LGA Language Reference Manual - V0.1

Hang Qian
Pindan Hao
Yuanli Dong
Tian Xia
{hq2124, ph2389, yd2270, tx2126}@columbia.edu
While .......................................................... 9
For ............................................................. 10
3.5 Others ......................................................... 10
Assign .......................................................... 10
Assignable ....................................................... 10
AssignObj ....................................................... 11
ObjAssignment ............................................... 11
Array .......................................................... 11
Object .......................................................... 11
AssignList ..................................................... 11
OptComma ..................................................... 12
Return .......................................................... 12
Comment ....................................................... 12
Invocation ...................................................... 12
Arguments ...................................................... 13
ArgList ........................................................ 13
FuncGlyph ...................................................... 13
ParamList ...................................................... 13
Param .......................................................... 13
Index .......................................................... 14
Parenthetical ................................................... 14

4 Sample Code ................................. 15
Chapter 1

Introduction

LGA, with a unique syntax, is designed to make writing expressive manipulation of graphics and animation very easily. By targeting on Javascript, LGA can be viewed as an alternative to handle web graphical and animation task.

The syntax of LGA is inspired by R\textsuperscript{1}, Coffeescript\textsuperscript{2} and Python\textsuperscript{3}. It is designed to be minimalized and expressive, just focus on the target task. Compare to the target language Javascript, LGA provides a more succinct, expressive and programmer-friendly syntax and core functions.

In this manual, we provide the comprehensive lexical and syntactic details of LGA, in order to easily generate the front-end part of the compiler. The following of this manual is structured as follows: Chapter 2 consists of the lexical details of LGA; Chapter 3 consists of the syntactic details of LGA; Chapter 4 gives a brief sample code.

\textsuperscript{1}R-Project: http://www.r-project.org/
\textsuperscript{2}Coffeescript: http://coffeescript.org/
\textsuperscript{3}Python: http://python.org/
Chapter 2

Lexical Conventions

2.1 Tokens

There are five classes of tokens: identifiers, keywords, literals, operators and separators. LGA use whitespaces as separators (similar to Python programming language). If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

Comments

The character # introduce a comment and the next \n (newline) terminates it. Comments do not occur within string or character literals.

Identifiers

Identifiers (names) must start with _ or any lowercase and uppercase letters. The rest of the string can contain the same characters plus number digits ('0' - '9').

Keywords

The following identifiers are reserved for the use as keywords, and may not be used otherwise. These include the superset of both JavaScript keywords and reserved words.

Numeric Literals

An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero), decimal otherwise. A floating constant consists of an integer part, a decimal part, a fraction part, an e or E, an optionally signed integer exponent and an optional type suffix, one of f, F, l, or L. The integer
Table 2.1: Keywords

and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

Character Literals

A character literal is a single character surrounded by single quotes.

String Literals

A string constant is a sequence of characters surrounded by double quotes. Double quotation marks can be contained in strings surrounded by single quotation marks, and single quotation marks can be contained in strings surrounded by double quotation marks.

2.2 Operators

Computational operators

\(+\) \(-\) \(*\) \(/\) \(\%\)

Compound assignment operators

\(-=\) \(+=\) \(/=\) \(*=\) \(\%=\) \(|=\) \&\&\= \?\=? \(<\<=\) \(\>=\) \&= \^= \|=\)

Logical operators

\&\& \|| \& | ^ !

Comparison operators

\(==\) \(!=\) \(<\) \(<=\) \(>=\)
Chapter 3

Grammar

3.1 Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by typewriter style and forms as a capitalized word, as Expression. Literal words, tokens, and characters are in all upper letter words, as TERMINATOR.

3.2 Program structure

Root

The Root is the top-level node in the syntax tree. Since we parse bottom-up, all parsing must end here.

Root:
  NULL
  Body
  Block TERMINATOR

Body

The Body node is any list of statements and expressions.

Body:
  Line
  Body TERMINATOR Line
  Body TERMINATOR

Line

Line:
  Expression
  Statement
Statement
Statement:
  Return
  Comment
  STATEMENT

Expression
Expression:
  Value
  Invocation
  Code
  Operation
  Assign
  If
  While
  For

Code
The Code node is the function literal. It’s defined by an indented block of Block preceded by a function arrow, with an optional parameter list.

Function is a body of executable code which gets specific number of parameters then process statements inside the function body and return values if needed. The following is some examples for functions:

\[
\begin{align*}
\text{sum} &= (x, y) \rightarrow x+y \\
\text{getdouble} &= (x) \rightarrow \text{sum} x, x \\
\text{givemefive} &= () \rightarrow 5 \\
\text{sayhey} &= (\text{hey}) \rightarrow \text{hey}
\end{align*}
\]

In the first case, the function named sum, input parameters are x and y. For the second case which has name of getdouble, it has only one parameter which is x. Function body starts after the arrow operator. In the first case, statements in function body sum up two input parameters. In the second case, it calls sum, the function has been defined previously, and pass the input parameter x to sum. Of course, LGA supports functions without any input parameter like shown in the third case. In that case, no parameter will be defined and there will only be a pair of empty parentheses. Default parameters are also well covered in LGA just like the fourth case. A function could be called with its name followed by a pair of parentheses inside which contains parameters if defined. The second is an example of calling function sum and passing x as parameter. At the end of function, the final value will be returned to the caller. For example, in the first case, the returned value will be (x+y). Parameters will be passed to functions by value. Any modification on the parameter inside a function will not put any influence on the original object/variable.
Value

Value literal is the types of things that can be treated as values, which means they can be assigned to, invoked as functions, indexed into, etc.

Value:
   Assignable
   Literal
   Parenthetical
   This

Block

LGA use whitespaces as levels of indentation. A Block is an indented block of expressions.

Block:
   INDENT OUTDENT
   INDENT Body OUTDENT

3.3 Identifiers, Numerics and Literals

Identifier

A literal identifier, which is a variable name or property.

Identifier:
   IDENTIFIER

AlphaNumeric

AlphaNumeric is separated from the other Literal matchers because they can also serve as keys in object literals.

AlphaNumeric:
   NUMBER
   STRING

Literal

All of immediate values. Generally these are fully compatible with our target language, which means can be printed
Literal:
  AlphaNumeric
  NULL
  BOOL

This
This:
  THIS
  @

  ThisProperty is a reference to a property on this
ThisProperty:
  @ Identifier

3.4 Control Flow

LGA supports common If, While and For loop as control flows.

If

Addition to regular if block, LGA supports Post-If style. For example x = 2 if y > 3. Code in block will be executed if evaluation result of expression is boolean true.
If:
  IfBlock
  IfBlock ELSE Block
  statement POST_IF Expression
  Expression POST_IF Expression

IfBlock:
  IF Expression Block
  IfBlock ELSE IF Expression Block

While

Similar to if block, Post-While style is support in LGA. For example x = x * 2 while x < 100. Code in block will be executed repeatedly if evaluation result of expression is boolean true.
While:
  WhileSource Block
  Statement WhileSource
  Expression WhileSource

WhileSource:
  WHILE Expression
For

For block iterate through `ForValue`. For example, in `for x in [1, 2, 3, 4]`, `x` is repeated assigned as elements in the array. To iterate an `Object`, use two `ForValues` separated by comma.

For:
   ForBody Block

ForBody:
   ForStart ForSource

ForStart:
   FOR ForVar

ForVar:
   ForValue
   ForValue, ForValue

ForValue:
   Identifier
   Array
   Object

ForSource:
   FORIN Expression

3.5 Others

Assign

Assignment of a variable, property, or index to a value

Assign:
   Assignable = Expression
   Assignable = TERMINATOR Expression
   Assignable = INDENT Expression OUTDENT

Assignable

This category consists of everything that can be assigned to.

Assignable:
   SimpleAssignable
   Array
   Object

SimpleAssignable:
   Identifier
   ThisProperty
AssignObj

Assignment when it happens within an object literal. The difference from the ordinary Assign is that these allow numbers and strings as keys. And we use as assign operator here.

AssignObj:
  ObjAssignable
  ObjAssignable : Expression
  ObjAssignable : INDENT Expression OUTDENT
  Comment

ObjAssignment

ObjAssignable:
  Identifiers
  AlphaNumeric
  ThisProperty

Array

Array:
  [ ]
  [ ArgList ]

Object

In LGA, an object literal is simply a list of assignments.

Object:
  { AssignList OptComma }

AssignList

Assignment of properties within an object literal can be separated by comma, as in Javascript, or simply by newline

AssignList:
  NULL
  AssignObj
  AssignObj , AssignObj
  AssignList OptComma TERMINATOR AssignObj
  AssignList OptComma INDENT AssignList OptComma OUTDENT
OptComma

An Optional, trailing comma.

OptComma:
  NULL
  ,

Return

In LGA, functions will always return their final values even though when we
don’t actually use any return statement or operator, as following: \( \text{sqr} = (x) \rightarrow x \times x \). The value of \( x \times x \) will be returned to the caller as the final value of the function. As shown, flowing off the end of function then the final value will be returned. Of course, return statement with which a function can return to the caller is also provided. Examples of using return statement are as following:

```
return
return ( expression )
```

The first sample dose not return any value but just terminal the process. The second sample returns value of the expression to the caller. With a return statement, logical flow of a function could be easily controlled. When some specific cases are captured, function could be terminated with or without returning a value.

Return:
  RETURN Expression
  RETURN

Comment

LGA only support inline comment, starting with a # and terminates with a newline

Comment:
  COMMENT

Invocation

Ordinary function invocation.

Invocation:
  Value Arguments
  Invocation Arguments
Arguments
The list of arguments to a function call.

Arguments:
  CALL_START CALL_END
  CALL_START ArgList OptComma CALL_END

ArgList
The ArgList is both the list of objects passed into a function call, as well as the contents of an array literal.

ArgList:
  Expression
  ArgList , Expression
  ArgList OptComma TERMINATOR Expression
  INDENT ArgList OptComma OUTDENT
  ArgList OptComma INDENT ArgList OptComma OUTDENT

FuncGlyph
FuncGlyph:
  ->

ParamList
The list of parameters that a function accepts can be of any length

ParamList:
  NULL
  Param
  ParamList , Param
  ParamList OptComma TERMINATOR Param
  ParamList OptComma INDENT ParamList OptComma OUTDENT

Param
Param:
  ParamVar
  ParamVar = Expression

ParamVar:
  Identifier
  Array
  Object
  ThisProperty
Index

Indexing into an object or array using bracket notation.

Index:
  INDEX_START IndexValue INDEX_END

IndexValue:
  Expression

Parenthetical

Parenthetical:
  ( Body )
  ( INDENT Body OUTDENT )
move_forward = (l) ->
  if @pos && @vec
    a = math.atan(@vec[0], @vec[1])
    @pos[0] += math.cos(angle) * l
    @pos[1] += math.sin(angle) * l
  return

square = {
  run : circle
  pos : [2, 5]
  vec : [1, 1]
  delay : 5
}

OBJ = [square]
OBJ.start()