LANGUAGE REFERENCE MANUAL

MANDALA

October 27, 2015

Edo Roth (enr2116)
Harsha Vemuri (hv2169)
Kanika Verma (kv2253)
Samantha Wiener (srw2168)

Contents

Introduction 2
Lexical Conventions 3
Syntax Notation 6
Types 8
Built-in Functions 11
Accessors 12
Expressions 13
Declarations 17
Scope and Linkage 18
Pre-processing 18
Line Splicing 18
Appendix A: Sample Programs 19
Appendix B: Grammar 22

INTRODUCTION

The Mandala programming language is written to facilitate the process of creating symmetric, geometric designs. The language draws its inspiration from mandalas, circular figures that exhibit both rotational and reflectional symmetry. (Some consider the Mandala to be a spiritual symbol representing the universe.) While this is our central inspiration, Mandala allows people to create their own artistic designs programmatically.

The language Mandala proposes to take advantage of the computational accuracy that would allow users to define geometrically symmetric patterns, while giving users the flexibility to create their own designs and realize them digitally in a simple way.

Lexical Conventions

Tokens

Mandala breaks down into six classes of tokens: identifiers, keywords, constants, strings, operators and other separators. It uses indentations to group blocks of code. Spaces at the end of the line, other tabs, newlines and more generally "white space" are ignored except to separate tokens and at the beginning of the line to determine indentation.

Comments

1. Inline comments are indicated by # and extend to the end of the line.

2. Block (multi-line) comments are indicated by \# and ending with #\.

Identifiers

An identifier is a combination of letters and numbers and underscores. An identifier must begin with a letter or underscore. Naming convention for identifiers is to use lowercase letters to identify variables, and capital letters to identify types.

Keywords

The following identifiers are keywords and may not be redefined for other purposes.

foreach is used to define a loop that allows the user to iterate through a range of numbers.

to is a keyword used in foreach statement to describe the range of the foreach statement.

figure is a keyword used when defining a Shape. figure specifies whether the Shape is a Circle, Triangle or Square. The keyword figure should be indented under the code to create a new Shape.

radius is an attribute of Layer that defines the Layer size, which is circular.

size is an attribute of Shape that describes its scale.

color is an attribute of Shape that will allow users to write a color and specify blue, red, green, yellow, orange, violet, indigo, teal, aqua and or specify the HEX color.

rotation is an attribute of Shape that specifies the degrees of rotation in clockwise direction from zero degrees at the top of the circle.

offset is an attribute of Layer that characterizes the offset of a single layer. By default, the first shape is placed at the top of the layer at 12 o'clock. The offset moves the placement of the first shape clockwise the number of degrees specified.

angularShift is an attribute of Layer that indicates the angle at which shapes are placed in the layer depending on where in the shape they are placed. When angularShift is set to false, the shapes are all placed at the same original angle no matter where in the layer they are. When the angularShift is set to true, the shape is rotated along with its position in the layer, and the shapes are angled radially.

if/else indicates whether or not to execute the following block of code depending on a boolean statement. There does not need to be a corresponding else statement for every if, but in order to use an else statement there must be an if statement preceding it.

true is a boolean that evaluates to 1 and can be used in a conditional statement.

false is a boolean that evaluates to 0 and can be used in a conditional statement.

break ends a loop it is placed in.

continue stops the current iteration of the loop and moves back to the top of the loop.

return allows users to return entities of any defined type from their functions.

def is used to indicate function declaration.

void is used in function declaration to indicate that the function does not return anything.

Punctuation

: colons are placed at the ends of function headers, and begin function definitions. They are also used to define parameters during function calls.

[ ] brackets are used for array literal declaration and array access.

() parentheses are used to define parameters in function declarations and for operator precedence.

"" double quotes are used for string literal declaration.

. periods are used for accessing values of custom types.

{} curly brackets are used in initializing arrays.

Arithmetic operators are defined later in the document.

Constants

Boolean Constants: **true** evaluates to 1 and **false** evaluates to 0.

Character Constants: Mandala supports the following character constants:

Escape Sequence	Definition
`\n`	newline
`\t`	tab
`\'`	single quote
`\"`	double quote
`\\`	backslash
\ hex	hex number

Floating Constants: floating point constants have an integer part, a decimal point, and a fractional part.

String Literals: A string constant is a sequence of characters surrounded by double quotes, for example "test".

Syntax Notation

Program Structure

The user calls various functions to do different actions. To make, fill and draw a mandala, the user must first create a Mandala, then create Layers, which can be filled with Shapes that a user can create. The Layers then must be added to the Mandala using **addTo** and finally the user can draw their Mandala. Some of the main functions such as **create**, **addTo**, and **draw** can be used to do different things based on the types they are called on. For example, create can be used to create different things like a Mandala, or a Layer, or Shape based on what is specified and assign a name to the thing that was created. See the sample programs in Appendix A for examples of this syntax.

Functions

Function Definitions

```
def ReturnType functionName(Type param1, Type param2, ...):
    functionBody
```

Function Calling

```
TypeName varName = functionName: param1, param2, ...
```

Assignment

Assignment of typed variables is with the "=" operator. Correct types must be provided for each variable assignment, i.e.

```
<Type> <var_name> = <value>
```

Assignment of attributes is through adjacency, for example to assign a value to the count attribute in a Layer, a user uses "count 8". Indentation is used to distinguish a hierarchy. In assignments, after a type like a Layer or a Shape is defined, attributes of those things such as size or radius are assigned on indented lines within the section of the overall type.

Arrays

Arrays of all types can be defined by using the bracket operator. They can be initialized with curly brackets listing the members of the array. Arrays may be used for all types, but each array may only contain members of exactly one type.

```
Type[] arrayName = {item1, item2, ..., itemN}
```

Arrays have a built-in size parameter:

```
arrayName.size
```

Statements

Expression Statements

Whitespace after a line has no syntactic meaning in Mandala, so an expression statement ends with a newline character. If an expression needs to span more than one line, the continuation operator can be used at the end of the line.

Conditional Statements

These are **if/else** statements. These are explained in the Lexical Conventions section above.

Loop Statements

```
foreach i = 1 to i = 5:
  Loop contents here
```

Loops over a given range of numbers (1 to 5 in this example). Use indentation to specify the contents of the loop.
Use **break** to end the loop it is placed in.
Use **continue** to stop the current iteration of the loop.

Return Statements

Functions can **return** entities of a defined type.

TYPES

Custom Types

Mandala represents the entire design that will be created by the user. A new Mandala object must be instantiated with the call:

```
 Mandala <name> = create Mandala,
```

where name can be any string. The value of name will then be used for all functionality pertaining to the Mandala object. There are two additional functions that may be used with a Mandala object – the built-in addTo function allows any created layers to be added to the design in the following way:

```
addTo: <Mandala>, <Layer1>, <Layer2>, ... , <LayerN>
```

Note that any layers that are never added to a Mandala will never be drawn - they stand alone in an abstract manner but not pictorially. Finally, any Mandala object can be drawn with the call:

```
draw: <Mandala>
```

This will bring up the display window, and show the complete creation represented by the Mandala object.

Layer represents an abstract circle upon which shapes can be placed. Like Mandala, a Layer is instantiated with the create constructor, but unlike Mandala, it has additional parameters that may be provided to specify additional properties. Syntax is of the form:

```
Layer<name> = create Layer:
  Shape <Shape>
  radius <Number>
  count <Number>
  angularShift <Boolean> (optional; default is true)
  offset <Number> (optional; default is 0)
```

Indentation indicates description of the given layer. These attributes may be defined in any given order, but they must include the attribute name correctly, and all be indented beneath the initial creation of the Layer. The only additional existing functionality of the Layer type is to be added to Mandala objects. As described above, the syntax is as follows:

```
addTo: <Mandala>, <Layer>
```

Shape is a type which represents various shapes (circles, triangles, and squares) that can be added to Layers and then drawn on Mandalas. Like the syntax for Layer, a Shape is created with an initial create constructor statement, and then provided parameters that must be indented below the initial statement. The attributes are type, size, color, and rotation. Type is the only required attribute. All other attributes are optional, and are given default values if left unspecified. Syntax:

```
Shape <name> = create Shape:
  figure <Geo>
  size <Number>
  color <Color> (optional; default is black)
  rotation <Number> (optional; default is 0)
```

Shapes may be added to Layers, or used in Templates and manipulated, as follows:

```
addTo <Layer or Template> <Shape>
```

create is the constructor for all of these custom types. The constructor creates a new instance of the type and takes parameters to fill in the various attributes of the type.

```
Type variable_name = create Type:
  attributeType attributeValue
  attributeType attribute Value
  etc.
```

For example:

```
Shape myShape = create Shape:
  figure circle
  size 5
```

Primitive Types

Number represents a floating point value, identical to the float type in C. The number range is from 1.2E-38 to 3.4E+38, and has 6 digits of precision. Numbers can be used when assigning other properties, but may also be declared on their own and assigned to variables. Examples:

```
Number x = 100

Layer l = create Layer:
  radius 4.5
  Shape <Shape>
  count 2
  offset -1.25
```

String is a wrapper over a character array, always defined as a series of characters enclosed by double quotes. Strings may be instantiated without the create method, and may also be used when assigning other properties simply by using double quotes.

```
String s = "hello"
```

Boolean can take a value of either true or false, which evaluate to 1 and 0, respectively.

```
Boolean foo = false
```

Array is used to hold a collection of types. Arrays can only hold elements the same type. I.e., in the following example, e1, e2, . . . e1N must all be of the same type.

```
<type>[] arrayName = {el1, el2, ... , elN}
```

Geo can be one of either Circle, Square or Triangle, and is used to define a Shape.

```
 Geo g = Circle
```

Type Conversion

There is no type conversion in Mandala. Where some languages differentiate between ints and floats for instance, Mandala just has one Number type, which is used for all numerical values. Any created variable must be created with a corresponding type, and it will remain that type for its entire existence during compilation and runtime.

BUILT-IN FUNCTIONS

addTo

```
  addTo: <Mandala>, <Layer>, <Layer>, ... <Layer>
```

Once a Layer is defined, in order to actually make that Layer a part of the drawable Mandala, the addTo function must be used. The addTo function must have at least two arguments – the Mandala, and at least one layer to be added. These added layers now become a part of the Mandala.

draw

```
 draw: <Mandala>
```

Draw is used to execute the program and actually draw the Mandala figure. Without this function call, the Mandala will exist as an abstract structure, but will never materialize on a user's screen. Draw takes all layers and their shapes that have been added to the Mandala and displays them to the user's screen.

Accessors

Accessors can be used to access the attributes of custom types (Mandala, Layer, and Shape). Examples:

```
 Layer[] mylayers = m1.getLayers
```

The above expression will return an array of the layers that are in that Mandala.

```
 Shapes[] myshapes = layer1.shapes
```

This above expression will return the shapes in a particular layer.
For example, if we create a shape circle1 as defined below:

```
Shape circle1 = create Shape:
  figure Circle
  size 3
  color yellow
  rotation 0
```

We can access the attributes of circle1 in the following way:

```
Geo mytype = circle1.type
Number mysize = circle1.size
Number mycolor = circle1.color
Number myrotation = circle1.rotation
```

There are also accessors for attributes of a Layer:

```
Layer layer1= create Layer:
  radius 10
  Shape circle1
  count 8
  offset 0
```

You can access the Layer attributes as follows:

```
Number myradius = layer1.radius
Shape mytype = layer1.Shape
Number mycount = layer1.count
Number myoffset = layer1.offset
```

Note again, that all of these accessors are defined only for custom types.

EXPRESSIONS

Literals

Literals are strings and numbers.

Primary Expressions

identifier
Identifiers are primary expressions.
literals
Literals are primary expressions. They are described above.
constant
A decimal or character constant is a primary expression.
String
A String is a primary expression. String literals are primary expressions, as described above.
(expression)
Parenthesized expressions are primary expressions. The type and value of a parenthesized expressions is the same as that of the expression without the delimiters. Parentheses allow expressions to be evaluated in a desired precedence. Parenthesized expressions are evaluated relative to each other starting with the expression that is most deeply nested.
primary-expression(expression-list)
Primary expressions followed by a parenthesized list of expressions is a primary expression.
expression[expression-list] Primary expressions followed by a bracketed list of expressions is a primary expression.
expressionexpression-list
Primary expressions followed by a braced list of expressions is a primary expression.

Unary Operators

expression++
The result is the incrementation of the expression by one after the evaluation of the expression, if the expression is contained in an lvalue expression. The type of the expression must be Number.
expression- -
The result is the decrementation of the expression by one after evaluation of the expression, if the expression is contained in an lvalue expression. Type of the expression must be Number.

++expression
The result is the incrementation of the expression by one before the evaluation of the expression, if the expression is contained in an lvalue expression. The type of the expression must be Number.
–expression
The result is the decrementation of the expression by one before the evaluation of the expression, if the expression is contained in an lvalue expression. The type of the expression must be Number.
not expression
The result is a Boolean indicating the logical not of the expression. The type of the result must be Boolean or Number, where 0 represents false and all other values true.
- expression
The result is the negative of the expression. The type of the expression must be Number. The type of the result is also Number.

Arithmetic Operators

expression * expression
The result is the product of the two expressions. The types of the expressions and the result must be Number.
expression / expression
The result is the quotient of the expressions, where the first expression is the dividend and the second is the divisor. The types of the expressions and the result must be Number.
expression % expression
The result is the remainder of the division of the expressions, where the first expression is the dividend and the second is the divisor. The signs of the expressions are ignored, and the absolute value of the remainder is returned. The types of the expressions must be Number.
expression + expression
The result is the sum of the expressions. The types of the expressions must be Number.
expression - expression
The result is the difference of the first and second expressions. The types of the expressions must be Number.
expression ˆ expression
The result is the exponentiation of the first expression by the second expression. The types of the expressions must be Number.

Relational Operators

expression == expression
The result is a Boolean indicating whether the two expressions are equivalent by value. The types of the expressions must be the same.
expression > expression
The result is a Boolean indicating whether the first expression is greater than the second by value. The types of the expressions must be Number.
expression < expression
The result is a Boolean indicating whether the first expression is less than the second by value. The types of the expressions must be Number.
expression >= expression
The result is a Boolean indicating whether the first expression is greater than or equal to the second by value. The types of the expressions must be Number.
expression <= expression
The result is a Boolean indicating whether the first expression is less than or equal to the second by value. The types of the expressions must be Number.

Logical Operators

expression or expression
The result is a Boolean indicating the logical or of the expressions. The types of the expressions must be the same.
expression and expression
The result is a Boolean indicating the logical and of the expressions. The types of the expressions must be the same.
expression xor expression
The result is a Boolean indicating the logical xor of the expressions. The types of the expressions must be the same.

Assignment Operators

Assignment operators have left associativity.
lvalue = expression
The result is the assignment of the expression to the lvalue. The type of the expression is the same as that of the lvalue.

Comma Operators

expression, expression
A pair of expressions separated by a comma is evaluated left to right and the value of the left expression is discarded. The type and value of the result are the type and value of the right expression. This expression should be avoided in those situations wherein the comma operator has a different meaning, such as in lists and function calls.

Constant Expressions

Syntactically, constant expressions are expressions restricted to a subset of operators. These are expressions that evaluate to a constant. Constant expressions may not contain assignments, unary operators, function calls, or comma operators.

Operator Precedence

Primary expressions and binary operators have left associativity. Unary operators have right associativity. Assignment operators have left associativity.

The precedence of operators is determined by the order of the sections in which they are shown above (with the highest precedence operators at the top). Operators within a section have the same precedence.

DECLARATIONS

Function Declarations

Mandala supports user-defined functions that are defined using the keyword def preceding each function definition. Arguments are given as a list, along with their types. The function signature ends with a colon and the body of the function is denoted via indentation.

```
def returnType funcName (argType funcArg1, argType funcArg2):
  # function body
  return <funcReturnValue>
```

Variable Declarations

For the custom types in Mandala (Mandala, Layer, Shape, and Geo), the create keyword is used to instantiate variables. For other primitives, this is unnecessary. However, for all types, the type being created must be specified upon variable instantiation.

```
type varName = create type

Mandala m = create Mandala

Shape <name> = create Shape:
  figure <Geo>
  size <Number>
  color <Color>
  rotation <Number>

Layer <name> = create Layer:
  radius <Number>
  Shape <Shape>
  count <Number
  offset <Number>
```

Primitive types can be declared as follows:

```
Number n = 100
String s = create String: "hello"
Boolean b = false
Number[] = {2, 3, 1, 0}
```

Scope and Linkage

Mandala will use block scoping, which means that any variable defined within a given level of indentation is accessible only within that level and any deeper level of indentation. Note that any Shapes that are defined within a Layer will still be drawn in a final Mandala, but the variable names are no longer accessible once outside of the Layer's indentation block.

Pre-processing

Mandala does not support any sort of pre-processing of variables or additional files.

Line Splicing

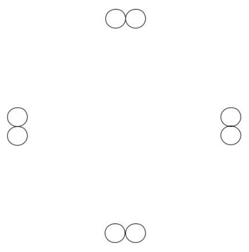
For long code statements, a user may need to split the line between two or more lines. While Mandala is designed to be mostly short simple statements, a backslash will be used to split lines. When using a backslash, no additional indentation is used in separate lines – the indentation of all following lines should match the indentation of the original lines.

```
Layer l = create Layer:
   offset 2 + 3.5 * 9.789 \
   + 27
   radius 6
   count 5
   Shape circle1
```

APPENDIX A: SAMPLE PROGRAMS

Example 1

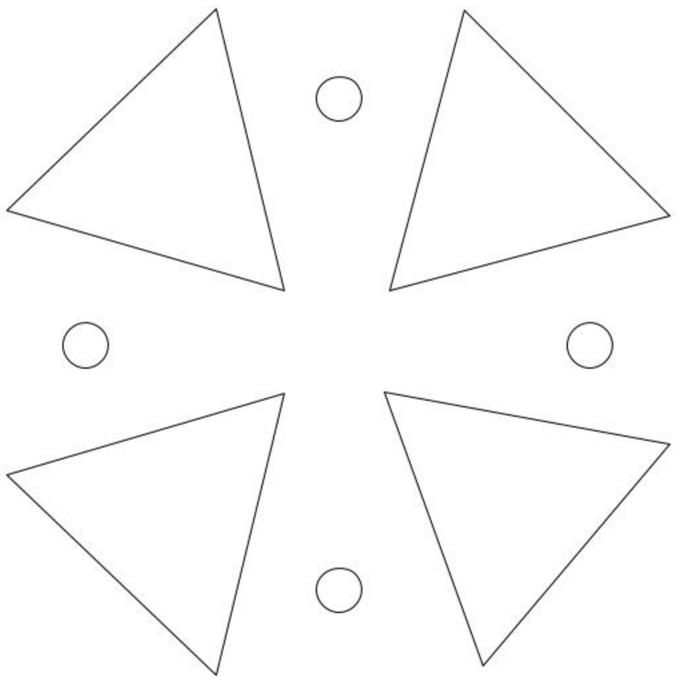
```
/# Creates and draws a Mandala of specified radius with one shape of specified type
   Draws eight shapes in a staggered fashion #/

def Void weirdlySpacedEight(Geo shape_name, Number r):
   Mandala m = create Mandala
   foreach i = 1 to i = 8:
     Shape temp_shape = create Shape:
       figure shape_name
       size r / 5
       rotation 0

     Number x = 5
     if i > 4:
       Number x = 85
     Layer temp_layer = create Layer:
       radius r
       Shape temp_shape
       count 1
       offset x + (i % 4 - 1) * 90
     addTo: m, temp_layer
   draw: m

weirdlySpacedEight: Circle, 10
```



Example 2

```
/# Creates a mandala with alternating circles and triangles
   of specified size and count #/

def Layer[] alternatingLayers(Number s1, Number s2, Number num_each):

  Shape myCircle = create Shape:
    figure Circle
    size s1
    rotation 0

  Shape myTriangle = create Triangle:
    type Triangle
    size s2
    rotation 45

  Layer temp1 = create Layer:
    radius 12
    Shape myCircle
    count num_each
    offset 0

  Layer temp2 = create Layer:
    radius 12
    Shape myTriangle
    count num_each
    offset (360 / num_each) / 2

  Layer[] layers = {temp1, temp2}

  return layers

layers = alternatingLayers(1, 5, 4)
```

```
def Void drawAllLayers(Layer[] layers):

  Mandala m = create Mandala
  foreach i = 1 to i = layers.size:
    addTo: m, layers[i-1]
  draw: m

drawAllLayers
```



Appendix B: Grammar

```
program:
  decls EOF

decls:
  fdecl
  vdecl
  cdecl

fdecl:
  DEF RETURN_TYPE ID LPAREN formals_opt RPAREN COLON vdecl_list stmt_list

formals_opt:
  /* nothing */
  | formal_list                  (*this is the list of formal parameters *)

TYPE:
    SHAPE
  | LAYER
  | MANDALA
  | NUMBER
  | STRING
  | GEO

formal_list:
    TYPE ID
  | formal_list COMMA TYPE ID

vdecl_list:
  /* nothing */
  | vdecl_list vdecl

vdecl:
  TYPE ID
```

```
stmt_list:
  /* nothing */
  | stmt_list stmt

stmt:
    expr
  | RETURN expr
  | IF LPAREN expr RPAREN stmt %prec NOELSE
  | IF LPAREN expr RPAREN stmt ELSE stmt
  | FOREACH expr_opt TO expr_opt COLON stmt

expr_opt:
  /* nothing */
  | expr

expr:
    LITERAL
  | ID
  | expr PLUS expr
  | expr MINUS expr
  | expr TIMES expr
  | expr DIVIDE expr
  | expr EQ expr
  | expr NEQ expr
  | expr LT expr
  | expr LEQ expr
  | expr GT expr
  | expr GEQ expr
  | ID ASSIGN expr
  | ID COLON actuals_opt
  | expr COMMA expr
  | LPAREN expr RPAREN
  | ID LBRACKET expr RBRACKET
  | PLUS PLUS expr
  | MINUS MINUS expr
  | expr PLUS PLUS
  | expr MINUS MINUS

actuals_opt:
  /* nothing */
  | actuals_list

actuals_list:
  expr
  | actuals_list COMMA expr

cdecl:
  (* define constructor declarations if they are different *)
  /* nothing */
  | CREATE TYPE COLON construct_args



construct_args:
  (* attributes for constructor *)
  mandala_args
  | layer_args
  | shape_args

mandala_args:
  /* nothing */

layer_args:
    SHAPE expr RADIUS expr COUNT expr OFFSET expr ANGULARSHIFT expr
  | SHAPE expr RADIUS expr COUNT expr OFFSET expr
  | SHAPE expr RADIUS expr COUNT expr ANGULARSHIFT expr
  | SHAPE expr RADIUS expr COUNT expr

shape_args:
    GEO expr SIZE expr COLOR expr ROTATION expr
  | GEO expr SIZE expr COLOR expr
  | GEO expr SIZE expr ROTATION expr
  | GEO expr SIZE expr
```