W4115 Programming Languages and Translators - Spring 2017

Lava

Language Reference Manual

Team members

An Wang (aw3001) - Language Guru
Yimin Wei (yw2907) - System Architect
Jiacheng Liu (jl4784) - Tester
Hongning Yuan (hy2486) - Manager

1 Introduction

Lava is an object-oriented, statically typed language that serves general purposes. Lava code is compiled to LLVM code and runs on LLVM, which makes it hardware-independent.

Since Lava is not compiled to JVM bytecode and is not run on JVM, it cannot be linked with existing Java libraries. However, it supports importing standard C libraries in a Lava way.

Type checking is done at compile time. Programmers will get compilation errors at compile time and thus runtime typing errors are prevented. Although generics and lambda expressions enable a method or class to work with types that are unknown when the method or class are defined, the type checking is still enforced when the method is invoked and objects of the class are initialized. There is no runtime for Lava.

Lava is a language that is designed to be a dialect of Java. Therefore the syntax and most basic functionality are very similar to Java. Java programmers can pick up Lava without much trouble.

2 Lexical conventions

2.1 Identifiers

Identifiers are sequences of characters used to name classes, variables, and functions.

1. Each identifier must have at least one character.
2. Identifiers can contain letters, digits, and the underscore symbol.
3. Identifiers should not begin with a digit.
4. Keywords cannot be used as identifiers.
5. Identifiers are case sensitive.

2.2 Keywords

The table below lists the buit-in keywords of Lava.

`while`	`if`	`else`	`return`
`for`	`break`	`continue`	`extends`
`new`	`void`	`boolean`	`char`
`int`	`float`	`String`	`array`
`public`	`private`	`class`	`this`

import			

2.3 Operators

Arithmetic	Relational	Logical	Assignment
+ (Addition)	== (equal to)	&& (logical and)	=
- (Subtraction)	!= (not equal to)	|| (logical or)	+=
* (Multiplication)	> (greater than)	! (logical not)	-=
/ (Division)	< (less than)		*=
% (Modulus)	>= (greater than or equal to)		/=
++ (Increment)	<= (less than or equal to)		
-- (Decrement)			

2.4 Comments

Lava supports two kinds of comments, one-line comments and multi-line ones.

One-line comments start with "//". Anything in that line that follows the double slash will be discarded by the compiler and will not execute.

```
int a=5; //int b=6;
//The statement int b=6 will not execute.
```

Multi-line comments are wrapped between "/*" and "*/". Everything that are between the start of such a comment, namely "/*", and the end of the comment, namely "*/", will be discarded by the compiler and will not execute.

```
int a=5;
/* Everything in the multi-line comment will not execute.
int b=6;
```

```
For example, the above line will not execute.
*/
```

The comments will not be treated like Lava code, nor are they designed to be. Comments are for things other than code.

2.5 whitespace

whitespace refers to the following characters: space(‘ ‘), tab(‘\t’), and newline(‘\n’). Any amount of whitespace are allowed outside entities, e.g. identifiers, keywords, and will be ignored.

3 Data Types

Lava is a statically typed language. A variable’s type must be specified when declared.

3.1 int

The int type is the type for integers. Capable of containing at least the [-2147483648, +2147483647] range;thus, it is at least 32 bits in size.

```
int num = 3;
```

3.2 float

The float type represent single-precision floating-point format. It occupies 4 bytes (32 bits) in computer memory and represents a wide dynamic range of values by using a floating point. In IEEE 754-2008 the 32-bit base-2 format is officially referred to as binary32. And Lava doesn’t support distinction between float and double.

```
float num = 3.14;
```

3.3 boolean

The boolean type is a binary indicator which can be set to either true or false. A boolean value can be one of the two values, either true or false.

```
boolean b = true;
```

3.4 char

char is the data type for characters. A character constant is a single character enclosed with single quotation marks, such as ‘a’.

```
char c = ‘a’;
```

3.5 String

String is a sequence of characters. Characters in a string a indexed, which starts at 0. A string's length equals to the number of characters it includes. String is not primitive type and has multiple functions with it.

```
String s = "Welcome to Lava";
```

3.6 Array

Array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, the length of array is fixed.

```
int[] array_example = new int[10];
```

4 Statements

A statement expresses some action to be carried out.

4.1 Expression Statements

An expression statement consists an expression to be executed, ended by a semicolon.

```
i = 1;                      /*Assignment Expression*/
Circle.draw();              /*Method invocation*/
c = new Circle();           /*Class object creation*/
```

4.2 Declaration Statements

A declaration statement declares a variable by specifying its name and type. The initialization can also be done within the declaration statement.

```
int x;
boolean checked = false;
Shape s = new Shape();
```

4.3 Control Flow Statements

Statements are generally executed in sequence, i.e. from top to bottom. The execution flow however can be altered by control flow statements, including condition statements, loop statements and branching statements.

4.3.1 Condition statement

The condition statements, also called "if statement", executes a section of code only if the specified condition is met. An `if` statement can be followed by an `else` statement, which

executes when the condition in the `if` statement is not met. These two statements can also be combined and nested.

Plain if statement

```
if (expression) {
      //code to execute when the expression is true
}
```

If and else statement

```
if (expression) {
      //section to execute when the expression is true
} else {
      //code to execute when the expression is false
}
```

Combined if else statement

```
if (expression1) {
      //code to execute when the expression1 is true
} else if (expression2) {
      //code to execute when the expression1 is false and
condition2 is true
} else {
      //code to execute when neither expression1 nor expression2 is
met
}
```

An example of *condition statement*

```
/*printing result based on score*/
if (score > 100) {
      print(“Good”);
} else if (score < 60) {
      print(“Poor”)
} else {
      print(“Fair”)
}
```

4.3.2 Loop Statements

Lava supports the loop statements defined by `for` and `while` keyword.

For statement

The for statement executes a section repeatedly as long as the specified condition is met.

```
for (initialization;condition;loop-execution) {
      //code to execute in the loop
}
```

There are three parameters in a for loop statement: initialization, condition and loop-execution

The initialization initializes the loop variable, whose scope is within the for-loop only. There can be more than one loop variables.

The condition is checked before every loop, if it is met then the loop body executes, otherwise the loop ends.

The loop-execution is executed after every loop, which is usually used to update loop variables.

All three parameters are optional.

An example of *for statement*

```
/*printing 012345678*/
for (int i=0;i<10;i=i+1) {
      print(i);
}
```

While statement

The while statement executes a section repeatedly as long as the specified condition is met.

```
while (condition) {
      // code to execute in the loop
}
```

The while statement executes the loop body as long as the condition is met.

An example of while statement

```
/*Loading the truck as long as it is not full*/
while (truck.isFull()==false) {
      truck.load();
}
```

4.3.3 Branching statements

Lava supports two branching statements, break and continue, which are used inside the loop body.

Break

A break statement terminates the loop of which it is included.

```
/*Breaking for loop*/
for (int i=0;i<1000;i++) {
      if (find(i)== true) {
            print("Found the item at " + i);
            break;
      }
}

/*Breaking while loop*/
i = 0;
while(true) {
      if (find(i)== true) {
            print("Found the item at " + i);
            break;
      }
      i++;
}
```

5 Functions

Functions are a piece of code which can be called to execute.

5.1 Function declaration&definition

```
access-modifier return-type function-identifier (parameters) {
      //function body
}
```

A function must be declared and defined before being used. Function declaration & definition includes access-modifier, return-type, function-identifier, parameters and function body.

`access-modifier`: `public` or `private`, which defines the accessibility of the function. Public functions can be accessed outside the class, while private function can only be accessed within the class.

`return-type` : The type of the return value of the function, which can be built-in type or user defined type (class).

`Function-identifier` : The identifier of the function, which must be unique within the class.

`function body` : The code to be executed when the function is called.

`parameters` : The parameters passed to the function, whose scope are within the function body.

A sample function:

```
public int square(int i) {
      return i*i;
}
```

6 Class

6.1 Class Definition

A class definition starts with the keyword 'class' and follows by the class name. The body of a class is limited in a pair of parentheses. One class can inherit multiple base classes. One class should contain a constructor and may contain the following elements: fields, methods. The following is a typical definition of a class.

```
class foo extends bar {
      //constructor
      public bar() {
            //code
      }
      public void sampleMethod() {
            //code
      }
      private int sampleField;
}
```

6.2 Fields and Methods

Fields are variables in class. Fields can be assigned with an initial value.

Methods should be invoked with an instance of class, such as

```
      instance.method();
```

In addition, both fields and methods can be defined as static, which allows user to invoke a function without an instance of class, such as

```
classname.method();
classname.variable;
```

If a method is declared as static, only static fields can be used in its body.

6.3 Constructor

The constructor is the function called when an instance is created. The constructor should have the exactly same name as the class name and be declared as public.

There is no default constructor. The constructor must be defined explicitly.

```
class foo extends bar {
      //constructor
      public bar(int a, int b) {
            //code
      }
      private int sampleField;
}
```

6.4 Access Modifiers

Two modifiers are supported in Lava, `public` and `private`. Public methods and fields can be referenced outside the class while private ones couldn't.

Access Modifiers can also be used to declare static functions. In such case, access modifiers should be placed ahead of the keyword 'static'

```
public static void foo() {...}
```

6.5 Inheritance

One class can inherit multiple base classes with the keyword 'extends'. Class names should be splitted by commas. If a class inherits multiple classes, these base classes should have no fields and methods with the same names otherwise the compilation will fail.

Access modifiers of fields and methods are inherited from base classes. To override methods of base classes, one should explicitly define that function again with the same access modifiers and argument lists in the derived class.

```
class Base {
      public void foo(int a, int b) {...}
}
class Derived extends Base {
```

```
        //Overriding
        public void foo(int a, int b) {...}
    }
```

All of the classes inherits from the class ‘Object’.

6.6 Type coversion

Type conversion between classes is not supported as Lava doesn’t have its runtime.

7 Miscellanies

7.1 Lambda expression

Lava supports Lamda expressions like Java 1.8 does.

For example, there is an array of User instances where the first field of a User is name and the second is age.

```
User[] users = new User[] {new User(“Tom”, 22),
                           new User(“Mike”, 25),
                           new User(“Lily”, 18)};
```

Arrays.sort() function will accept an array and a lambda expression, and sort the array, using this lambda function to decide the relative order of two elements in it.

```
Arrays.sort(users, (User a, User b) -> a.age - b.age);
```

Essentially what the lambda function does is that it takes 2 input elements at a time, a and b, and compare them on the field of age. The return value of this compare function will be the age of User a minus that of User b, which will be treated by the sorting function as the relative precedence of the two users.

After the sorting the user array will be in ascending order on the field of age.

The syntax of a lambda expression is as follows:

```
(argument1, argument2, ...) -> a single-line expression
```

or

```
(argument1, argument2, ...) -> {
     // multiple statements
}
```

A lambda expression is comprised of 3 parts:

1. An argument list wrapped in brackets, where arguments are separated by commas:
   ```
   (int x, int y)
   ```
2. An arrow token:
   ```
   ->
   ```
3. The function body which can be a single expression or a statement block:
   ```
   x+y
   ```
   or
   ```
   {
       print("Summing integers\n");
       return x+y;
   }
   ```
   If the body is a single expression, it's evaluated and returned. If it's in the block form, the body will be executed like the body of a function and the `return` keyword will end the execution and give the result back to the caller of this lambda function. Every path in the control flow must have a return statement or throw an exception. Otherwise the syntax is illegal and the program cannot pass the compiler.

Note that the lambda expression can have nothing as argument input and nothing as returned result.

```
() -> {print("Nothing here\n")};
```

In the lambda expression, the program has no knowledge about variables outside the expression itself. In other words, it has no access to variables in the caller of the lambda expression. The program only knows whatever is passed to it in the argument list.

7.2 Generic

Lava supports the use of generics, which enables the programmer to use the class itself as parameters in the definition of classes and methods.

The use of generics brings two main benefits to programmers.

1. Methods and classes are more flexible as one method can work for multiple input and output types.
2. Casts can be replaced by using generics. The runtime class check are moved to compile time so the programmers are notified of type errors in an early stage, where they are more easily fixed than type errors in runtime.

Normally type parameters are single capital letters as it's easy to distinguish them from normal class names.

7.2.1 Generic classes

A generic class is defined with the generic class identifiers stated in angle brackets. A defined class identifier is treated as one normal known class names in the code. More than one generic types can be defined at one time. The generic types are declared in one pair of angle brackets, separated by commas, following the class declaration.

```
class ClassName <GenericType1, GenericType2, ...>{
      // The generic types can be used in fields
      GenericType1 foo;

      // And the types of arguments or return type
      public GenericType2 doSomething(GenericType1 arg){

      }
}
```

For example,

```
class Basket<A, B>{
      /*
      The capital letter A stands for an actual type, which is not
      specified until an object of Basket is declared.
      */
      A a;
      B b;

      // The declared generic types can be used as arguments
      public void set(A a){
            this.a=a;
      }
      // And return type
      public A getA(){
            return this.a;
      }
}
```

In the example, as the declaration goes in the angle brackets, Bracket requires two generic types, which can be of any type but primitives. Once declared, the Basket class will know what the generic types A and B stand for the rest of the code. For instance, in the following example, this new Basket instance will replace the generic type A by String and B by Integer.

```
Basket<String, Integer> b = new Basket<String, Integer>();
b.setA("A string"); // This will work
```

```
b.setA(new Integer(20)); // This is illegal as b wants String for
a
String s=b.getA();

// It's illegal if the classes don't match
// Basket<String, String> c = new Basket<String, Integer>();
Basket<Integer, String> d = new Basket<Integer, String>();
d.setA(new Integer(20));
```

7.2.2 Generic methods

Rather than declaring the generic type in the scope of a class, Lava supports generic declarations only for a method, which means the declared generic types only work in the scope of the specific method.

The generic declaration is put in a pair of angle brackets before the return type of the method. A generic method follows the syntax below:

```
access-modifier <generic-types> return-type function-identifier
(parameters) {
      //function body
}
```

For example,

```
class Apple{
      String getName(){
            return "an apple";
      }
}
class Orange{
      public String getName(){
            return "an orange";
      }
}
class Alien{
}
class Person{
      public <T> void eat(T t){
            print("I am eating a ");
            print(t.getName());
      }
}
```

In the above example, the method `eat(T t)` accepts an instance of any type. The way of invoking this method is as follows. The real type of the generic should be declared explicitly in the angle brackets preceding the function call.

```
Person p=new Person();
p.<Apple>eat(new Apple());
p.<Orange>eat(new Orange());
//p.<Alien>eat(new Alien());
// This is illegal as Alien type doesn't have method getName()
```

8 Built-in functions

Built-in functions in Lava are implemented by invoking the C standard library.

8.1 print

The print function takes an argument in int, float, char and string. This function will output to the stdout.

```
print(3);
print(3.57);
print("aaaaa\n");
print('a');
```

8.2 println

The println function takes the same type of argument as print, but it will append a newline character at the end of the input. For example, the following expression:

```
println(3);
```

is equivalent to

```
print(3);
print("\n");
```