().getText()
        return self.memory.accessId(ctx, identifier)

    def visitAssignment(self, ctx:MapSParser.AssignmentContext):
        if ctx.variableAssignment() is not None:
            self.visit(ctx.variableAssignment())
        elif ctx.pointFieldAssignment() is not None:
            self.visit(ctx.pointFieldAssignment())
        else:
            self.visit(ctx.listAssignment())

    def visitVariableAssignment(self, ctx:MapSParser.VariableAssignmentContext):
        identifier = ctx.IDENTIFIER().getText()
        self.memory.assignValue(ctx, identifier, self.visit(ctx.expression()))
        return identifier

    def visitProgram(self, ctx:MapSParser.ProgramContext):
        return self.visitChildren(ctx)

    def visitStatement(self, ctx:MapSParser.StatementContext):
        return self.visitChildren(ctx)

    def visitVariableDeclaration(self, ctx:MapSParser.VariableDeclarationContext):
        return self.visitChildren(ctx)

    def visitListElementExpression(self,
ctx:MapSParser.ListElementExpressionContext):
        return self.visitChildren(ctx)

    def visitPointVariableDeclaration(self,
ctx:MapSParser.PointVariableDeclarationContext):
        identifier = ctx.IDENTIFIER().getText()
        self.memory.storeId(ctx, identifier, self.visit(ctx.pointExpression()),
InterpreterPoint)
        return identifier

    def visitPerimeterDeclaration(self,
ctx:MapSParser.PerimeterDeclarationContext):
        return self.visitChildren(ctx)


    def visitHeightVariableDeclaration(self,
ctx:MapSParser.HeightVariableDeclarationContext):
        identifier = ctx.IDENTIFIER().getText()
        self.memory.storeId(ctx, identifier, self.visit(ctx.heightExpression()),
InterpreterHeight)

    def visitLakeVariableDeclaration(self,
ctx:MapSParser.LakeVariableDeclarationContext):
        identifier = ctx.IDENTIFIER().getText()

        lake = None
        displacement = None
```

```
        perimeter = None
        perimeterFunc = None

        pointExpression = ctx.pointExpression()
        if pointExpression is not None:
            displacement = self.visit(pointExpression)

        p = self.visit(ctx.perimeterDeclaration())
        if (type(p)==InterpreterList and p.innerType==InterpreterPoint):
            perimeter = p.get()
        else:
            perimeterFunc = p
        lake = InterpreterLake(displacement, perimeter, perimeterFunc)
        self.memory.storeId(ctx, identifier, lake, InterpreterLake)
        self.memory.world().addLake(lake)
        return lake

    def visitRiverVariableDeclaration(self,
ctx:MapSParser.RiverVariableDeclarationContext):
        identifier = ctx.IDENTIFIER().getText()
        source = None
        pointExpression = ctx.pointExpression()
        if pointExpression is not None:
            source = self.visit(pointExpression)
        river = InterpreterRiver(source)
        self.memory.storeId(ctx, identifier, river, InterpreterRiver)
        self.memory.world().addRiver(river)
        return river

    def visitReturnStatement(self, ctx:MapSParser.ReturnStatementContext):
        if not self.in_function:
             self.errorListener.interpreterError(f"'return' used outside of
function", ctx)
             return

        if ctx.expression() is not None:
            value = self.visit(ctx.expression())
        else:
            value = None
        raise self.ReturnException(value)

    class ReturnException(Exception):
        def __init__(self, value):
            self.value = value

    def visitFunctionDeclaration(self, ctx:MapSParser.FunctionDeclarationContext):
        name = ctx.IDENTIFIER().getText()
        if ctx.parameters() is not None:
            params = self.visit(ctx.parameters())
        else:
            params = []
        if ctx.type_() is not None:
            returnType = self.visit(ctx.type_())
        else:
```

```
            returnType = None
        body = ctx.statement()

        if name  in self.memory.functions:
            line = self.memory.functions.get(name).ctx.start.line
            self.errorListener.interpreterError(f"Function '{name}' already
defined.\n"
                                                +f"Previous definition of {name}
at line {line}.", ctx)
            return

        self.memory.functions[name] = InterpreterFunction(params, returnType,
body, ctx)

    def visitParameters(self, ctx:MapSParser.ParametersContext):
        parameters = []
        names = set()

        types = ctx.type_()
        idents = ctx.IDENTIFIER()

        for param_type_ctx, param_name_ctx in zip(types, idents):
            param_type = self.visit(param_type_ctx)
            param_name = param_name_ctx.getText()

            if param_name in names:
                self.errorListener.interpreterError(
                    f"Param name '{param_name}' can't be used twice in one
function declaration.", ctx)
                return []

            names.add(param_name)
            parameters.append((param_type, param_name))

        return parameters

    def visitFunctionCall(self, ctx:MapSParser.FunctionCallContext):
        func_name = ctx.IDENTIFIER().getText()
        if func_name not in self.memory.functions:
            self.errorListener.interpreterError(f"Function '{func_name}' not
defined", ctx)
            return

        func_ctx = self.memory.functions.get(func_name)

        self.in_function += 1
        params = func_ctx.params
        args = ctx.expression()
        if len(args) != len(params):
            self.errorListener.interpreterError(f"Function '{func_name}' expects
{len(params)} arguments", ctx)
            self.in_function -= 1
            return
```

```
        scopes = len(self.memory.scopes)
        self.memory.pushScope()
        for param, arg in zip(params, args):
            arg_val = self.visit(arg)
            self.memory.storeId(ctx, param[1], arg_val, param[0])

        result = None
        try:
            for stmt_node in func_ctx.body:
                self.visit(stmt_node)
        except self.ReturnException as e:
            result = e.value
        except RecursionError:
                self.errorListener.interpreterError("Recursion limit reached
(possibly infinite recursion)", ctx)
                return None
        finally:
            self.in_function -= 1
            while len(self.memory.scopes) != scopes:
                self.memory.popScope()

        return_type = func_ctx.return_type
        if return_type is None:
            if result is not None:
                self.errorListener.interpreterError("'return' with a value, in
function returning void", ctx)
            return None

        if return_type is not type(result):
            if not(return_type is float and type(result) is int):
                self.errorListener.interpreterError(
                    f"returning {type(result).__name__} from a function with
return type {return_type.__name__}", ctx)
                return None
        return result

    def callFunctionByName(self, ctx, name, args):
        func_ctx = self.memory.functions.get(name)
        params = func_ctx.params
        if len(args) != len(params):
            self.errorListener.interpreterError(f"Function '{name}' expects
{len(params)} arguments", ctx)
            return

        scopes = len(self.memory.scopes)
        self.memory.pushScope()
        self.in_function += 1

        try:
            for (param_type, param_name), arg_val in zip(params, args):
                self.memory.storeId(ctx, param_name, arg_val, param_type)
            result = None
            for stmt_node in func_ctx.body:
                self.visit(stmt_node)
```

```
        except self.ReturnException as e:
            result = e.value
        except RecursionError:
                self.errorListener.interpreterError("Recursion limit reached
(possibly infinite recursion)", ctx)
                return None
        finally:
            self.in_function -= 1
            while len(self.memory.scopes) != scopes:
                self.memory.popScope()

        return_type = func_ctx.return_type
        if return_type is not type(result):
            if not (return_type is float and type(result) is int):
                self.errorListener.interpreterError(
                    f"Returning {type(result).__name__} from function declared to
return {return_type.__name__}", ctx)
                return None
        return result

    def visitIfStatement(self, ctx:MapSParser.IfStatementContext):
        # IF-y
        if self.visit(ctx.expression(0)):
            self.memory.pushScope()
            for stmt_node in ctx.statement():
                self.visit(stmt_node)
            self.memory.popScope()
            return
        # EIF-y
        eif_expression_index = 1
        eif_block_index = 0
        while eif_expression_index < len(ctx.expression()) and \
            eif_block_index < len(ctx.blockStatement()):
            if self.visit(ctx.expression(eif_expression_index)):
                self.visit(ctx.blockStatement(eif_block_index))
                return
            eif_expression_index += 1
            eif_block_index += 1
        # ELSE-y
        if eif_block_index < len(ctx.blockStatement()):
            self.visit(ctx.blockStatement(eif_block_index))
        return None

    def visitBlockStatement(self, ctx:MapSParser.BlockStatementContext):
        self.memory.pushScope()
        for stmt_node in ctx.statement():
            self.visit(stmt_node)
        self.memory.popScope()
        return None

    def visitRepeatFixedLoop(self, ctx:MapSParser.RepeatFixedLoopContext):
        identifier = ctx.IDENTIFIER().getText()
        if identifier is None:
            self.errorListener.interpreterError("Repeat loop requires identifier",
```

```
ctx)
            return

        expression = self.visit(ctx.expression())
        if expression is None or type(expression) is not int:
            self.errorListener.interpreterError(f"Repeat loop requires integer
expression, not {type(expression).__name__}", ctx)
            return

        for i in range(expression):
            self.memory.pushScope()
            self.memory.storeId(ctx, identifier, i, int)
            for stmt_node in ctx.statement():
                self.visit(stmt_node)
            self.memory.popScope()

    def visitRepeatRangeLoop(self, ctx:MapSParser.RepeatRangeLoopContext):
        identifier = ctx.IDENTIFIER().getText()
        if identifier is None:
            self.errorListener.interpreterError("Repeat loop requires identifier",
ctx)
            return

        start = self.visit(ctx.expression(0))
        end = self.visit(ctx.expression(1))
        if start is None or end is None or type(start) is not int or type(end) is
not int:
            self.errorListener.interpreterError(f"Repeat loop requires integer
expression, not {type(start).__name__} and {type(end).__name__}", ctx)
            return

        step = 1
        if start > end:
            step = -1
        elif start == end:
            return

        for i in range(start, end, step):
            self.memory.pushScope()
            self.memory.storeId(ctx, identifier, i, int)
            for stmt_node in ctx.statement():
                self.visit(stmt_node)
            self.memory.popScope()


    def visitWhileLoop(self, ctx:MapSParser.WhileLoopContext):
        condition = self.visit(ctx.expression())
        if condition is None or type(condition) is not bool:
            self.errorListener.interpreterError(f"While loop requires boolean
expression, not {type(condition).__name__}", ctx)
            return

        while condition:
            self.memory.pushScope()
```

```
            for stmt_node in ctx.statement():
                self.visit(stmt_node)
            self.memory.popScope()
            condition = self.visit(ctx.expression())
            if condition is None or type(condition) is not bool:
                self.errorListener.interpreterError(f"While loop requires boolean
expression, not {type(condition).__name__}", ctx)
                return

    #region Expression
    def visitScopeAccessExpr(self, ctx: MapSParser.ScopeAccessExprContext):
        tokens = [child.getText() for child in ctx.children if child.getText() !=
'::']
        if not tokens or tokens[0] != 'parent':
            self.errorListener.interpreterError("Invalid scoped access
expression.", ctx)
            return None

        steps_up = len(tokens) - 1
        var_name = tokens[-1]
        if steps_up >= len(self.memory.scopes):
            self.errorListener.interpreterError(f"No such parent scope ({steps_up}
levels up).", ctx)
            return None
        return self.memory.accessId(ctx, var_name, idType=None,
levels_up=steps_up)

    def visitCastExpr(self, ctx:MapSParser.CastExprContext):
        t = self.visit(ctx.type_())
        try:
            value = t(self.visit(ctx.expression()))
            return value
        except (ValueError, TypeError):
            self.errorListener.interpreterError(f"Cannot cast to {t.__name__} from
{type(self.visit(ctx.expression())).__name__}", ctx)
            return None

    def visitAndExpr(self, ctx:MapSParser.AndExprContext):
        left = self.visit(ctx.expression(0))
        if not isinstance(left, bool):
            self.errorListener.interpreterError(f"Invalid operand for 'and':
{left} (expected boolean)", ctx)
        if isinstance(left, bool) and not left:
            return False
        right = self.visit(ctx.expression(1))
        if not isinstance(right, bool):
            self.errorListener.interpreterError(f"Invalid operand for 'and':
{right} (expected boolean)", ctx)
        return left and right

    def visitOrExpr(self, ctx:MapSParser.OrExprContext):
        left = self.visit(ctx.expression(0))
        if not isinstance(left, bool):
            self.errorListener.interpreterError(f"Invalid operand for 'or': {left}
```

```
(expected boolean)", ctx)
        if isinstance(left, bool) and left:
            return True
        right = self.visit(ctx.expression(1))
        if not isinstance(right, bool):
            self.errorListener.interpreterError(f"Invalid operand for 'or': {left}
(expected boolean)", ctx)
        return left or right

    def visitNotExpr(self, ctx:MapSParser.NotExprContext):
        operand = self.visit(ctx.expression())
        if not isinstance(operand, bool):
            self.errorListener.interpreterError(f"Invalid operand for 'not':
{operand} (expected boolean)", ctx)
        return not operand

    def visitAddSubExpr(self, ctx:MapSParser.AddSubExprContext):
        left = self.visit(ctx.expression(0))
        right = self.visit(ctx.expression(1))

        if not isinstance(left, (int, float)) or not isinstance(right, (int,
float)) or isinstance(left, bool) or isinstance(right, bool):
            self.errorListener.interpreterError(f"Cannot add/subtract types
{type(left).__name__} and {type(right).__name__}", ctx)

        if ctx.getChild(1).getText() == '+':
            return left + right
        else:
            return left - right

    def visitUnaryMinusExpr(self, ctx:MapSParser.UnaryMinusExprContext):
        value = self.visit(ctx.expression())
        if not isinstance(value, (int, float)) or isinstance(value, bool):
            self.errorListener.interpreterError(f"Cannot negate non-number type:
{type(value).__name__}",ctx)
        return -value

    def visitMulDivExpr(self, ctx:MapSParser.MulDivExprContext):
        left = self.visit(ctx.expression(0))
        right = self.visit(ctx.expression(1))

        if not isinstance(left, (int, float)) or not isinstance(right, (int,
float)) or isinstance(left, bool) or isinstance(right, bool):
            self.errorListener.interpreterError(f"Cannot multiply/divide types:
{type(left).__name__} and {type(right).__name__}", ctx)

        if ctx.getChild(1).getText() == '*':
            return left * right
        else:
            if right == 0:
                self.errorListener.interpreterError("Division by zero", ctx)
            else:
                if isinstance(left, int) and isinstance(right, int):
                    return left // right
```

```
                return left / right
        return 0


    def visitSqrtExpr(self, ctx:MapSParser.SqrtExprContext):
        left = self.visit(ctx.expression(0))
        if ctx.expression(1) is None:
            right = 2
        else:
            right = self.visit(ctx.expression(1))

        if not (type(left) in (int, float) and type(right) in (int, float)):
            self.errorListener.interpreterError(f"Sqrt (^) only supports numbers,
not: {type(left).__name__} and {type(right).__name__}", ctx)
            return None

        if right == 0:
            self.errorListener.interpreterError("Root degree cannot be zero", ctx)
            return None

        return math.pow(left, 1 / right)

    def visitParenExpr(self, ctx:MapSParser.ParenExprContext):
        return self.visit(ctx.expression())

    def visitPowExpr(self, ctx:MapSParser.PowExprContext):
        left = self.visit(ctx.expression(0))
        if ctx.expression(1) is None:
            right = 2
        else:
            right = self.visit(ctx.expression(1))

        if not (type(left) in (int, float) and type(right) in (int, float)):
            self.errorListener.interpreterError(f"Pow (^) only supports numbers,
not: {type(left).__name__} and {type(right).__name__}", ctx)
            return None

        if isinstance(left, int) and isinstance(right, int):
                    return int(math.pow(left, right))

        return math.pow(left, right)


    def visitCompareExpr(self, ctx:MapSParser.CompareExprContext):
        left = self.visit(ctx.expression(0))
        right = self.visit(ctx.expression(1))

        if not ((type(left) in (int, float) and type(right) in (int, float)) or
(type(left) is bool and type(right) is bool)):
            self.errorListener.interpreterError(f"Cannot compare types:
{type(left).__name__}, {type(right).__name__}", ctx)

        comp = ctx.getChild(1).getText()
        if comp == '=' or comp == '!=':
```

```
            if not isinstance(left, (int, float)) or not isinstance(right, (int,
float)):
                self.errorListener.interpreterError(f"Cannot compare
{type(left).__name__} and {type(right).__name__}", ctx)
            if comp == '=':
                return left == right
            else:
                return left != right
        else:
            if isinstance(left, bool):
                self.errorListener.interpreterError(f"Cannot compare
{type(left).__name__} and {type(right).__name__}", ctx)
            if comp == '>':
                return left > right
            elif comp == '<':
                return left < right
            elif comp == '>=':
                return left >= right
            elif comp == '<=':
                return left <= right

    def visitTrygExpr(self, ctx):
        func_name = ctx.getChild(0).getText()
        arg = self.visit(ctx.expression())

        if type(arg) not in (int, float):
            self.errorListener.interpreterError(f"Argument of {func_name} must be
numeric.", ctx)
            return None

        if func_name == 'sin':
            return math.sin(arg)
        elif func_name == 'cos':
            return math.cos(arg)
        elif func_name == 'tg':
            return math.tan(arg)
        elif func_name == 'ctg':
            return 1 / math.tan(arg) if arg != 0 else float('inf')

    def visitPointAccessExpr(self, ctx:MapSParser.PointAccessExprContext):
        return self.visitChildren(ctx)

    def visitPointAccess(self, ctx:MapSParser.PointAccessContext):
        identifier = ctx.IDENTIFIER().getText()
        point = self.memory.accessId(ctx, identifier, InterpreterPoint)
        XorY = ctx.getChild(1).getText()
        if XorY == ".x":
            return point.x
        return point.y


    def visitListAccessExpr(self, ctx:MapSParser.ListAccessExprContext):
        return self.visitChildren(ctx)
```

```
    def visitListAccess(self, ctx:MapSParser.ListAccessContext):
        list_name = ctx.IDENTIFIER().getText()
        index = self.visit(ctx.expression())

        if type(index) is not int:
            self.errorListener.interpreterError("List index must be an integer.",
ctx)
            return None

        lst = self.memory.accessId(ctx, list_name, InterpreterList)
        if lst is None:
            return None

        elements = lst.get()
        try:
            return elements[index]
        except IndexError:
            self.errorListener.interpreterError(f"List index {index} out of bounds
for '{list_name}'.", ctx)
            return None

    def visitPointFieldAssignment(self,
ctx:MapSParser.PointFieldAssignmentContext):
        var_name = ctx.IDENTIFIER().getText()
        point = self.memory.accessId(ctx, var_name, InterpreterPoint)
        if point is None:
            return

        field = ctx.getChild(1).getText()  # '.x' or '.y'
        value = self.visit(ctx.expression())

        if type(value) not in (int, float):
            self.errorListener.interpreterError(f"Cannot assign non-numeric value
to {var_name}{field}.", ctx)
            return

        if field == '.x':
            point.x = float(value)
        elif field == '.y':
            point.y = float(value)
        else:
            self.errorListener.interpreterError(f"Unknown point field '{field}'",
ctx)

    def visitListAdd(self, ctx:MapSParser.ListAddContext):
        list_name = ctx.IDENTIFIER().getText()
        item = self.visit(ctx.expression())
        lst = self.memory.accessId(ctx, list_name, InterpreterList)
        if lst is None:
            return None
        if lst.innerType is not type(item):
            if lst.innerType is float and type(item) is int:
                lst.elements.append(float(item))
                return
```

```
            else:
                self.errorListener.interpreterError(f"Cannot add
{type(item).__name__} to list of {lst.innerType.__name__}.", ctx)
                return
        lst.elements.append(item)

    def visitListUpdate(self, ctx:MapSParser.ListUpdateContext):
        list_name = ctx.IDENTIFIER().getText()
        index = self.visit(ctx.expression(0))
        new_value = self.visit(ctx.expression(1))

        if type(index) is not int:
            self.errorListener.interpreterError("List index must be an integer.",
ctx)
            return

        lst = self.memory.accessId(ctx, list_name, InterpreterList)
        if lst is None:
            return

        if lst.innerType is not type(new_value):
            if lst.innerType is float and type(new_value) is int:
                new_value = float(new_value)
            else:
                self.errorListener.interpreterError(f"Cannot add
{type(new_value).__name__} to list of {lst.innerType.__name__}.", ctx)
        try:
            lst.elements[index] = new_value
        except IndexError:
            self.errorListener.interpreterError(f"Index {index} out of range for
list '{list_name}'", ctx)
        return self.visitChildren(ctx)


    def visitPrintStatement(self, ctx:MapSParser.PrintStatementContext):
        value = self.visit(ctx.expression())
        if type(value) is InterpreterList:
            for element in value.elements:
                printValue(element)
        else:
            printValue(value)

def printValue(value):
    if isinstance(value, bool):
        print("true" if value else "false")
    elif type(value) in (str, int, float):
        print(value)
    elif type(value) is InterpreterPoint:
        print(f"({value.x}, {value.y})")
    elif type(value) is InterpreterHeight:
        print(f"(({value.place.x}, {value.place.y}), {value.z}, {value.steep})")
```

```
#region Main
def main():
    filename = sys.argv[1]
    input_stream = FileStream(filename)

    lexer = MapSLexer(input_stream)
    stream = CommonTokenStream(lexer)
    parser = MapSParser(stream)

    error_listener = ErrorListenerMapS()
    lexer.removeErrorListeners()
    parser.removeErrorListeners()
    lexer.addErrorListener(error_listener)
    parser.addErrorListener(error_listener)

    tree = parser.program()

    if error_listener.syntax_errors:
        for err in error_listener.syntax_errors:
            print(f"{err}")
    else:
        interpreter = MapInterpreter(error_listener)
        interpreter.visit(tree)
        if error_listener.interpreter_errors:
            for err in error_listener.interpreter_errors:
                print(f"{err}")
        else:
            draw_image_from_InterpreterWorld(interpreter.memory.world())


if __name__ == "__main__":
    main()
#endregion Main
```

InterpreterMemory.py

```
from InterpreterContainers import *
from ErrorListenerMapS import ErrorListenerMapS
from antlr4 import ParserRuleContext

class InterpreterMemory():
    def __init__(self, errorListener_: ErrorListenerMapS):
        self.functions = {}
        self.identfierDict = {}
        self.scopes = [ScopeFrame({}, function_scope=0)]
        self.intereterWorld = InterpreterWorld()
        self.error_listener = errorListener_

    def currentScope(self):
        return self.scopes[-1]
```

```
    def pushScope(self, is_function_scope=False):
        function_scope = self.currentScope().function_scope
        if is_function_scope:
            self.scopes.append(ScopeFrame({}, function_scope+1))
        else:
            self.scopes.append(ScopeFrame({}, function_scope))

    def popScope(self):
        if len(self.scopes) > 1:
            self.scopes.pop()
        else:
            self.error_listener.interpreterError("This should be impossible.",
None)

    # Aby dostać zmienną podajemy ctx, identifier, opcjonlanie: typ zmiennej
    def accessId(self,ctx: ParserRuleContext, identifier, idType = None,
levels_up: int = 0):
        count = 0
        function_scope = self.currentScope().function_scope
        for frame in reversed(self.scopes):
            scope = frame.variables
            if frame.function_scope == function_scope or frame.function_scope ==
0:
                if identifier in scope:
                    idvalue = scope[identifier]
                    if type(idvalue) == InterpreterIdentifier and ( idType is None
or idvalue.type_() == idType ):
                        if count >= levels_up:
                            return idvalue.get()
                count += 1

        if levels_up == 0:
            self.error_listener.interpreterError(f"No variable named:
{identifier}.", ctx)
        else:
            self.error_listener.interpreterError(f"No variable named
'{identifier}' found {levels_up} scope level(s) up.", ctx)
        return None


    # Aby przechować zmienną podajemy ctx, identifier, wartość, opcjonlanie: typ
zmiennej
    def storeId(self, ctx: ParserRuleContext, identifier, value, idType = None):
        current = self.currentScope().variables
        if identifier in current:
            line = current[identifier].ctx_().start.line
            self.error_listener.interpreterError(f"Variable with name:
{identifier} already defined.\n"
                                                 + f"Previous definition of
{identifier} at line {line}.", ctx)
            return None

        if idType != type(value):
            if type(value) is int and idType is float:
```

```
                value = float(value)
            elif hasattr(value, "innerType") and idType[0]==type([]) and idType[1]
== value.innerType:
                pass
            else:
                self.error_listener.interpreterError(f"Value of type
{type(value).__name__}, cannot be assigned to variable {identifier} of type
{idType.__name__}.", ctx)
                return None

        if idType is None:
            idValue = InterpreterIdentifier(value, type(value), ctx)
        else:
            idValue = InterpreterIdentifier(value, idType, ctx)
        current[identifier] = idValue

    def releaseId(self, ctx: ParserRuleContext, identifier):
        for scope in reversed(self.scopes):
            if identifier in scope.variables:
                scope.variables.pop(identifier)
                return
        self.error_listener.interpreterError(f"No variable named: {identifier}.",
ctx)


    def assignValue(self, ctx: ParserRuleContext, identifier, value):
        function_scope = self.currentScope().function_scope
        for scope in reversed(self.scopes):

            if identifier in scope.variables:
                idObject = scope.variables[identifier]
                if idObject.type_() != type(value):
                    if type(value) is int and idObject.type_() is float:
                        value = float(value)
                    else:
                        self.error_listener.interpreterError(f"Value of type
{type(value)}, cannot be assigned to a variable {identifier} of type
{idObject.type_()}.", ctx)
                        return None
                idObject.value = value
                return
        self.error_listener.interpreterError(f"No variable named: {identifier}.",
ctx)

    def world(self):
        return self.intereterWorld
```

InterpreterContainers.py

```
class InterpreterIdentifier:
    def __init__(self, value, type, ctx):
```

```
        self.value = value
        self.t = type
        self.ctx = ctx


    def type_(self):
        return self.t

    def get(self):
        return self.value

    def ctx_(self):
        return self.ctx

    # def sameType(self, other):
    #     if type(self) != type(other):
    #         return False
    #     return self.type == other.type

class InterpreterList:
    def __init__(self, type, elements):
        self.innerType = type
        self.elements = elements

    def get(self):
        return self.elements

class InterpreterPoint:
    def __init__(self, x: float,y: float):
        self.x = x
        self.y = y

class InterpreterRiver:
    def __init__(self,points:
list[InterpreterPoint],source=None,direction=None,length=100):
        self.points = points
        self.source = source
        self.direction = direction
        self.length = length

class InterpreterHeight:
    def __init__(self, place: InterpreterPoint, z: float, steep: float):
        self.place = place
        self.z = z
        self.steep = steep

class InterpreterLand:
    def __init__(self, displacement: InterpreterPoint=None,  perimeter:
list[InterpreterPoint]=None, height: list[InterpreterHeight]=None,
perimeterFunc=None, heightFunc=None, perimeterShape=None):
        self.displacement = displacement
        self.perimeter = perimeter
        self.height = height
        self.perimeterFunc = perimeterFunc
```

```
        self.heightFunc = heightFunc

class InterpreterLake:
    def __init__(self, displacement: InterpreterPoint=None,  perimeter:
list[InterpreterPoint]=None,perimeterFunc=None,perimeterShape=None):
        self.displacement = displacement
        self.perimeter = perimeter
        self.perimeterFunc = perimeterFunc

class InterpreterSquare:
    def __init__(self, size: int, rotation: int):
        self.size = size
        self.rotation = rotation

class InterpreterCircle:
    def __init__(self,size: int):
        self.size = size

class InterpreterWorld:
    def __init__(self, lands : list[InterpreterLand] = [], size: InterpreterPoint
= InterpreterPoint(500,500),lakes : list[InterpreterLake] = [] ,rivers :
list[InterpreterRiver] = []):
        self.lands = lands
        self.size = size
        self.lakes = lakes
        self.rivers = rivers

    def addLand(self, land: InterpreterLand):
        if land is not None:
            self.lands.append(land)

    def addLake(self, lake: InterpreterLake):
        if(lake is not None):
            self.lakes.append(lake)

    def addRiver(self, river: InterpreterRiver):
        if(river is not None):
            self.rivers.append(river)

class InterpreterFunction:
    def __init__(self, params, return_type, body_ctx, ctx):
        self.params = params
        self.return_type = return_type
        self.body = body_ctx
        self.ctx = ctx

class ScopeFrame:
    def __init__(self, variables: dict, function_scope: int):
        self.variables = variables
        self.function_scope = function_scope
```

Word.py

```
from PIL import ImageDraw,Image
import numpy as np
from Land import *
from Lake import *
from River import *
import random

def draw_image_from_InterpreterWorld(intworld: InterpreterWorld):
    w = World.from_intworld(intworld)
    w.draw()

class World:
    def __init__(self,lands: list[Land],size: list[int],lakes: list[Lake] =
[],rivers: list[River] = []):
        self.lands = lands
        self.lakes = lakes
        self.size = size
        self.rivers = rivers
        self.pixels = np.full((size[0],size[1],3),[0,0,255])
        self.hmap = np.full((size[0],size[1]),np.nan)
        self.all_river_points = []

    @classmethod
    def from_intworld(cls,intworld: InterpreterWorld):
        size = point_to_list(intworld.size)
        lands = [Land.from_intland(x) for x in intworld.lands]
        lakes = [Lake.from_intlake(x) for x in intworld.lakes]
        rivers = [River.from_intriver(x) for x in intworld.rivers]
        return cls(lands,size,lakes,rivers)

    def height_phases_positive(self,n: int) -> list[float]:
        maks=-np.inf
        for land in self.lands:
            M = np.nanmax(land.height_map)
            if M>maks: maks=M
        return [i/n*maks for i in range(n+1)]

    def height_phases_negative(self,n: int) -> list[float]:
        mini = np.inf
        for land in self.lands:
            m = np.nanmin(land.height_map)
            if m<mini: mini=m
        return [i/n*mini for i in range(n+1)]

    def color_phases_positive(self,n: int) -> list[list[int]]:
        a = n//2
        b = n-a
        phases1 = [[255*i//a,255,0] for i in range(a)]
        phases2 = [[255,255-255*i//b,0] for i in range(b+1)]
        return phases1 + phases2
```

```
    def color_phases_negative(self,n: int) -> list[list[int]]:
        return [[0,255-255*i/n//2,0] for i in range(n+1)]


    def give_color(self,land: Land,n: int):
        h_pos = self.height_phases_positive(n)
        h_neg = self.height_phases_negative(2)
        c_pos = self.color_phases_positive(n)
        c_neg = self.color_phases_negative(2)
        print(len(c_neg))
        x_move = land.start[0]
        y_move = land.start[1]
        land_size_x = land.height_map.shape[1]
        land_size_y = land.height_map.shape[0]
        for (row,col),value in np.ndenumerate(land.height_map):
            if not np.isnan(value):
                self.hmap[int(self.size[0]//2-row-y_move+land_size_y//2)]
[int(col+x_move+self.size[1]//2-land_size_x//2)]=value
                if value>=0:
                    for i,y in enumerate(h_pos):
                        if y>=value:
                            self.pixels[int(self.size[0]//2-row-
y_move+land_size_y//2)][int(col+x_move+self.size[1]//2-land_size_x//2)]=c_pos[i]
                            break
                else:
                    for i,y in enumerate(h_neg):
                        if y<=value:
                            self.pixels[int(self.size[0]//2-row-
y_move+land_size_y//2)][int(col+x_move+self.size[1]//2-land_size_x//2)]=c_neg[i]
                            break


    def give_color_to_lake(self,lake: Lake):
        x_move = lake.start[0]
        y_move = lake.start[1]
        land_size_x = lake.height_map.shape[1]
        land_size_y = lake.height_map.shape[0]
        for (row,col),value in np.ndenumerate(lake.height_map):
            if value==0:
                self.pixels[int(self.size[0]//2-row-y_move+land_size_y//2)]
[int(col+x_move+self.size[1]//2-land_size_x//2)] = [0,180,255]
                self.hmap[int(self.size[0]//2-row-y_move+land_size_y//2)]
[int(col+x_move+self.size[1]//2-land_size_x//2)] = 0

    def give_color_to_river(self,river: River):
        river_new = river
        while not
np.isnan(self.hmap[self.size[0]//2+int(river_new.current_point[0]),self.size[1]//2
+int(river_new.current_point[1])]) and river_new.current_point not in
self.all_river_points:
            river_new.river_points.append(river_new.current_point)

self.pixels[self.size[0]//2+int(river_new.current_point[0]),self.size[1]//2+int(ri
```

```
ver_new.current_point[1])]=[0,180,255]
            self.all_river_points.append(river_new.current_point)
            river.river_points.append(river_new.current_point)
            river_new.current_point = self.get_lowest_neighbor(river_new)


    def get_lowest_neighbor(self,river: River):
        min_value = np.inf
        first_value = self.hmap[self.size[0]//2+int(river.current_point[0])]
[self.size[1]//2+int(river.current_point[1])]
        min_neighbor = river.current_point
        descents = []
        neighbor_choices = []
        for neighbor in river.get_neighbors():
            value = self.hmap[self.size[0]//2+int(neighbor[0])]
[self.size[1]//2+int(neighbor[1])]
            if(np.isnan(value)): return neighbor
            if(value<min_value):
                min_value=value
                min_neighbor = neighbor
            descent = first_value-value
            if(descent>=0 and neighbor not in river.river_points):
                descents.append(descent)
                neighbor_choices.append(neighbor)
        suma = sum(descents)
        probabilities = [x / suma for x in descents]
        if(len(neighbor_choices)>0):
            choice = random.choices(neighbor_choices,weights=probabilities,k=1)[0]
            return choice
        else: return min_neighbor

    def draw(self):
        for land in self.lands:
            self.give_color(land,10)
        if self.lakes:
            for lake in self.lakes:
                self.give_color_to_lake(lake)
        if self.rivers:
            for river in self.rivers:
                self.give_color_to_river(river)
        arr = self.pixels.astype(np.uint8)
        print("Kształt tablicy:", arr.shape)
        img_rgb = Image.fromarray(arr, mode='RGB')
        img_rgb.save("obraz_rgb.png")
        img_rgb.show()
```

Land.py

```
import numpy as np
import matplotlib.pyplot as plt
from Perimeter import *
```

```
from scipy.interpolate import griddata
from matplotlib.path import Path
import math

def heights_to_ndarray(heights: list[InterpreterHeight]) -> np.ndarray:
    li = []
    for height in heights:
        l = point_to_list(height.place)
        l.append(height.z)
        li.append(l)
    return np.array(li)

class Land:
    def __init__(self,points3D: np.ndarray,perimeter: Perimeter,start:
list[int],function = None):
        self.start = start
        if(function==None):
            self.height_map =
self.interpolate_heightmap_from_points(points3D,perimeter)
        else:
            self.height_map = self.get_heightmap_from_function(function,perimeter)

    @classmethod
    def from_intland(cls,intland: InterpreterLand):
        start = point_to_list(intland.displacement)
        perimeter = Perimeter.from_intpoint(intland.perimeter)
        if intland.heightFunc is None:
            points3D = heights_to_ndarray(intland.height)
            return cls(points3D,perimeter,start)
        return cls(None,perimeter,start,intland.heightFunc)

    @classmethod
    def from_two_argument_function(cls,function,perimeter: Perimeter,start:
list[int]):
        return cls(np.zeros((3,3)),perimeter,start,function)

    def get_heightmap_from_function(self,function,perimeter: Perimeter):
        x = perimeter.x
        y = perimeter.y
        drawn = np.full((int(max(y)-min(y)),int(max(x)-min(x))),np.nan)
        for (row,col),value in np.ndenumerate(drawn):
            drawn[row,col]=function(row,col)
        row_indices, col_indices = np.indices(drawn.shape)
        x_coords_grid = min(x) + col_indices.ravel()
        y_coords_grid = min(y) + row_indices.ravel()
        indices_flat = np.column_stack((x_coords_grid, y_coords_grid))
        boundary_points = np.column_stack([perimeter.x, perimeter.y])
        boundary_path = Path(boundary_points)
        mask = boundary_path.contains_points(indices_flat).reshape(drawn.shape)
        drawn[~mask] = np.nan
        return drawn

    def interpolate_heightmap_from_points(self,points3D: np.ndarray,perimeter:
Perimeter) -> np.ndarray:
```

```
        x = np.concatenate([points3D[:,0],perimeter.x])
        y = np.concatenate([points3D[:,1],perimeter.y])
        z = np.concatenate([points3D[:,2],np.array([1]*len(perimeter.x))])
        xi = np.linspace(min(x),max(x),int(max(x)-min(x))+1)
        yi = np.linspace(min(y),max(y),int(max(y)-min(y))+1)
        xi,yi = np.meshgrid(xi,yi)
        zi = griddata((x,y),z,(xi,yi),method='cubic')
        boundary_points = np.column_stack([perimeter.x, perimeter.y])
        boundary_path = Path(boundary_points)
        grid_points = np.column_stack([xi.flatten(), yi.flatten()]).reshape(-1, 2)
        mask = boundary_path.contains_points(grid_points).reshape(xi.shape)
        zi[~mask] = np.nan
        return zi

    def __str__(self):
        plt.figure(figsize=(10,6))
        plt.imshow(
            self.height_map,
            origin='lower',
            cmap='viridis',
            aspect='auto'
        )
        plt.colorbar(label = 'Wartosc z')
        plt.show()
        return "Land_shown"
```

Lake.py

```
import numpy as np
import matplotlib.pyplot as plt
from Perimeter import *
from scipy.interpolate import griddata
from matplotlib.path import Path
import math

class Lake:
    def __init__(self,perimeter: Perimeter,start: list[int] = [0,0]):
        self.perimeter = perimeter
        self.start = start
        self.height_map = self.get_heightmap_from_perimeter(perimeter)
    @classmethod
    def from_intlake(cls,intlake: InterpreterLake):
        start = point_to_list(intlake.displacement)
        perimeter = Perimeter.from_intpoint(intlake.perimeter)
        return cls(perimeter,start)

    def get_heightmap_from_perimeter(self,perimeter: Perimeter) -> np.ndarray:
        x = perimeter.x
        y = perimeter.y
        lake_canvas = np.full((int(max(y)-min(y)),int(max(x)-min(x))),np.nan)
        row_indices, col_indices = np.indices(lake_canvas.shape)
```

```
        x_coords_grid = min(x) + col_indices.ravel()
        y_coords_grid = min(y) + row_indices.ravel()
        indices_flat = np.column_stack((x_coords_grid, y_coords_grid))
        boundary_points = np.column_stack([perimeter.x, perimeter.y])
        boundary_path = Path(boundary_points)
        mask =
boundary_path.contains_points(indices_flat).reshape(lake_canvas.shape)
        lake_canvas[mask] = 0
        print(lake_canvas)
        return lake_canvas

    def __str__(self):
        plt.figure(figsize=(10,6))
        plt.imshow(
            self.height_map,
            origin='lower',
            cmap='viridis',
            aspect='auto'
        )
        plt.colorbar(label = 'Wartosc z')
        plt.show()
        return "Land_shown"
```

River.py

```
import numpy as np
import matplotlib.pyplot as plt
from Land import *
from scipy.interpolate import griddata
from matplotlib.path import Path
import math

class River:
    def __init__(self, source: list[int]):
        self.source = source
        self.current_point = source
        self.river_points = []
    @classmethod
    def from_intriver(cls,intriver: InterpreterRiver):
        s = point_to_list(intriver.source)
        return cls(s)

    def get_neighbors(self):
        x = self.current_point[0]
        y = self.current_point[1]
        neighbors = [[x-1,y-1],[x,y-1],[x+1,y-1],[x+1,y],[x+1,y+1],[x,y+1],[x-
1,y+1],[x-1,y]]
        return neighbors
```

Perimeter.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import make_interp_spline
from InterpreterContainers import *
from typing import Callable, List, Union, Any
import random

def point_to_list(point: InterpreterPoint) -> list[int]:
    return [point.x,point.y]

class Perimeter:
    def __init__(self,points: np.ndarray,origin: str="points"):
        if(origin=="points"):
            self.x, self.y = self.interpolate_from_points(points)
            self.points = points
        elif(origin=="square"):
            self.x, self.y = self.interpolate_from_points(points,degree=1)
            self.points =points
        else:
            self.points = points
            self.x = points[:,0]
            self.y = points[:,1]

    @classmethod
    def from_intpoint(cls,intpoints: list[InterpreterPoint]):
        l = [point_to_list(x) for x in intpoints]
        points = np.array(l)
        return cls(points,"points")

    @classmethod
    def from_radial_function(cls,function: Callable[[float],float]):
        theta = np.linspace(0, 2 * np.pi, 100)
        x = function(theta) * np.cos(theta)
        y = function(theta) * np.sin(theta)
        coordinates = np.column_stack((x, y))
        return cls(coordinates,"function")

    @classmethod
    def from_intsquare(cls, square: InterpreterSquare):
        p1 = cls.rotate([-square.size//2,-square.size//2],square.rotation)
        p2 = cls.rotate([-square.size//2,square.size//2],square.rotation)
        p3 = cls.rotate([square.size//2,square.size//2],square.rotation)
        p4 = cls.rotate([square.size//2,-square.size//2],square.rotation)
        points = np.array([p1,p2,p3,p4,p1])
        return cls(points,"square")

    @classmethod
    def from_intcircle(cls, circle: InterpreterCircle):
        theta = np.linspace(0, 2 * np.pi, 100)
        x = circle.size * np.cos(theta)
        y = circle.size * np.sin(theta)
        coordinates = np.column_stack((x, y))
```

```
        return cls(coordinates,"circle")

    def rotate(point,rotation):
        x = point[0]*np.cos(rotation/np.pi*180)-
point[1]*np.sin(rotation/np.pi*180)
        y =
point[0]*np.sin(rotation/np.pi*180)+point[1]*np.cos(rotation/np.pi*180)
        return [x,y]

    @classmethod
    def from_random_land(cls,size,change):
        theta = np.linspace(0, 2 * np.pi, 10)
        x = size * np.cos(theta)
        y = size * np.sin(theta)
        coordinates = np.column_stack((x, y))
        first_and_last = 1
        for i,x in enumerate(coordinates):
            if(i==0):
                first_and_last = random.uniform(1-change, 1+change)
                coordinates[i]=np.array([x[0]*first_and_last,x[1]*first_and_last])
            elif(i==9):
                coordinates[i]=np.array([x[0]*first_and_last,x[1]*first_and_last])
            else:
                rand = random.uniform(1-change, 1+change)
                coordinates[i]=np.array([x[0]*rand,x[1]*rand])
        return cls(coordinates,"randomland")


    def interpolate_from_points(self,points: np.ndarray, degree: int = 2,
number_of_points: int = 200) -> tuple[np.ndarray,np.ndarray]:
        x,y = points.T
        t = np.arange(len(x))
        spline_x = make_interp_spline(t,x, k=degree)
        spline_y = make_interp_spline(t,y, k=degree)
        t_fine = np.linspace(min(t), max(t), number_of_points)
        x_fine = spline_x(t_fine)
        y_fine = spline_y(t_fine)
        return x_fine,y_fine

    def to_intpoints(self) -> List[InterpreterPoint]:
        print(self.x.shape)
        print(self.y.shape)
        intpoints = []
        points = np.column_stack((self.x, self.y))
        for x in points:
            intpoints.append(InterpreterPoint(x[0],x[1]))
        return intpoints

    def __str__(self):
        plt.figure(figsize=(8,8))
        plt.plot(self.x, self.y, 'ro', label='krzywa')
        plt.grid(True)
        plt.show()
        return "Perimeter_shown"
```

ErrorListenerMapS.py

```
from antlr4.error.ErrorListener import ErrorListener
from antlr4 import ParserRuleContext
import sys

class ErrorListenerMapS(ErrorListener):
    def __init__(self):
        super(ErrorListenerMapS, self).__init__()
        self.syntax_errors = []
        self.interpreter_errors = []

    def syntaxError(self, recognizer, offendingSymbol, line, column, msg, e):
        error_message = f"Syntax error at line {line}, column {column}: {msg}"
        self.syntax_errors.append(error_message)

    def interpreterError(self, msg, ctx: ParserRuleContext):
        line = None
        col = None
        if ctx is not None and ctx.start is not None:
            line = ctx.start.line
        if ctx is not None and ctx.start is not None:
            col = ctx.start.column
        error_message = f"Error at line {line}, column: {col}: {msg}"
        print(error_message)
        sys.exit()
        self.interpreter_errors.append(error_message)
```