Primer

* Kevin Jack Hanna(legajo:52034)
* Fernando Bejarano(legajo: 52043)
* Federico Bond(legajo: 52247)

November 30, 2014

Contents

Introduction

This document describes the design of the Primer language. Primer is an imperative language for the Java Virtual Machine (from now on, the JVM). It has a clean syntax inspired by languages like Scala and Kotlin, but does not support object-oriented programming.

Design Considerations

The initial AST class design was heavily inspired by the JRuby project. The rest of the class structure grew organically while we were developing the project.

Grammar in BNF (Backus Naur Form)

Statement ::= If
| While
| Call
| Function
| Return
| CONTINUE
| BREAK
| Declaration
| Assignment
| AssignmentAdd
| AssignmentSubtract
| AssignmentMultiply
| AssignmentDivide ;

Expression ::= Expression PLUS Expression
| Expression MINUS Expression
| Expression TIMES Expression
| Expression DIVIDE Expression
| Expression AND Expression | Expression OR Expression
| Expression LESS_THAN Expression
| Expression LESS_EQUAL_THAN Expression | Expression1 GREATER_THAN Expression | Expression GREATER_EQUAL_THAN Expression
| Expression EQUAL Expression
| Expression NOT_EQUAL Expression | MINUS Expression %prec UMINUS | LEFT_PAREN Expression RIGHT_PAREN
| Call
| Variable
| Literal

| Expression MODULUS Expression
;

If ::= IF Expression OPEN_BRACKET StatementList CLOSE_BRACKET |
IF Expression OPEN_BRACKET StatementList CLOSE_BRACKET ELSE
OPEN_BRACKET StatementList CLOSE_BRACKET ;

While ::= WHILE Expression OPEN_BRACKET StatementList CLOSE_BRACKET

Call ::= IDENTIFIER LEFT_PAREN Args RIGHT_PAREN

Args ::= Expression COMMA Args
| Expression
|
;

Literal ::= INT
| STRING
| TRUE
| FALSE
| NIL
;

Function ::= DEF IDENTIFIER LEFT_PAREN FunctionArgs RIGHT_PAREN
OPEN_BRACKET StatementList CLOSE_BRACKET

FunctionArgs ::= IDENTIFIER COMMA FunctionArgs
| IDENTIFIER
|
;

Return ::= RETURN Expression
;

Assignment ::= IDENTIFIER ASSIGN Expression

AssignmentAdd ::= IDENTIFIER ASSIGN_PLUS Expression

AssignmentSubtract ::= IDENTIFIER ASSIGN_MINUS Expression

AssignmentMultiply ::= IDENTIFIER ASSIGN_TIMES Expression

AssignmentDivide ::= IDENTIFIER ASSIGN_DIVIDE Expression

Declaration ::= VAR IDENTIFIER ASSIGN Expression

Variable ::= IDENTIFIER

Difficulties encountered

One of the first difficulties we found was configuring everything correctly. The JFlex and Java CUP projects are quite old and while they have received constant

updates, the documentation is lacking in several areas, with many obsolete code examples.

The other main obstacle was coming up with a suitable workflow for adding new language features. Writing integration tests early on made it possible to work on the codebase with confidence that we would not break any previous functionality.

Preliminary research

After some research we found out that the Jasmin library suggested in class was not the best choice for implementing the language, since it was no longer maintained and lacked useful abstraction for generating the required bytecode from the Abstract Syntax Tree. We decided to use the widely known ASM library, which proved to be a valuable inclusion in the project.

Supporting libraries

The project was very conservative in its use of libraries. For generating the lexer and parser code we used JFlex and Java CUP.

To generate bytecode from the AST returned by the parser, we chose the ASM library.

We also relied on Google's Guava libraries for some helper methods.

Development

Compilation Pipeline

The compilation process begins by reading the script file. We create a `Script` instance that represents a Primer script. From there, we build a Lexer that will emit a stream of tokens to be consumed by the Parser. The parser comsumes these tokens and returns an AST root node (which is always a `BlockNode`). Also, if available, any optimizations are applied by recursively walking through the AST. The final step is to visit the AST and write the bytecode for each instruction.

Notes on the workflow

We found a very useful workflow for adding the bytecode implementations for each node. We would write a Java implementation of the syntax construct we

were implementing and look at the generated bytecode (the IntelliJ IDEA IDE has a very useful bytecode view that is accessed from the View menu). The one thing that we had to be careful for was to keep the JVM from optimizing constant expressions. In most cases, assigning the expression components to variables is enough to prevent this. So, instead of writing:

```
if (true && false) {
    ...
}
```

We would write:

```
boolean a = true;
boolean b = false;
if (a && b) {
    ...
}
```

The Java bytecoe instruction listings found in Wikipedia also proved very useful, since it details the contents of the stack before and after every instruction.

Optimization passes

We designed an optimization pass that traverses the AST and simplifies nodes based on compile-time folding of constant expressions. It also handles `if` statements whose condition can be determined to be always true. The optimization pass also discards any `else` clauses and replaces the `if` node with a block. Thus, the compiled code will not contain any unnecessary jumps.

The code for this optimization can be found in the `ConstantFoldingVisitor` class.

Benchmarking

We compared Primer code against equivalent Java and Ruby implementations of the fibonacci function. The results are listed in the table below.

Java Implementation

```
public class Fibonacci {

  private static int fibonacci(int n) {
```

```
        if (n < 2) {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

    public static void main(String[] args) {
        System.out.println(fibonacci(30));
    }
}
```

Ruby Implementation

```
def fibonacci(n)
  if n < 2
    return n
  end
  return fibonacci(n - 1) + fibonacci(n - 2)
end

puts fibonacci(32)
```

Primer Implementation

```
def fibonacci(n) {
   if n < 2 {
     return n
   }
   return fibonacci(n - 1) + fibonacci(n - 2)
}

println(fibonacci(32))
```

Results

Language	Average run time*	Tokens
Java	0.17 s	60
Ruby	0.69 s	32
Primer**	0.39 s	36

* Average of ten invocations using the `time` UNIX command. Tested on a MacBook Pro 2.4 GHz (mid 2010)

** Includes compilation time.

We can see from the above table that Primer achieves a performance between that of Java and Ruby, but with a onciseness (measured in number of tokens) much closer to the last one.

Since Primer compiles to JVM bytecode, there is ample room for optimization, and a Sufficiently Smart Compiler could generate code that runs as fast as equivalent code written in Java.

Future extensions

Thorough the project we kept a `TODO.md` file with ideas for things we could add to the language. Some of them landed on the final release, but others demanded too much time and are listed below as future extensions.

Lazy boolean operators

Contrary to the behavior of many languages, primer does not support lazy boolean operators, and has to evaluate both sides of each expression every time. To fix it, we need to revise the bytecode generator so that it outputs correct jump instructions for these cases.

Estimated time to implement: one day.

Java interoperability

The main obstacle for achieving Java interoperability was the ability to handle and express the full range of types in Java. Projects such as Jython or JRuby managed to run Java code from a dynamically typed context, but this requires some non-trivial type handling.

Estimated time to implement: two weeks.

Literals for commonly used objects

While calling constructors such as `list()` is an acceptable solution, most modern languages include some syntax for describing lists, maps, ranges and regexes. Example notation could be:

```
var list = [1, 2, 3]
var map = { "foo": 1, "bar": 2 }
var range = 1..10
var regex = /[^abc]*/
```

Estimated time to implement: 3 days.

List and map indexing syntax

Currently, we use the `list_get` and `hash_get` functions for accessing list and map elements. A more elegant solution involves the use of `[]` syntax for performing this task.

```
list
# => [1, 2, 3]
list[1]
# => 2

hash
# => { "foo": 1, "bar": 2 }
hash["foo"]
# => 1
```

Estimated time to implement: 1 day.

Lambda expressions

Being able to manipulate functions as regular objects has proved to be a powerful programming technique. Many modern languages have adopted some of support for this kind of programming. Implementing such a system is anything from trivial, especially in a platform like the JVM. Fortunately there has been a surge on documentation on this topic since the inclusion of lambdas in Java 8.

```
var lambda = (n) -> 2 * n
var list = [1, 2, 3]
list_map(list, lambda)
# => [4, 5, 6]
```

Estimated time to implement: three weeks.

Conclusion

We found out that implementing a programming language on top of the JVM is both approachable and fun. The supporting libraries are mature and provide useful abstractions for simplifying the work.

We learned about JVM internals that are rarely exposed most Java programmer and gained a better understanding of the Java platform.