W4115 Programming Languages and Translators - Spring 2017

Project Proposal

Lava

Yet Another Dialect of Java on JVM

Team members

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

Motivation

Java 8 was released in March 2014 and people were so excited to see that Java made a big step in terms of language features, probably the most attractive among which was the lambda expression.

We are greatly inspired by this new version of Java and we decide to integrate more interesting features into Java to make it more flexible. Our main expectation is to integrate some elements of functional programming with Java and see what we can have.

The Java island is almost entirely of volcanic origin, and contains 45 active volcanos. Lava is the essence of volcano, and therefore the essence of Java, which will also be the name of our language.

Target

We aim to build a language based on Java that has the basic functionalities and syntax of Java. Lava, like Java, will be compiled to bytecode and run on JVM. And based on that, we will make improvements to existing language features, and then learn from C# and python to introduce a few more keywords and some functional programming functionality.

We will first implement some interesting features in Java like generics and lambda expression. We also hope to make improvements to which points have been criticized for long. For example, the type erasure in generics makes List<Apple> and List<Fruit> two irrelevant classes although Apple directly inherits Fruit. So we must define List<? extends Fruit> rather than List<Apple> to pass the compiler check. Because we don't need to maintain the backward compatibility we can safely abandon type erasure to make *List<Fruit> fruits = new ArrayList<Apple>()* work. This is one main example of some counter-intuitive syntax in Java that we expect to improve on.

```
public class Fruit {...omitted...}
public class Apple extends Fruit {...omitted...}
List<? extends Fruit> fruits = new ArrayList<Apple>();
```

We also want to import some interesting and powerful features from other languages, C# and python being the perfect examples. C# learned from Java a lot and then made quite some meaningful improvements in its functionality, for example, the dynamic keyword. We will give more detailed example in the Language Description.

We will, at the same time, get rid of many features of Java. We want to focus on making the new features that we desire to have. And some features that mean little to us, like inner classes and switch statements, will be intentionally omitted and will not be implemented. Also there will be some undesired features that we intentionally discard, for example, type erasure for generics.

Finally, since we aim at using JVM as the running environment, we expect to integrate the powerful Java libraries to our Lava programs. For example, in our Lava program, programmers should be able to import java.util and java.io, just like Groovy and Scala programmers do all the time. This will greatly enhance the power of our Lava language.

Language Description

Types

- int
- float
- char
- boolean
- String
- Array
- List

Operators

- Arithmetic
  - +,-,*,/
- Conditional
  - ==, !=, >=, <=, >, <
- Logical
  - && (AND), || (OR), ! (NOT)
- Operational
  - = (assignment)

Flow Statements

- if
- for
- while
- return

Scope

Scopes are defined by brackets, which are necessary even if there is only one line in if, for or while statements. Indentation and line break does not matter to scopes.

Miscellaneous
Comments are supported, including one line comment(//) and block comment (“/**/”).
Statements must be ended by semi-colons.

Feature
1, Polymorphism (with Reflection)
2, Dynamic keyword
3, Property lambda expressions
4, Generic programming
5, Operator overload and function override

Class
Included: constructor, destructor, public, private, field, method

Excluded: inheritance, interface, abstract, final and other high-level features

New: Create an instance by the name of class

```
        Apple a = new Class(“apple”)(); /* equivalent to Apple a = new
Apple();*/
```

As we decided to use reflection and dynamic to achieve polymorphism we consider the class hierarchy is not necessary in our approach. Therefore, class inheritance is ruled out in our design. What’s more in order to concentrate on what we really want to do, some high-level features are not in our plan.

In addition, we introduce a syntactic sugar of reflection to allow create an instance by class name, which will make our dynamic keyword easier to use.

Lambda Expressions
Basically it is just like lambda property added in Java SE8. Lambda expressions enable you to treat functionality as method argument, or code as data. For example, you can pass functionality as an argument to another method, such as what action should be taken when someone clicks a button.

For example, given the code below,
```
User users = new User[] {new User(“Tom”, 22),
                         new User(“Mike”, 25),
                         new User(“Lily”, 18)};
```

we can replace the code below,

```
Arrays.sort(users, new Comparator<User>() {
        public int compare(User a, User b) {
                return a.age - b.age;
        }
});
```

with this line of code utilizing lambda expression,

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

Generics
All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type.

Here is the example for generics,

```
public class GenericMethodTest {
   public static < E > void print( E five ) {
       System.out.println(five);
   }

   public static void main(String args[]) {
      Integer five_int = 5;
      Double five_double = 5.0;
      String five_string = “five”;
      print(five_int);      // pass an Integer
      print(five_double);   // pass a Double
      print(five_string);   // pass a String
   }
}
```

Integrate Java Libraries
Lava should also be able to integrate the powerful Java libraries. For example, in our Lava program, programmers should be able to import `java.util` and `java.io`, just like Groovy and Scala programmers do all the time.

```
import java.util.*;
import java.io.*;
```

Reflection
Java Reflection is a process of examining or modifying the runtime behavior of a class at run time.The reflection feature is a very powerful feature which can distinguish it from C++. As a

static programming language, it supports to create, use an instance of class dynamically without knowing the exact name before running it. However, the grammar of reflection in Java is a little bit complicated. So we borrowed the keyword “dynamic” to implement the most functionalities in reflection.
Because we plan to use keyword “dynamic” to cover the most part of the Java Reflection features, we are going to implement only three functions about reflection.

1. `Object.getName() // get the class name of an object`
2. `Object.getDeclaredFields() // get all of the fields in a class`
3. `Object.getDeclaredMethods() // get all of the methods in a class`

Keyword dynamic
The dynamic keyword is new to C# 4.0, and is used to tell the compiler that a variable's type can change or that it is not known until runtime. Think of it as being able to interact with an Object without having to cast it. To some extent, we can consider it as syntactic sugars of reflection. The keyword “dynamic” can be used in the following two ways.

1. As a type of variable

```
dynamic cust = GetCustomer();
dynamic cust2 = new Class(“customer”)();
//equivalent to:
//Class c = Class.forName(“customer”);
//Object cust2 = c.newInstance();

//equivalent to:
//Method method = cust.getClass().getMethod("Process", null);
//cust.invoke(method, null);
cust.Process(); // works as expected

cust.MissingMethod(); // No method found!
```

The creation and invoking with dynamic variable have the same effect with a piece of Java reflection code.

2. As a type in generic

The dynamic keyword could also be used as a type when using generic. In this case, all of the variable regards the type of the generic will be declared as dynamic variable.

```
ArrayList<dynamic>  foo; /*like the ArrayList<object>, you can put
everything in*/
foo.add(new Game());
foo.add(new Piano());
foo.add(new Card());
foo.foreach(x=>x.play()); /*but allow to invoke any function or
without casting to a specific type*/
```

Source code sample

```
import java.util.*

// Basic class construction
public class Game{
    private String game_name;
    public Game(String _game_name) {
        game_name = _game_name;
    }
    public void play() {
        System.out.println("Start playing " + game_name );
    }
}

// Simple case matching
public class Instrument {
    private String instrument_name;
    public Instrument(String _instrument) {
        instrument_name = _instrument_name;
    }
    public void play() {
        if (instrument_name.equals("piano")){
            System.out.println("Do~Re~Mi~");
        }else if (instrument_name.equals("drum")) {
            for (int i=0;i<10;++i){
                System.out.println("Duang~");
            }
        }else{
            System.out.println("Unknown Instrument");
        }
    }
}

// Recursive
public class Joke{
    private void make_jokes(int n){
        if (n>0)   {
            System.out.println("Hahaha~");
            make_jokes(n-1);
        }
    }
    public void play() {
        make_jokes(10);
    }
}

/* This is the main class */
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
        // We can create a list of anything, not implicitly casting
to Object
        ArrayList<dynamic> array;
        array.add(new class("Instrument")("Piano"));
        array.add(new Game("Chess"));
        array.add(new Joke());

        List<dynamic> list;
        list.add(new Instrument("Drum"));
        list.add(new class("Game")("Bridge")); // Game("Bridge")
        list.add(new class("Joke")()); // Joke()

        // We can assign anything to this dynamic object
        dynamic container;
        container = array;
        container.foreach(item->item.play());

        container = list;
        container.foreach(item->item.play());
    }
}
```