Lucifer Language Reference Manual

Michael Fagan (mef2224)
Elliott Morelli (gnm2123)
Cherry Chu (ccc2207)
Robert Becker (rb3307)

February 24th 2021

Contents

1 Lexical **4**
1.1 Identifiers 4
1.2 Comments 4
1.3 Separators 4
1.4 White Space 4
1.5 Reserved Keywords 4

2 Primitive types **5**
2.1 Integer 5
2.2 Float 5
2.3 Char 6
2.4 Boolean 6

3 Literals **6**
3.1 Integer Literals 6
3.2 Float Literals 6
3.3 Bool Literals 7
3.4 Char Literals 7

4 Arrays **7**

5 Expressions **8**
5.1 Primary Expressions 8
5.2 Precedence of Operators 8
5.2.1 Expressions formed from function calls 9
5.2.2 Expressions formed from object function calls 10
5.2.3 Expressions formed from array accesses 10
5.3 Statements 10
5.4 Declarations 10

5.4.1 Assignments . . . 11
5.4.2 Object Instantiations . . . 11
5.4.3 Blocks and Control Flow . . . 11

6 Operators **11**
6.1 Arithmetic Operators . . . 11
6.2 Comparison Operators . . . 12
6.3 Negation Operator . . . 12
6.4 Logical Operators . . . 13
6.5 Assignment Operators . . . 13

7 Program Structure **13**
7.1 The `if` Statement . . . 13
7.2 The `while` Statement . . . 14
7.3 The `for` Statement . . . 14
7.4 The `runGame()` Statement . . . 15
7.4.1 How to use runGame() . . . 15
7.5 What rungame() does internally . . . 15
7.6 Scope . . . 16

8 Standard Functions **16**
8.1 Function Declarations . . . 16
8.2 Function Calls . . . 16
8.3 Return Statements . . . 17

9 Built-in Functions **17**
9.1 The `init()` function . . . 17
9.2 The `checkKey(<int>)` function . . . 17

10 Built-in Classes **18**
10.1 Constructing Built-in Objects . . . 18
10.2 Accessing and Updating Instance Variables of Built-in Objects . 18
10.2.1 Accessing Instance Variables . . . 18
10.2.2 Updating Instance Variables . . . 18
10.3 Using Functions of Built-in Objects . . . 19
10.4 The `Entity` Class . . . 19
10.4.1 Instance Variables . . . 19
10.4.2 Constructor . . . 19
10.4.3 Functions . . . 19
10.5 The `Player` Class . . . 20
10.5.1 Player Instance Variables . . . 20
10.5.2 Player Constructor . . . 20
10.5.3 Player Functions . . . 21
10.6 Extending Classes . . . 21
10.6.1 Syntax Rules for Extending Classes . . . 21
10.6.2 Fields . . . 22

10.6.3 Constructor 22
10.6.4 Adding Functions 22
10.6.5 Extended Class Example 22

11 Sample Code **23**

1 Lexical

1.1 Identifiers

Identifiers in Lucifer are sequences of characters are composed of ASCII letters, decimal digits and the underscore character _ .

The first character of an identifier has to be a letter.

1.2 Comments

Lucifer supports both multi-line and single-line comments. Comments are ignored by parser.

Single-line comment starts with double backslash //.

```
int r = 1; // r is radius of a circle
```

Multi-line comment are enclosed by /* and */.

```
float x = 2.91;

/*

This is a multi line comment

*/
```

1.3 Separators

Separators separate tokens. Lucifer accepts `( ) { } [ ] ; , .` as separators.

Example of separator usage

```
int x = (1 + 2) * (2 + 3); x + 1;
```

1.4 White Space

White Space characters are separator that are discarded during parsing. Valid white space characters include space characters and newline characters. Tab characters are invalid.

1.5 Reserved Keywords

```
return

if

elif

else
```

```
noelse
for
while
int
bool
true
false
float
void
char
string
class
new
extends
fun
Entity
Player
runGame
```

Lucifer also accepts the predefined SDL Scancodes as keywords. They can be looked up in `https://wiki.libsdl.org/SDLScancodeLookup`

2 Primitive types

Lucifer supports four primitive types of objects: integers, character literals, single-precision floating point numbers and booleans.

2.1 Integer

Integer has 4 bytes.The 32-bit int data type can hold integer values in the range of -2,147,483,648 to 2,147,483,647.

Here is an example to declare and define integer variable.

```
int a = 1;
```

2.2 Float

Float has 8 bytes.Float data type has size ranges from 1e-37 to 1e37.

An example to declare and define float variable is

```
float foo = 3.14;
```

2.3 Char

Char has 1 byte and it can be any ASCII characters.

2.4 Boolean

Boolean has 1 byte. It is a binary data type with either true or false value. Boolean values are represented by the reserved keyword `true` and `false`.

3 Literals

Lucifer supports integer literals, float literals, char literals and bool literals.

3.1 Integer Literals

An integer literals is a sequence of decimal digits.

The regular expression for integer literal is

```
[0-9]+
```

Examples of integer literals are

```
72
0
1327
```

The context free grammar for integer literal in the parser is:

```
expr:  LITERAL
```

3.2 Float Literals

A float Literal consists of an integer part, a decimal point, a fraction part, an e and an optionally signed integer exponent.

The regular expression representing the float literal is

```
((([0-9]+[\.][0-9]*)|([\.][0-9]+))(([e][+-]?[0-9]+)?))|([0-9]+[e][+-]?[0-9]+)
```

Examples of float literals are

```
0.5e+15

8.3

1.

13e9
```

The context free grammar for float literal in the parser is:

```
expr:  FLIT
```

3.3 Bool Literals

A bool literal is represented by keywords `true` and `false`.

The context free grammar for bool literal in the parser is:

```
expr:  BLIT
```

3.4 Char Literals

A char literal is a sequence of ASCII characters enclosed by double quotation characters (" "). If the char literal content contains any double quotation characters, the character should be preceded by the escape character \.

The regular expression for string literals is

```
"([^"\\]|\\.)*"
```

Example of char literals are

```
"Hello World!"

"foo"

"bar\"baz"

"abc345@#&"
```

The context free grammar for char literal in the parser is:

```
expr:  CLIT
```

4 Arrays

Arrays in Lucifer are indexed collections of values. Lucifer supports array literals containing primitive types and other arrays.

Syntax for initializing an array is as follows:

```
<valtype> [] <arrayid> = new <valtype>[<arrsize>]
```

Where <`valtype`> is either a <`type`> as seen in 3.5.1, an array type, a <`parent`> class name(Entity or Player), or an extended class identifier. <`arrayid`> is the unique identifier for the array, and <`arrsize`> is an integer literal that represents the size of the array.

1. Values in arrays can be accessed and overwritten.

Brackets indicate and access to array elements:
Syntax:

> <`arrid`> `[`<`index`>`]` where <`arrid`> is the unique identifier for the array and <`index`> is some expression that evaluates to an integer.
>
> For multidimensional arrays, additional brackets would be used to access values of nested arrays.
> <`arrid`> `[`<`index1`>`][`<`index2`>`]`
>
> `arr[0]` //access to element at index 0 in array arr
>
> `arr[0][1]` //access to element at index 1 in array element arr[0]

5 Expressions

5.1 Primary Expressions

Primary expressions involving '.' and function calls group left to right.
Expressions include the following:

1. Literals, whose grammar rules are described in 3.1

2. Identifiers, who have the token `ID`

5.2 Precedence of Operators

Operators in Lucifer follow rules of precedence and associativity that determine the order in which expressions will be evaluated.

Table 2 lists each operator and indicates the precedence and associativity of each. Table 3 shows some simple examples of precedence and associativity. Parentheses can be used to override these rules.

The list below shows the grammar rules for operator expressions:

`expr:`

```
expr + expr
expr - expr
expr * expr
expr / expr
expr % expr
expr == expr
expr != expr
expr <= expr
expr > expr
expr >= expr
expr && expr
expr || expr
-expr
!expr
```

Tokens (From High to Low Priority)	**Operators**	**Associativity**
`!  -`	Unary negation	R-L
`* / %`	Multiplicative	L-R
`+ -`	Additive	L-R
`< <= > >=`	Relational comparisons	L-R
`== !=`	Equality comparisons	L-R
`&&`	Logical AND	L-R
`||`	Logical OR	L-R
`=`	Assignment	R-L
`,`	Comma	L-R

Table 1: Operator Precedence and Associativity

5.2.1 Expressions formed from function calls

```
func-call-expr:
identifier.(actualparams-list)
```

Expression	Results	Comments
`3 + 2 * 5`	13	Multiplication is done before division
`3 + (2 * 5)`	13	Same order as before, but parentheses provide clarification
`(3 + 2) * 5`	25	Parentheses override the precedence rules
`true || true && false`	true	Logical AND has higher priority than logical OR
`true || (true && false`	true	Same order as before, but parenthesis provide clarification
`(true || true) && false`	false	Parentheses override the precedence rules

Table 2: Precedence and Associativity Examples

5.2.2 Expressions formed from object function calls

```
obj-func-call-expr:
identifier.expression(actualparams-list)
```

5.2.3 Expressions formed from array accesses

```
arr-access-expr:
expression [ expression ]
```

5.3 Statements

5.4 Declarations

Syntax for declaring variables takes the following form:
`<type> <identifier> ;`

where a `<type>` is any of the following:

```
type:

   int
  bool
  float
  char
```

Rules for function declarations can be found in Section 8.1

5.4.1 Assignments

Primitive Variable Assignment syntax:

```
<type> <varidentifer> = <expression>;
```

where the left side of the statement is the variable's type and identifier and the right hand side is an expression.

5.4.2 Object Instantiations

Object Instantiations take the following form:
`<objtyp> <objidentifier> = new <objtyp> (args_opt);`

Where `<objtyp>` is either the identifier of an extended class , or the keyword name of a built-in class `(Entity, Player)`, `<objidentifier>` is the identifier for the object, `new` is the keyword for instantiation, and `args_opt` is an optional list of actual parameters that the constructor takes.

5.4.3 Blocks and Control Flow

If statements, While Statements, For Statements, and the runGame() statement are all considered `statements` in Lucifer.
Their rules and syntax can be found in Section 7.

6 Operators

6.1 Arithmetic Operators

Lucifer supports standard two-operands arithmetic operations: addition, subtraction, multiplication, division, and modulo. Two-operands operations require the two operands to be of the same type. For the modulo operation, both operands must be of type integer.

```
/* Addition */

a = 1 + 3;

b = 3.4 + 5.2;

c = a + b;

/* Subtraction */

a = 1 - 3;

b = 3.4 - 5.2;

c = a - b;
```

operator	**meaning**	**usage**
`==`	equal to	`foo == bar`
`!=`	not equal to	`foo != bar`
`>`	greater than	`foo > bar`
`<`	less than	`foo < bar`
`>=`	greater than or equal to	`foo >= bar`
`<=`	less than or equal to	`foo <= bar`

Table 3: Comparison Operators

```
/* Multiplication */
a = 1 * 3;
b = 3.4 * 5.2;
c = a * b;

/* Division */
a = 1 / 3; // resulting quotient is truncated to integer value
b = 3.4 / 5.2;
c = 20.3 / b;

/* Modulo */
a = 17 % 3;
b = 5 % a;
```

Lucifer also supports single-operand arithmetic negation.

```
/* Negation */
int a = -2;
float b = -3.14;
```

6.2 Comparison Operators

Comparison operators compare the two operands and determines how they relates to each other. The two operands are `expression`tokens that must evaluate to the same type, and the result of comparison is a boolean type.

Details of each operators and their usage are illustrated in Table 1.

6.3 Negation Operator

Negation operator is denoted by the character ! and is applied to a single operand of boolean type to negate its value.

Example of negation operator usage

```
if (!(foo == 1)) print("hello world.");
```

6.4 Logical Operators

Lucifer support the OR and AND logical operators. Logical operator evaluates the truth value of the two operands of boolean type.

The OR operator `||` evaluates to true if either of the operands is true, false otherwise.

```
if ((bar == 2) || (foo == 1)) print("bar is 2 and foo is 1.");
```

The AND operator `&&` evaluates to true if both operands are true, false otherwise.

```
if ((bar == 2) && (foo == 1)) print("Either bar is 2 or foo
is 1.");
```

6.5 Assignment Operators

Assignment operator is denoted by the character =. It is used to define or assign value to a variable.

The left operand is a variable and the right operand is a value to be stored in the variable on the left.

Example of assignment operator usage

```
int a;

a = 3 + 2;
```

7 Program Structure

A typical program structure in Lucifer contains extended class definitions and function definitions, followed by a main() function.
In Lucifer, statements allowed in global scope are class definitions, variable declarations, and function definitions.
The starting execution point for a program in Lucifer is the main() function which is required to be the last function definition in the program file.

7.1 The `if` Statement

The if/else statement in Lucifer supports branching of program flow based on the evaluation of a boolean expression. An if/else statement introduces a new scope and therefore must be surrounded by curly brackets.
The `if` statement has the following general form

```
if (condition) {then-statement}
else {else-statement}
```

When there is no action for the else part, the statement takes the following form.

```
if (condition) {then-statement}
noelse
```

When there is more than one condition to compare, `elif` statement is used

```
if (condition) {then-statement}
elif (condition) {then-statement}
else {else-statement}
```

The context free grammar for `if` statement is:

```
stmt:  IF (expr) stmt elif_opt ELSE stmt
stmt:  IF (expr) stmt elif_opt NOELSE
elif_opt:  ϵ | elif_opt ELIF (expr) stmt
```

7.2 The `while` Statement

The while loop statement has the following syntax:

```
while ( expression ) {statement}
```

The statement within the loop is executed repeatedly so long as the value of the expression evaluates to true. The value of the expression is tested before each execution of the statement.

The context free grammar for `while` statement is:

```
stmt:  WHILE (expr) stmt
```

7.3 The `for` Statement

The for loop statement has the following syntax:

```
for (init; condition; increment)
{statement}
```

Only the condition part is required, while the init and increment parts are optional. The increment part updates the condition after every loop. The statement within the for loop is executed repeatedly as long as the condition is true.

The context free grammar for `for` statement is:

```
stmt:  FOR (expr_opt; expr; expr_opt) stmt
```

7.4 The `runGame()` Statement

The runGame() loop in Lucifer is responsible for making internal calls to SDL library functions in order to render Entity and Player objects in a window.

7.4.1 How to use runGame()

The `runGame()` loop should be the last statement in a Lucifer program, within the main() method. It can contain expressions.

runGame() takes three integer arguments: desired framerate, display window width, and display window height. These arguments of runGame() are optional.

```
main(){

  runGame(<int> framerate,<int> window_width, <int> window_height){
      expressions
        }
}
```

7.5 What rungame() does internally

1. runGame() internally calls SDL_SetRenderDrawColor() and SDL_RenderClear() to prepare the window for rendering.
2. runGame() internally calls SDL_PollEvent() in order to update the `keyboard` scancode array, which is accessible through the checkkey() function.
3. runGame() runs all code written within its function body. This should be game logic that updates positions and states of Players and Entities. In this stage, runGame() calls loadTexture() and SDL_QueryTexture() for each Player and Entity.
4. runGame() draws any Entities and Players that are accessible in the scope of its function body by calling SDL_QueryTexture() and SDL_RenderCopy().
5. runGame() renders the updated scene by calling SDL_RenderPresent().
6. runGame() caps the framerate of the game by calling SDL_GetTicks() and SDL_Delay().

7.6 Scope

Variables declared within a function have the scope of that function, and are accessible only within that function's body. Likewise, variables declared in class definitions, if statements, and while statements are only accessible within the subscopes created by those statements. Subscopes are designated using curly brackets: {}.

8 Standard Functions

8.1 Function Declarations

Functions in Lucifer have the following syntax:

```
fun <functionId> <returnType> (<type> arg1,<type> arg2...<type>
argn){function body}
```

Where `functionId` is a unique identifier and `returnType` is the type that the function returns.The arguments are variable declarations that are passed into the function body when the function is called. Arguments are optional, but must have their type stated in the function definition.

Example of function declaration:

```
fun add int (int a, int b){

  return a + b;

  }
```

In this example, `add` is the functionId, `int` is the returnType, and `int a, int b` are the arguments.

8.2 Function Calls

When calling a function, the function call must have the same number, ordering, and type of actual parameters as the arguments declared in the function declaration.

To supply actual parameters to a function and run the function body, the following syntax is used:

```
functionId(actual parameters);
```

Example of a function call:

```
  int g = 5;
  int h = 10;
  int res = add(g,h);
```

In this example, `add(g,h);` runs the function body of the `add` function and supplies `g` and `h` as actual parameters.

8.3 Return Statements

The `return` statement allows a function to return an expression of the type specified in its function declaration. If the type of the expression in the `return` statement does not match the return type of the function, the compiler will throw an error.The keyword `void` is used in the function declaration if the function does not return anything.

```
return ( expr_opt ) ;
```

9 Built-in Functions

9.1 The `init()` function

The init() function in Lucifer is responsible for initializing SDL and internally creates its window. It internally calls SDL_CreateWindow(), SDL_CreateRenderer(), SDL_SetHint(), and IMG_Init().

It must be called in the main() before a rungame() loop can be used.

```
main(){

  init();

}
```

9.2 The `checkKey(<int>)` function

The checkkey(¡int¿) function in Lucifer returns true if the key matching the scancode passed to it has been pressed since its last call.

```
main(){

  checkKey(<int>);

}
```

10 Built-in Classes

Lucifer has built-in Entity, and Player classes which may be used to construct objects.

10.1 Constructing Built-in Objects

Built-in Objects are instantiated with Javalike syntax, using the `new` keyword:

```
<classid> <obectid> = new <classid> (args_opt);
```

Example:

```
Player p = new Player(75,100, "texture.png",[82,81]);
```

After constructing an object, refer to it by its variable name to access its variables and functions.

10.2 Accessing and Updating Instance Variables of Built-in Objects

To access or update an object's instance variable, use the '.' character before the variable name.

10.2.1 Accessing Instance Variables

Syntax for accessing intstance variables of objects is as follows:

```
<objectid> . <variableid>;
```

Example:

```
Player p = new Player(75,100, "texture.png",[82,81]);
int i = p.x;
//i now holds the integer value of 75
```

10.2.2 Updating Instance Variables

```
Player p = new Player(75,100, "texture.png",[82,81]);
int i = 40;
p.x = i;
//p.x now holds the integer value of 40
```

10.3 Using Functions of Built-in Objects

To call an object's function, use the '.' character before the function name and two parentheses `()` after the function name.
Actual parameters of the function, if any, should be in between the parentheses:

```
<objectid>.<functionid>(actual_list_opt);
```

Example:

```
 p.addHitbox(50,100);
```

10.4 The `Entity` Class

An Entity in Lucifer describes any game object that will be rendered on the SDL Window. Entity serves as a superclass for the Player object.

10.4.1 Instance Variables

Entity stores a 2D (x, y) position, the name of a texture file(JPG,PNG) and a 2D hitbox (hx,hy).

```
int x;
int y;
String texture;
int hx;
int hy;
```

10.4.2 Constructor

The expected parameters for the Entity constructor are its starting (center) x position, its starting y position, and its texture file for its sprite.

```
Entity e = new Entity(50,50, "texture.png");
```

10.4.3 Functions

```
fun void addHitbox(int x, int y);
```

Updates hitbox variables hx and hy.

```
 fun void changeX(int dx);
```

Updates x to x+dx.

```
fun void changeY(int dy);
```

Updates y to y+dy.

10.5 The `Player` Class

In Lucifer, Player Objects are sprites that can be directly controlled by hardware input. Player is a subclass of Entity.

10.5.1 Player Instance Variables

Player objects inherit Entity instance variables and functions. Player objects also contain an array of integers that store the keycodes for player controls.

```
//Inherited from Entity
int x;
int y;
String texture;
int hx;
int hy;

//Extended by Player
int[] controls;
```

10.5.2 Player Constructor

The Player constructor has four arguments: starting (center) x position, its starting y position, its texture file for its sprite, and an int value used to designate the size of its internal, array, which is an array of SDL Scancodes.

SDL Scancodes refer to hardware input from specific keyboard keys. Their corresponding integer values can be found in this reference: https://wiki.libsdl.org/SDLScancodeLookup Either an integer value or the SDL Scancode value is accepted, as shown below:

```
Player p = new Player(75,100, "texture.png",2);

p.addControl(0,SDL_SCANCODE_UP);

Player p = new Player(75,100, "texture.png",2);

p.addControl(0,82);
```

10.5.3 Player Functions

As a subclass of `Entity`, `Player` inherits all of the functions of `Entity`. However, Player also contains addControl() and removeControl() functions.

```
fun void addControl(int i,int newcontrol);
```

This function replaces the scancode in Player's `controls` array at index `i` with the `newcontrol` int value.

```
fun void removeControl(int i,int newcontrol);
```

This function replaces the scancode in Player's `controls` array at indexiwith the `newcontrol` int value.

10.6 Extending Classes

In Lucifer, functionality can be added to `Entity` and `Player` classes by writing extended class definitions.

The keywords Entity and Player are tokenized as `parent`s in Lucifer's grammar. These new objects will inherit the basic functions of the objects they extend, and can have an updated constructor, additional instance variables and functions added to them.

10.6.1 Syntax Rules for Extending Classes

Use the `class` keyword to begin a class definition, followed by a unique identifier for the extended class.
The class identifier should be followed by the `extends` keyword along with a `parent` name (Entity or Player).

```
class <classidentifier> extends <parent>( ){

  vdecl_list

  constructor

  fnct_list
```

}

In this extended class definition, `vdecl_list`
refers to a list of variable declarations, `constructor`
refers to the constructor syntax which can be found in 10.6.3, and `fnct_list`
refers to a list of function declarations, whose syntax can be found in Section 8.1

10.6.2 Fields

Inside of an extended class, the superclass' instance variables will be accessible without needing to specify them. Any instance variables entered inside the class will be added to the extended class.

10.6.3 Constructor

If a constructor is not written for a class, then it will default to calling super() for its constructor.

An extended class can also take in additional parameters for its constructor. Each additional parameter will have a type followed by its name in the constructor's definition.

If a class is written with a defined constructor, then super() can be called to initialize the values as the parent would by passing the corresponding values along.

```
  <classidentifier> ( formals_opt ) {

    stmt_list

}
```

10.6.4 Adding Functions

Functions can be added to the class by creating a function inside of the extended class.

Once created, these new functions can be used in the same way as other class functions by entering the class name, followed by a . and the function name, with any parameters that need to be passed through in parentheses.

10.6.5 Extended Class Example

class myEntity extends Entity {

```
    int value; //additional field for myEntity

    myEntity(int x, int y, String texture, int hx, int hy, int myValue){

        super(x, y, texture, hx, hy); // call Entity's constructor

        value = myValue; // initializes value based on constructor
        // parameter

    }

    fun int addxy(){

        return x + y;

    }

}
```

11 Sample Code

Here is the code for running a simple game with an obstacle and a player that displays the player and entity as stationary textures.

```
-------------------------------------------------------
init();
Entity rock = new Entity(50,50, "rock.png");
rock.addHitBox(5, 5);
Player p1 = new Player(75,100, "player1.png",2);
p1.addControl(0,81);
runGame(){ //statements go here };
```