L-System Fractal Generator: Language Reference Manual

Michael Eng, Jervis Muindi, Timothy Sun

Contents

1 Program Definition **3**

2 Lexical Conventions **3**

2.1 Comments . . . 3

2.2 Identifiers . . . 3

2.3 Keywords . . . 3

2.4 Constants and Literals . . . 4

2.5 Operators . . . 5

2.6 Punctuators . . . 5

3 Meaning of Identifiers **6**

3.1 Scoping . . . 6

3.2 Object Types . . . 6

4 Type Conversions **7**

5 Expressions and Operators **7**

5.1 Precedence and Associativity Rules . . . 7

5.2 Primary Expressions . . . 8

5.3 Function Calls . . . 8

5.4 Arithmetic Operators . . . 9

5.5 Relational Operators . . . 10

5.6 Equality Operators . . . 10

5.7 Boolean Operators . . . 10

5.8 Assignment Operator . . . 11

5.9 Constant Expressions . . . 11

6 Declarations **11**

6.1 Function Declarations 11

6.2 Variable Declarations 12

7 Statements **13**

7.1 Expression Statement 13

7.2 If Statement 13

7.3 While Loops 13

7.4 Return Statements 13

8 System Functions **14**

8.1 Turtle Functions 14

8.2 The print() Function 15

1 Program Definition

A program in our language is made up of statements which consist of function declarations, function implementations and expression statements. A program in our language is to be written in a single source file, and combining different source files is currently not supported.

The structure of the program in the source file is as follows:

```
<<function declarations and implementations at top of source file>>

<<main program code>>
```

The main program code section should contain code that will be executed when the program is first run. It is the the entry point of the program. Unlike the function declaration, the main program code does not have a "function signature" attached to it. Instead, any code that immediately follows the last defined function is assumed to be part of the main program code.

An example of the program structure:

```
def compute sqrt(double x){
  # computes square root of x
}
def draw hilbert(int n){
  # code to draw the Hilbert curve
}
hilbert(10); # paint and draw a Hilbert curve on-screen.
```

A program written in any other order causes a syntax error and will be reported as such by the compiler.

2 Lexical Conventions

2.1 Comments

Comments are single-line and prefaced by the `#` symbol.

```
#This is a comment.

hilbert(5); #This is also a comment
```

2.2 Identifiers

An identifier is a sequence of letters, digits, and underscores (`_`) in which the first character is not a digit. An identifier can consist of both upper and lower case letters. The language is case-sensitive and will differentiate identifiers with identical letters in different cases.

2.3 Keywords

The following terms are a list of reserved keywords in the language and cannot be used for any other purpose:

```
alphabet
boolean
compute
def
double
draw
else
false
if
int
lambda
return
rules
string
true
while
```

2.4 Constants and Literals

Our language provides functionality for literals (also known as constants) of type `int`, `double`, `string`, and `boolean`. If any of these literals are assigned to a variable, that variable's declared type must match up with the literal's type- no automatic conversion will occur.

Integer constants

An integer constant consists of a sequence of numbers without a decimal point. An example:

```
int x = 45;
```

where 45 is the integer constant. We do not provide support for octal or hexadecimal representation of integers.

String constants

A string constant is enclosed in double quotation marks, such as `"apples"`. We provide support for the following escape sequences within string constants:

Character Name	Escape Sequence
newline	`\n`
horizontal tab	`\t`
double quotation marks	`\"`
backslash	`\\`

Examples of statements involving string constants include:

```
string s = "string\n";
print("|column 1 \t column 2 \t column 3 \t |\n");
```

String concatenation is supported in our language and is accomplished by using the `+` operator.

Double-precision floating point constants

A double constant consists of a sequence of numbers and a decimal point. At least one number must appear before the decimal point. Examples of valid double constants include:

```
0.345
0.0
1.0793211
3.141592654
```

But do not include sequences like:

```
.314
.010
0.0.0
..0
.02.2
2..9
```

Boolean constants

Boolean constants consist of the keywords true and false. A valid example of their use is:

```
boolean b = true;
if(b==true){
  #code
}
```

2.5 Operators

An operator is a symbol that denotes a specific operation that is to be performed. Below is a list of operators:

Symbol	Explanation
(,)	These must occur in pairs and can be separated by an expression in between.
+	Performs addition
+	Performs subtraction, or the unary minus operator
+	Performs multiplication
+	Performs division
+	Performs the modulo operation
+	Performs assignment

Operators are discussed in greater detail later on in the manual.

2.6 Punctuators

A punctuator is a symbol that does not specify a specific operation to be performed. It is primarily used in formatting code and it does have meaning. A punctuator can be only one of the symbols below:

Symbol	Explanation
;	Statement terminator
{, }	Used for grouping code, e.g. in functions

3 Meaning of Identifiers

3.1 Scoping

The region of a program in which a certain identifier is visible and thus accessible, is called its scope. The scoping in our language is *global*. That is, all identifiers declared are visible throughout the entire program. Also, identifiers become visible only after being declared and thus, it is illegal to refer to identifiers that have not yet been declared. For example :

```
int foo = 5;
int bar = 10;
int x = 25;
int sum = foo + bar + x + y;  # This is illegal. You cannot access y.
int y = 15;
```

Function Scope

Similar to the scoping of identifiers, functions also have global scope. Additionally, you cannot refer to a function that has not been seen yet. For example :

```
def compute sqrt(x) {
  ...
}

def compute twice_square_root(x) {
double root =  sqrt(x)
return add(root, root) # This is illegal. Add must be declared before it's used.
}

def compute add(x,y){
 ...
}
```

3.2 Object Types

Our language supports the four fundamental types of objects:

- integer
- floating point
- string
- boolean

Character Types

The only supported character type is the string type. This can store a string of potentially unlimited length and does not have an upper bound limit. The length is only limited by the amount of computing resources (e.g. memory) available.

Integer and Floating Point types

The only supported integer type is int which can store 32-bits worth of data and the only supported floating point type is double which can store 64-bits of data. Both of these data types are signed.

Boolean Type

This type is a truth value and can only store a bit of information. Specifically, it may only take a value of either `true` or `false`.

4 Type Conversions

Integer type variables (also known as ints) can be converted into double type variables with casting. The reverse cannot be performed. When casting, the converted variable must have a different identifying name than the original variable. For example,

```
int x = 45;
double y = (double)x;
```

is valid, but

```
double x = 45.0;
int y = (int)x;
```

and

```
int x = 45;
double x = (double)x;
```

will cause a compilation error.

5 Expressions and Operators

5.1 Precedence and Associativity Rules

The language follows classical mathematical order of operations, prioritizing multiplication and division over addition and subtraction. The logical AND operator takes precedence over the logical OR operator. Expressions inside parentheses are evaluated with top precedence.

Expression	Result	Comments
`3 + 2 * 6`	`15`	Multiplication occurs before addition.
`3 + (2 * 6)`	`15`	Expression within parentheses is evaluated first, though the answer doesn't change from the above expression.
`(3 + 2) * 6`	`30`	Expression within parentheses is evaluated first.
`FALSE || TRUE && FALSE`	`FALSE`	The AND operator takes precedence over the OR operator.
`(FALSE || TRUE) && TRUE`	`TRUE`	The expression within the parentheses is evaluated first.
`FALSE || (TRUE && FALSE)`	`FALSE`	The expression within the parentheses is evaluated first.

The following table is a list of operator precedence and associativity for our computational functions, adapted from the C language reference manual. The relevant operators for drawing functions have the same priority as listed in this table. The table is ordered from highest to lowest priority.

Tokens	Operators	Class	Associativity
Identifiers, literals, parenthesized expressions	Primary expression	Primary	
`( )`	Function calls	Postfix	L
`(type)`	Casting	Unary	R
`-`	Unary Minus	Binary	R
`^`	Exponentiation	Binary	R
`* /`	Multiplicative	Binary	L
`+ -`	Additive	Binary	L
`< <= => >`	Relational comparisons	Binary	L
`== !=`	Equality comparisons	Binary	L
`&&`	Logical AND	Binary	L
`||`	Logical OR	Binary	L
`=`	Assignment	Binary	R
`,`	Comma	Binary	L

5.2 Primary Expressions

The following are all considered to be primary expressions:

- Identifiers: An identifier refers to either a variable or a function. Examples include `x_1` and `hilbert`, but not `2nd`.
- Constants: A constant's type is defined by its form and value. See Section 2.4 for examples of constants.
- String literals: String literals are translated directly to Java strings by our compiler, and are treated accordingly.
- Parenthesized expressions: A parenthesized expression's type and value are equal to those of the unparenthesized expression. The presence of parentheses is used to identify an expression's precedence and its evaluation priority.

5.3 Function Calls

Our language supports two kinds of functions: compute functions and drawing functions. To be able to call a function, it must have been declared and implemented previously in the program. That is, it is a syntax error to call a function which has not yet been seen by the compiler. Also, recursive function calls are not supported at present. The syntax of a compute function call is

```
Function_Name(Argument_Parameter_List);
```

The call must match and agree with the signature of a previously declared function. The grammar for the compute function is:

```
identifier (argument-expression-list)
argument-expression-list: argument-expression
                          argument-expression-list, argument-expression
```

The identifier refers to the name of a user-defined function. For example, assume there is a compute function with the signature below:

```
def compute sum(int a, b){ ... }
```

It can be called like so:

```
sum(1,3);
```

On the other hand, the drawing function will always have at least one parameter, and the first parameter an integer that corresponds to depth of the L-system, that is, how many times the L-system production rules are applied in drawing it. For example assume there is a draw function with the signature below:

```
def draw DragonCurve(int n){ ... }
```

It can be called like so:

```
DragonCurve(10);
```

The tentative grammar for the draw function call is :

```
identifier (argument-expression-list)
```

5.4 Arithmetic Operators

Arithmetic operators involving manipulating arithmetic expressions. The grammar that defines what is an arithmetic expression is given below:

```
arithmetic expression:  NumberLiteral
                          VariableIdentifier
```

The operators `*` (multiply), `/` (divide), and `%` (mod) are what we call multiplicative operators and they group from left to right. These operators form what we call multiplicative expressions. The grammar for these expressions is:

```
multiplicative expression :  arithmetic expression
        multiplicative-expression * arithmetic expression
        multiplicative-expression / arithmetic expression
        multiplicative-expression % arithmetic expression
```

The `*` operator performs arithmetic multiplication and its operands must have arithmetic type (i.e. be numbers of either type)

The `/` operator performs arithmetic division and operands must also have arithmetic type.The result of this operation is similar to the type of the operands. That is, if both the operands are integers, then so is the result. If both are doubles, then so is the result. If the types do not match up, it is a syntax error.

The binary `%` (mod) operator gives the remainder from the division of the first expression (the dividend) by the second (the divisor).

The operators `+` and `-` are known as additive operators and they associate from left to right. The grammar syntax for additive operators is given below :

```
additive-expression:  multiplicative-expression
      additive-expression + multiplicative-expression
      additive-expression - multiplicative expression
```

The operator `+` performs arithmetic addition and so the operands must have arithmetic type type. The value returned by this operator is the sum of the operands.

Note that it is not possible to use the `+` operator to perform string concatenation.

Similarly, the operator - performs arithmetic subtraction and so the operands must also have arithmetic type. The value returned by this operator is the difference between the two operands.

5.5 Relational Operators

Relational operators are used to compare variables to one another in relational expressions. The two variables on either side of a relational operator must share the same type. For instance, `"I don't know" < 3` would cause a compiler error. A relational expression will evaluate to a boolean constant, that is, to `true` or `false`. Strings, ints, and doubles are the valid types for comparison when using this type of operator. With respect to strings, if `a` and `b` are string variables, `a < b` checks to see if `a` comes before `b` in the lexicographical order, `a > b` checks to see if `a` comes after `b` in the order, `a <= b` checks for `a < b` and also evaluates to `true` if `a` is equivalent to `b` (see below for equality operator definitions), and `a >= b` checks for `a > b` and also evaluates to true if `is equivalent to \verb`b=. The grammar for relational expressions is as follows:

```
Relational expression:
  Variable rel_operator variable
Variable:
  Arithmetic expression
  constant or literal (e.g., previously instantiated int d, 6, or "rope")
Rel_operator:
  <
  <=
  >
  >=
```

5.6 Equality Operators

The equality operators work the same way as the relational operators. However, they check to see if a variable is equal or not equal to another variable. Again, the two variables in an equality expression must share the same type. Strings, ints, and doubles are valid types to use with these operators. With respect to strings, `==` will only evaluate to true if the strings are exactly the same, that is, they are case sensitive.

```
Equality expression:
  Variable eq_operator variable
Variable:
  Arithmetic expression
  constant or literal
Eq_operator:
  ==
  !=
```

5.7 Boolean Operators

The language features support for the logical AND and logical OR operations. For logical AND, if both values being ANDed together are true, then the expression evaluates to true. Otherwise, the expression evaluates to false. For logical OR, if one or both values being ORed together is true, the expression evaluates to true. Otherwise, the expression evaluates to false.

A boolean expression takes the form of `boolean boolean_operator boolean`. The expressions on each side of the `boolean_operator` (`&&` or `||`) must evaluate to booleans. As a result, the expressions on each side

of the boolean operator must be either boolean constants (true or false), relational expressions (`a < b`), or equality expressions (`g == h`).

```
Boolean expression:
  Boolean expression && Boolean expression
  Boolean expression || Boolean expression
  Boolean
Boolean:
  Equality expression
  Relational expression
  Boolean_constant
Boolean_constant:
  true
  false
```

5.8 Assignment Operator

The assignment operator, `=`, is used to associate an identifier with a value. For instance,

```
int d = 4;
```

will cause the identifier `d` to be associated with a value of 4, and to have type int. As such, the next time d is used in an expression, it will evaluate to 4. The type declared on the left hand side of the assignment must match the expression on the right hand side of the expression. That is, an identifier declared as a boolean cannot be assigned a value that is a double literal.

5.9 Constant Expressions

Constant expressions are expressions that evaluate to string and number literals. For instance, `"test"`, `4`, and `4.266` are all constant expressions. Assigning a non-boolean type value to a variable using the assignment operator, and then using that variable's identifier also evaluates to a string or number literal, and is classified as a constant expression. For instance:

```
int d = 4;
int f = d + 3;
```

`f` would equal 7, since `d` evaluates to 4 after its assignment.

Constant expressions are a subset of primary expressions.

6 Declarations

6.1 Function Declarations

Our language supports two kinds of functions: compute functions and drawing functions. All functions must be preceded with the `def` keyword. Functions are both declared and implemented at the same time. Thus if one compute function calls another compute function, for example, then that other function must have been declared and implemented before. Otherwise, this is a syntax error. Also, drawing functions cannot be called in a compute function, but compute functions can be called in a drawing function.

Compute Functions

Compute functions are normal functions and they typically do some kind of computation like finding the square root of a number. All compute functions must have the `compute` specifier after the `def` specifier and they must also always return a value of type double (i.e. there is no need to specify a return type). If a function does not need to return a value, then it may simply return the value 0 to indicate that it completed successfully. Compute functions can have any number of named argument parameters. The argument parameter passing mechanism in our language is always pass-by-value. An example of a compute function declaration and implementation is below :

```
def compute sum (int a, int b){
  return a + b;
}
```

Drawing Functions

Drawing functions are the functions that specify the structure of an L-system that will be eventually be drawn. These functions take in at least one parameter, and the first parameter must be an integer. This parameter specifies how many times the L system rules will be applied in drawing the L system. The structure of the body of a drawing function is as follows:

```
alphabet: (List of letters separated by commas);
rules: List of rules;
```

The syntax for the rules is:

```
letter -> letter_sequence
```

There is a special rule that must be included in every L-System, defined as `lambda -> letter_sequence`. This indicates the initial state of the L-System on which the production rules are applied. An example of a drawing function:

```
def draw hilbert(int n){
  alphabet: (A,B;f,r,l,s);
  rules:{
    lambda -> s A;
    A -> l B f r A f A r f B l;
    B -> r A f l B f B l f B r;
    A = ;
    B = ;
    f = turtle_move(150/3^n);
    r = turtle_turn(-90);
    l = turtle_turn(90);
    s = turtle_setpos(0,0);
  }
}
```

6.2 Variable Declarations

Variable declarations are used to set variables equal to computed values. The type used in a declaration must match with the type that the expression on the right hand side of the assignment operator returns. A variable declaration consists of the following grammar:

```
Variable declaration:
  Type identifier = variable_expr
Type:
  int
  double
  bool
  string
variable_expr:
  string_literal
  compute_function_expression
  arithmetic expression
  relational expression
  equality expression
  constant expression
```

7 Statements

7.1 Expression Statement

An expression statement is composed of primary statements with a semicolon at the end of the line.

7.2 If Statement

If statements come in the following two varieties:

```
if (expression)
  statement1
if (expression)
  statement1
else
  statement2
```

For both `if` statements, `expression` must be of boolean type, and `statement1` executes if `expression` evaluates to `true`, while `statement2` in the second variety executes if `expression` evaluates to `false`.

7.3 While Loops

While loops allow for executing a statement any number of times.

`while (expression) statement`

As with if statements, `expression` must be of boolean type, and statement executes until the expression evaluates to `false`. The evaluation of expression comes before the execution of the statement.

7.4 Return Statements

A compute function returns a value to the caller through return statements.

```
return number;
return;
```

The returned result of the function, `number`, must be of double type. In the case of the second type, a so-called "dummy" return, the function returns `0.0`.

8 System Functions

8.1 Turtle Functions

For L-system drawings, we use turtle graphics, which draws images based on a user supplying relative positioning commands on a cursor. One can imagine the cursor being a turtle with a pen on its tail, and the user telling the turtle to go forward, turn, or lift its tail. In the following functions, r, theta, x, and y are all of double type. Furthermore, let `X`, `Y`, and `Theta` be the position and orientation of the cursor, respectively, and let `down` be the current state of the "tail."

`turtle_move(r)`

- The turtle moves "forward" by `r` pixels. Formally, the change in `X` and `Y` is cos(`Theta`)`r` and sin(`Theta`)`r`, respectively. If `down` is set to `true`, then the line between (`X`,`Y`) and (`X`+cos(`Theta`)`r`,`Y`+sin(`Theta`)`r`) is drawn.

`turtle_turn(theta)`

- The turtle turns counterclockwise by `theta` (in degrees). Formally, the change in `Theta` is `theta`.

`turtle_up()`

- This function sets `down` to `false`.

`turtle_down()`

- This function sets `down` to `true`.

`turtle_isdown()`

- This function returns `down == true`

`turtle_getangle()`

- This function returns `Theta`.

`turtle_setangle(theta)`

- This function sets `Theta` to `theta`.

`turtle_getx()`

- This function returns `X`.

`turtle_gety()`

- This function returns `Y`.

`turtle_setpos(x,y)`

- This function sets `X:=x`, `Y:=y`.

8.2 The print() Function

The `print()` function takes in any expression or literal and prints out the result. That is, it can directly print out arithmetic expressions and it can print out string literals. For example,

```
int x = 7;
print(x);
```

prints out 7, while

```
print("This is a Hilbert curve.");
```

prints out exactly that string.