LIVA

A Lite Version of Java

Shanqi Lu
sl4017

Jiafei Song
js4984

Zihan Jiao
zj2203

Yanan Zhang
yz3054

Table of Contents

CHAPTER 1 INTRODUCTION

Liva is a general purpose programming language and a lite version of Java. It is designed to let programmers who are familiar with class-based languages feel comfortable with developing common algorithms like GCD. It is lite in the sense that it maintains some but not all features in Java. It has the similar syntax and abstract data types in Java and supports object-oriented paradigm and inheritance. However, generics and nested classes are beyond the scope of this project, hence they are not to be implemented.

The Liva programming language is strongly typed. The compiler checks whether arguments passed to a function match expected types and return an error if not. It is a portable language and compiled down to LLVM.

This language reference manual is organized as follows:

- Chapter 2 describes the lexical conventions of the Liva programming language.
- Chapter 3 describes types. Types are divided into two categories: primitive types and reference types.
- Chapter 4 describes classes including class declarations and inheritance.
- Chapter 5 describes statements.
- Chapter 6 describes expressions.

CHAPTER 2 LEXICAL CONVENTIONS

This chapter specifies the lexical conventions of Liva programming language. A compiler takes a program which consists of a sequence of characters and reduce it to a sequence of elements, which are tokens, white space and comments. The tokens are identifiers, keywords, literals, separators, and operators.

Element:
    White Space| Comment| Token
Token:
    Identifier| Keyword| Literal| Separator| Operator

2.1 White Space

White space in Liva is defined as space character, tab character, form feed character(page-breaking) and line terminator character. White space characters are ignored by a compiler except as they serve to separate tokens.

2.2 Comments

There is one kind of comments:

- /* text */

All characters from "/*" to "*/" are ignored.

2.3 Identifiers

An identifier is a sequence of letters, digits and underscore '_'. It can only begin with a letter. Identifiers are the names of variables, methods and classes. They are case-sensitive.

2.4 Keyword

Keywords are reserved and cannot be used as identifiers.

- *Keyword:*

for	*new*	*if*	*boolean*	*this*	*break*
double	*implements*	*else*	*import*	*return*	*extends*
int	*char*	*interface*	*void*	*class*	*float*
while					

2.5 Literals

Literals are syntactic representations of numeric, character, boolean or string data. They are used for representing values in programs.

2.5.1 Boolean Literals

There are two boolean literals:

- **`true`** represents a true Boolean value
- **`false`** represents a false Boolean value

2.5.2 Integer Literals

Integer numbers in Liva are in decimal format. Negative decimal numbers such as `-10` are actually expressions consisting of the operator '–' and integer literal. The primitive type of integer literal is `int`.

2.5.3 Floating Point Literals

Floating point numbers are expressed as decimal fractions and consist of:

- an optional '+' or '–' sign; if omitted, the value is positive,
- one of the following formats

Format			Example
integer digits			9
integer digits	.		7.
integer digits	.	integer digits	17.31
	.	integer digits	.56

2.5.4 Character Literals

Character literals are expressed as a single quote: `'a'`, `'#'`, `'π'`

2.5.5 String Literals

String literals begin with a double quote character `"`, followed by zero or more characters and a terminating double quote `"`
Within string literals, there can be escape sequences but not unescaped newline.

2.5.6 Escape Sequences for Character and String Literals

An escape sequence is used to represent a special character. It begins with a backslash character (\), which indicates that the following characters should be treated specially. Escape sequences are listed in the table below.

Name	Character
TAB	\t
newline	\n
double quote	\"
single quote	\'
backslash	\\

2.5.7 The Null Literal

Null is a special literal which represents a null value which does not refer to any object. The null literal is formed as: `null`

2.6 Separators

Separators are tokens used for separating tokens.

```
{       }       (       )       ;       ,       .
```

2.7 Operators

The expression section of this manual will explain behaviors of these operators. Here lists all the operators.

```
=    >    <    !    ==   >=
<=   !=   &   |   +    –
*     \    %
```

CHAPTER 3 Types

The Liva programming language supports two kinds of types: primitive types and reference types. There are also two kinds of data values: primitive values and reference values accordingly. There is also a special null type.

Primitive types are boolean types and numeric types. Reference types are class types and array types

3.1 Primitive Types

Primitive types are redefined and their names are reserved keywords.

3.1.1 Integral Types

The integral types are int and char.

The integer data type is a 32-bit sequence of digits, which has a minimum value of -2^31 and a maximum value of 2^31-1. An integer literal is a sequence of digits preceded by an optional negative sign. A single zero cannot be preceded by a negative sign.

```
int x = 10;
int y = -50;
int z = 0;
```

The char data type belongs to integral types whose values are 16-bit unsigned integers.

```
char x = 'a';
```

3.1.2 Floating-Point Types

The floating-point data type is a signed-precision 32-bit format values.

```
float x = 1.5;
float y = -5.1;
```

3.1.3 The Boolean Type

The boolean data type has two possible values: true and false. A boolean is its own type and cannot be compared to a non-boolean variable. Therefore, expression “true == 1” would lead to an error.

```
boolean x = true;
boolean y = false;
```

3.2 Reference Types

There are two kinds of reference types: class types and array types.

3.2.1 Objects

An object can be a class instance or an array.

3.2.2 The Class String

An instance of class String is a sequence of characters. The class String supports the following built-in methods:

- charAt(int index)
  This method returns the character located at the String's specified index but returns an error if the index is out of range. Indexes start from zero.
- length()
  This method returns the length of a string.

```
String w = "Liva";
String x = new String("Liva");
int y = x.length();
char z = x.charAt(1);
```

3.2.3 Arrays

Arrays can be seen as a special type. An array object contains a number of variables. All elements in an arrays must have the same type.

```
int[] ai;
char ac[] = { 'a', 'b', 'c', ' ' };
```

CHAPTER 4 Classes

A class declaration defines a new reference type and how it is implemented. Classes contain fields and methods. Field declarations describe class variables while method declarations describe programs that may be invoked by other programs

4.1 Class Declarations

Classes are defined in the following way. The optional extends clause in a class declaration specifies the superclass of the current class.

```
class MyClass extends SuperClass{
        //field, constructor
        // method declarations
}
```

4.2 Class Members

Members of a class consist of:

- Members inherited from its superclass
- Members declared in the class declaration

4.3 Field Declarations

Field Declarations specify variables of a class type.

4.4 Method Declarations

Method declarations specify executable code that might be invoked.

```
/* Field declarations and method declarations */
class Calculation{
    int z;

    void addition(int x, int y){
        z = x+y;
    }

    void Subtraction(int x,int y){
        z = x-y;
    }
}
```

4.5 Constructor Declarations

If the constructor is not defined, the compiler generates a default one.

```
/* User defined constructor*/
class Calculation{
    int z;
    constructor (int z){
        this.z = z;
    }
    void addition(int x, int y){
        z = x+y;
    }

    void Subtraction(int x,int y){
        z = x-y;
    }
}
```

4.5 Inheritance

Inheritance is that a subclass acquires all the behaviors and properties of a super class.

```
class My_Calculation extends Calculation {

    void multiplication(int x, int y) {
        z = x * y;
    }
}
```

CHAPTER 5 Statements

Statements include: *if, else, for, break, continue, return*, as well all expressions which are explained in the following. Except as indicated, statements are executed in sequence

5.1 Expression Statements

An expression statement consists of an expression followed by a semicolon.

```
expression;
```

Usually expression statements are assignments or function calls.

```
/* Object creation expressions */
Student e1 = new Student ();

/* Object creation expressions */
c = 8933.234;
```

5.2 Declaration Statements

A declaration statement declares a variable by specifying its data type and name. It also could initialize the variable during the declaring.

```
/* declare a variable with data type and name */
char a;
int b =10;
float c;

int array1[] = { 2, 5, -2, 6, -3, 8, 0, -7, -9, 4 };
String name= "class";

boolean isMatch = false;
```

5.3 Control Flow Statements

5.3.1 If-then and If-then-else

There are two forms of conditional statements.

For the first case, the conditional expression that is evaluated is enclosed in balanced parentheses. The section of code that is conditionally executed is specified as a sequence of statements enclosed in balanced braces. If the conditional expression evaluates to false, control jumps to the end of the if-then statement.

```
if (expression) {
      statement
}
```

In the second case the second sub-statement is executed if the expression is false. As usual the 'else' ambiguity is resolved by connecting an else with the last encountered elseless if.

```
if (expression) {
      statement1
} else {
      Statement2
}
```

5.3.2 Looping: for

The 'for' condition will also run in a loop so long as the condition specified in the 'for' statement is true. The 'for' statement has the following format:

```
for (expression1; expression2; expression 3) {
      statement
 }
```

The first expression specifies initialization for the loop and it is executed once at the beginning of the 'for' statement; the second specifies a test, made before each iteration, such that the loop is terminated when the expression becomes false; the third expression typically specifies an increment or decrease which is performed after each iteration.

The following example uses a 'for' statement to print the numbers from 0 to 10:

```
for (int num=0; num < 11; num ++) {

    print(num);

}
```

5.3.3 Looping: while

The 'while' statement has the form:

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

The 'while' statement will be executed in a loop as long as the specified condition in the while statement is true. The expression must have type boolean, or a compile-time error occurs.

- If the value for expression is true, then the contained statement is executed

    - If execution of the statement completes normally, then the entire ‘while’ statement is executed again, beginning by re-evaluating the expression.
    - If execution of the statement completes abruptly

- If the value of the expression is false, no further action is taken and the ‘while’ statement completes normally.

5.3.4 Branching: break, continue, and return

The break statement causes termination of the smallest enclosing for statement; control passes to the statement following the terminated statement. The expression for ‘break’ statement is shown below:

```
break;
```

The continue statement causes control to pass to the loop-continuation portion of the smallest enclosing for statement; that is to the end of the loop. The expression for ‘continue’ statement is show below:

```
continue;
```

A function returns to its caller by means of the ‘return’ statement, which has one of the forms:

```
return;
return(expression);
```

In the first case no value is returned when a method is declared void. For the first case, the users could specify no return statement for simplification. In the second case, simply put the value (or an expression that calculates the value) after the return Keyword, then the value of the expression is returned to the caller of the function.

5.4 Method Creation and Method Call

The user could write the user-defined methods.

```
returnType nameOfMethod (Parameter List) {
 // method body
}
```

- `returnType:` Method may return a value.
- `nameOfMethod:` This is the method name. The method signature consists of the method name and the parameter list.
- `Parameter List:` The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- `method body:` The method body defines what the method does with statements.

For using a method, it should be called. There are two ways in which a method is called i.e. method returns a value or returning nothing (no return value).

Following is the example to demonstrate how to define a method and how to call it with a returned value:

```
void main(String[] args) {
    int a = 11;
    int b = 6;
    int c = minFunction(a, b);
    prin("Minimum Value = " + c);
}

int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;
    return min;
}
```

5.5 Print to Console

The *print()* function takes one or more parameters and prints them one by one to standard output. The parameter type may be string, number, or object. It is in the following form:

```
print (parameters);
```

Here is an example to accept an *int* and print the *int* to the console.

```
print (1);
```

Another example to accept a string and print the string to the console:

```
print (“CS4115 is fun!”)
```

5.6 Empty Statement

An empty statement does nothing and has the following form:

```
;
```

CHAPTER 6 Expressions

Much of the work in a program is done by evaluating *expressions*, such as assignments to variables, or for their values, which can be used as arguments or operands in larger expressions, or to affect the execution sequence in statements, or both.

This chapter specifies the meanings of expressions and the rules for their evaluation.

6.1 Evaluation, Denotation, and Result

Liva evaluates a larger expression by evaluating smaller parts of it. So the result of an expression is important. When an expression in a program is *evaluated* (*executed*), the result denotes one of three things:

- A variable
- A value
- Nothing (for void functions and methods)

An expression denotes nothing if and only if it is a method invocation that invokes a method that does not return a value, that is, a method declared void. Such an expression can be used only as an expression statement (in statement chapter), because every other context in which an

expression can appear requires the expression to denote something.

If an expression denotes a variable, and a value is required for use in further evaluation, then the value of that variable is used. In this context, if the expression denotes a variable or a value, we may speak simply of the *value* of the expression. In this way, we may say each expression denotes a value in a certain type.

6.2 Type of an Expression

For an expression that denotes to a variable, the value stored in a variable is always compatible with the type of the variable. In other words, the value of an expression whose type is T is always suitable for assignment to a variable of type T.

The rules for determining the type of an expression that denotes to a value are explained separately below for each kind of expression. Including arithmetic operations, relation operations, bitwise/conditional operations, assignment.

6.3 Evaluation Order

Liva guarantees that the operands of operators appear to be evaluated in a specific *evaluation order*, namely, from left to right.

6.3.1 Left-Hand Operand First

The left-hand operand of a binary operator appears to be fully evaluated before any part of the right-hand operand is evaluated.

6.3.2 Evaluate Operands before Operation

Liva guarantees that every operand of an operator (except the conditional operators &, |) appears to be fully evaluated before any part of the operation itself is performed.

For example, in an assignment expression, the assignment will not be evaluated until the right hand operands (if it is another expression) is evaluated.

6.3.3 Evaluation Respects Parentheses and Precedence

Liva respects the order of evaluation indicated explicitly by parentheses and implicitly by operator precedence.

6.4 Lexical Literals

A literal denotes a fixed, unchanging value. This kind of expression could be evaluated without being broken into small expressions.

The type of a literal is determined as follows:

- The type of an integer literal is int.
- The type of a floating-point is float.
- The type of a boolean literal is boolean.
- The type of a character literal is char.
- The type of a string literal is String.
- The type of the null literal null is the null type; its value is the null reference.

Evaluation of a lexical literal always completes normally.

6.5 The Arithmetic Operations

The value of an equality expression is numeric (int or double, depends on the operands). The operators +, -, *, /, and % are called the arithmetic operators. They have the same precedence and are syntactically left-associative (they group left-to-right). The type of the arithmetic expression is the promoted type of its operands.

The type of each of the operands of arithmetic operators must be a type that is convertible to a primitive numeric type, or a compile-time error occurs. For example: “+” two objects of a user-defined class is prohibited.

+	Adds values on either side of the operator
-	Subtracts right hand operand from left hand operand
*	Multiplies values on either side of the operator
/	Divides left hand operand by right hand operand
%	Divides left hand operand by right hand operand and returns remainder

6.6 The Relational Operations

The value of an equality expression is always boolean. The equality operators may be used to compare two operands that are convertible to numeric int type, or two operands of type boolean.

All other cases result in a compile-time error.

==	(Addition) Adds values on either side of the operator
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to (no less than) the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to (no greater than) the value of right operand, if yes then condition becomes true.

6.7 The Bitwise and Conditional Operations:

Unlike Java, Liva uses &, |, ^, ~ for both bitwise and conditional operations. That depends on the operands. The operands of a Bitwise/Conditional Operation should both be `int` or `boolean`.

&	Binary AND if both operands are `int` type / Logic AND if both operands are `boolean` type
|	Binary OR if both operands are `int` type / Logic OR if both operands are `boolean` type
^	Binary XOR if both operands are `int` type / Logic XOR if both operands are `boolean` type
~	Binary Complement if both operands are `int` type / Logic INVERT if both operands are `boolean` type

6.8 Method Invocation Expressions

A method/function invocation expression is used to invoke an instance method (declared in previous chapters). The result type of the chosen method is determined as follows: If the chosen method/function is declared with a return type of void, then the result is void. Otherwise, the result type is the method/function's declared return type.

6.9 Array Access Expressions

An array access expression contains two subexpressions, the *array reference expression* (before the left bracket) and the *index expression* (within the brackets).

Note that the array reference expression may be a name or any primary expression that is not an array creation expression. The type of the array reference expression must be an array type. For the index expression, the promoted type must be `int`, or a compile-time error occurs.

The result of an array access expression is a variable of type T, namely the variable within the array selected by the value of the index expression.

6.10 Assignment

In Liva, the only assignment operator is “=”.

The result of the first operand of an assignment operator must be a variable. This operand may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access or an array access (defined previously).