Teaching Programming Languages: Java as a Metalanguage

Ariel Ortiz
Computer Science Department
ITESM Campus Estado de México
Atizapán, Estado de México, MEXICO

Abstract

This paper describes the experiences learned from teaching an undergraduate Programming Languages courses using an interpreter building approach in which Java was the implementation language.

Keywords

Programming languages, interpreter, Java, Scheme, functional programming.

1 Introduction

The Java programming language has become lately one of the most important emerging tools for software development. In spite of this, we should not underestimate the importance of learning and mastering other programming languages and paradigms that might be superior to Java in expressiveness when developing applications for certain problem domains. For instance, LISP is easier to use than Java when processing symbols and lists. The same applies to Prolog when using unification and backtracking or to SQL when requiring a data definition and manipulation language. But even though the Java platform may seem unsuitable for some specific areas, it has the great advantage of being able to incorporate other technologies. This can be seen, for example, with the JDBC API, which allows us to embed SQL instructions inside a Java program.

Programming languages such as Scheme, Prolog, Logo or even other object oriented languages such as Smalltalk or Eiffel may be taught to Computer Science students using a translator-based approach using Java as a *metalanguage* (a language used for implementing language translators). Other languages, such as Pascal [7] and Scheme [1,6], have been used for this purpose as well. However, the author firmly believes that the Java language/platform offers considerable advantages over other languages in this respect. Specifically, Java has an excellent integration with modern parser building tools. Additionally, Java's rich API does not only include important interpreter data structures (for example: stacks and hash tables), but also opens the possibility of smoothly integrating interpreters with some very interesting facilities (for example: graphics, threads, distributed objects, database connectivity, etc.). If you add Java's commonly acknowledged strengths (object oriented, simple, dynamic, secure, automatic memory management, platform independent), you end up with a powerful interpreter building tool.

The remainder of this paper describes the experience on teaching undergraduate students how to build an interpreter for a small functional language using Java as the implementation language.

2 The Esquemita programming language

During the fall semester of 2000, the author started using a translator-based approach using Java to teach the undergraduate Programming Languages course (a mandatory third year course) at the Institute of Technology and Higher Education of Monterrey, Campus Estado de Mexico (ITESM CEM). Specifically, a step-by-step development of a working interpreter of the language *Esquemita* was presented during class. When students take this course, they already have a reasonable experience with regular expressions, free context grammars, and object-oriented

software development using Java and the Unified Modeling Language (UML).

Before starting any work on the interpreter, a general overview of the theory of language translation was covered. Each translation phase was studied at a conceptual level. Afterwards, the JavaCC [9] (*Java Compiler Compiler*) parser-building tool was presented together with a few simple yet illustrative toy interpreters.

2.1 General overview of the language

Esquemita is a small yet representative subset of the Scheme [8] programming language. A tradeoff between language completeness and simplicity had to be made. With this in mind, the Esquemita interpreter was designed to support only the following features:

- Data types: symbols, lists (pairs and the null list) and procedures. For boolean values, the empty list represents false, everything else is considered true.
- Primitive procedures: *car*, *cdr*, *cons* and *null?*
- Special forms: *quote*, *define*, *if* and *lambda*
- Procedure application (not properly tail recursive)

It should be noted that Esquemita is a purely functional subset of Scheme. Despite the fact that this is a very small language, it does allow students to obtain a general impression of what the real Scheme language is like. Furthermore, extending the interpreter's simple framework may result in some interesting student projects.

2.2 Interpreter design and implementation

A *read-eval-print-loop* (REPL) is the typical way of using interpreters, and Esquemita was no exception. This simple execution model is shown in figure 1. As can be observed, runtime errors might happen during the evaluation of program expressions. To simplify error management, a special unchecked Esquemita exception class was defined. This exception is only caught at highest level of the read-eval-print-loop.

The overall interpreter design may be briefly described as a Java class hierarchy (see figure 2) in which the main classes represent some kind of a specialized Esquemita data type. All instances of these classes respond to a method called `eval`. This method's contract is to return the result of evaluating the Esquemita datum being represented. During the read-eval-print-loop, the interpreter only deals with generic `SType` objects; polymorphic calls are used to "eval" each of these objects. Overridden `toString` methods are then called to print the external representation of every evaluated object.

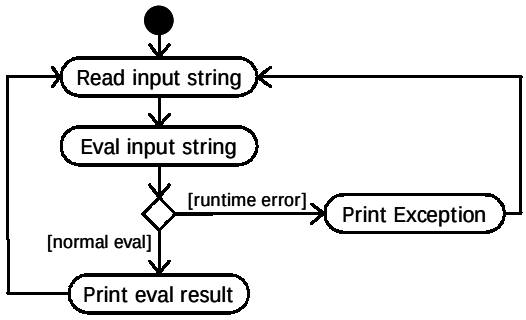

Figure 1: UML Activity Diagram of a *Read-Eval-Print-Loop*

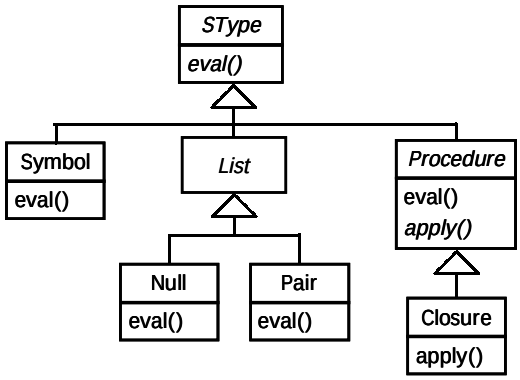

Figure 2: UML Class Diagram of Esquemita Types

The `eval` methods for the `Null` and `Procedure` classes are extremely simple: they merely evaluate to themselves. Speaking in Java terms, they just return `this`.

The real work of the interpreter occurs in the `eval` method of the `Pair` class. Special forms (*quote*, *define*, *if* and *lambda*) and procedure applications are taken care of here. This is because special forms and procedure calls are always found syntactically inside a sequence of one or more *pairs* (a non-null list).

Instances of classes that represent Esquemita functions (user-defined functions and primitive operations) must extend the `Procedure` class. In this case, the `apply` method is the key for carrying out function application (procedure execution). The overridden `apply` methods, defined for each of the `Procedure` subclasses, contain the actual instructions for each of the Esquemita functions. A global environment object is used to bind the variable names with their expected primitive operation objects. This same global environment is used to place new bindings whenever the "*define*" special form is used. The "*lambda*" special form, on the other hand, creates a *lexical closure* (user-defined function) that requires cloning the global environment and extending this new copy with its local lexical variable bindings. These environments are very easily implemented using the `java.util.HashMap` object.

Esquemita variables are represented internally using instances of the `Symbol` class; calling the corresponding `eval` method is how variable access is done. As should be expected, the entire job consists in a simple hash lookup.

2.3 Building the interpreter

The steps taken to present students the interpreter design and implementation are summarized bellow:

- The lexical analyzer was constructed from our language's token specification.
- Similarly, the syntax analyzer was developed from our language's grammar specification. For both lexical and syntax analyzers, the JavaCC tool was used. Because Scheme's lexical and syntax structure is so simple, this was done pretty quickly. Students tested these two parts of the interpreter using a driver program that read the source programs from the standard input. Basically, the syntax analyzer translates a grammatically valid program to an Esquemita list representation of a syntax tree.
- Using UML activity, class, and package diagrams [4], the general structure of the interpreter was explained to the students.
- The implementation of the classes explained in the previous point was carried on. This was, of course, the project's most time consuming activity. In order to incrementally test the interpreter, a simple graphical user interface was designed using Java's Swing API in order to implement the read-eval-print-loop.

It is important to note that most students were actually typing, compiling, and testing incrementally their interpreter using their own mobile computer equipment during class hours.

2.4 Learning the Esquemita language

The students didn't have any previous experience with Scheme or LISP before this course. The Esquemita language was partially presented to them during the interpreter development, explaining only what was strictly required at any time. When the interpreter obtained its final form, a more comprehensive presentation of our Scheme subset was offered. It was fascinating to see the students' faces when they saw running their first Esquemita programs over an interpreter they had been painstakingly building for quite a while.

Because the Esquemita language is so small, it allowed us to extend it through some interesting experiments. For example, using only Esquemita's built in types, how can you represent positive integer numbers and operations? One possible solution to this problem is briefly summarized in table 1. These kinds of problems are an excellent way of encouraging students to reason specifically about data representations and abstraction in general.

2.5 Embedding Esquemita code in Java

Esquemita may not only be used in its own read-eval-print-loop, it can also be embedded in ordinary Java code (just like what JDBC does to SQL). The following Java snippet shows how this can be done:

```
SType result =
   Scheme.eval("(car '(a b c))");
String s = result.toString();
// s = "a"
```

Table 1: Implementing Positive Integer Numbers and Operations

Concept	*Esquemita implementation*
0	`'()`
1	`'(())`
2	`'(()())`
n	`'(() ... ())`
zero?	`(define zero? null?)`

add1	`(define add1`
	`  (lambda (x)`
	`  (cons '() x)))`

sub1	`(define sub1`
	`  (lambda (x)`
	`  (cdr x)))`

=	`(define =`
	`  (lambda (a b)`
	`  (if (zero? a)`
	`  (if (zero? b) 'true '())`
	`  (if (zero? b) '() (= (sub1 a) (sub1 b))))))`

+	`(define +`
	`  (lambda (a b)`
	`  (if (zero? a) b (add1 (+ (sub1 a) b)))))`

*	`(define *`
	`  (lambda (a b)`
	`  (if (zero? a) a (+ b (* (sub1 a) b)))))`

The Esquemita code can appear inside a Java string, but it can also be read from a file. The embedding facility allows us to build programs that combine whatever features we care to use from both the Esquemita and Java languages.

3 Building interpreters for other languages

Because the Esquemita interpreter was mostly instructor-guided, an extra-class project was assigned to all students in order to make sure that they understood how to build an interpreter by themselves. The project consisted in building a Logo interpreter. As a hosting platform, Java worked fairly well for Logo's turtle graphics. Logo has some common elements with Scheme, yet in some respects it is completely different. Particularly, Logo is an imperative language, not a functional one. Consequently, students had to fully understand, at the implementation level, the differences between these two computational models.

4 Related work

Using Java to build interpreters and compilers has been common since the inception of the Java programming language. Appel [3] explains how to build language compilers (including those for functional languages) using Java. This approach is fine for a compiler design course, but not for a programming languages course. This is because, as experience shows us, writing a virtual machine (interpreter) for a language is generally easier and quicker than writing a compiler that has to produce a different language target code.

Kawa [5] and Silk [2] are some well known examples of Scheme translators programmed in Java. Esquemita is unlike these systems in that it was designed with some very specific instructional objectives in mind. Esquemita is a sufficiently small and simple that it can be explained, built and even modified by undergraduate students in about four class weeks. This is something that can't be done with these other systems; their design goals are just different.

5 Lessons learned

Traditionally, we had taught the programming languages course through a survey of the most important programming paradigms. A representative language of each paradigm was studied in turn. This approach, although partially successful, left many students with the feeling that unconventional programming paradigms are something esoteric and useless in practice.

By contrast, building interpreters in Java gave students the opportunity to build real non-trivial working object-oriented systems. They applied their Java knowledge to learn new languages and paradigms, and at the same time they practiced advanced Java programming techniques. The interpreter is an excellent case study that demonstrates how a software system can significantly incorporate class hierarchies, abstract classes, polymorphism, exception handling, and GUIs.

There is anecdotic evidence that students particularly enjoyed the course and that their Java programming skills increased significantly. Several students held that the use of their mobile computer equipment during class was an essential feature of the course's success.

The course evaluation, in which students freely and anonymously express their opinions about the course once it has completed, shows some interesting results. 90% of the students completed this evaluation. In a scale of 1 to 7, in which 1 means totally agree and 7 totally disagree, the results of three of the most relevant questions are as follows:

1. The instructor organizes and offers the course in a way that encourages learning: Average = 1.148, Deviation = 0.456
2. The instructor's performance in this course is excellent: Average = 1.037, Deviation = 0.192
3. Globally speaking, this course has offered a significant added value for my professional formation: Average = 1.074, Deviation = 0.267

As a comparison, the average of all faculty members in the same Computer Science department is around 1.8 for each of the three questions above. In previous semesters, the author obtained evaluations around 1.4 for the same course. The author believes that this reflects a near excellent student satisfaction for the new proposed approach.

As a future direction, the author is planning on teaching the Prolog language using this same interpreter building approach with Java.

The Esquemita source code is available for download at the author's Web page:

http://webdia.cem.itesm.mx/ac/aortiz/esquemita.html

6 References

[1] Abelson, H., and Sussman, G. *Structure and Interpretation of Computer Programs.* Second edition. MIT Press, 1996.

[2] Anderson, K., Hickey, T. and Norvig, P. *SILK A Java-based dialect of Scheme.* Available WWW: http://tigereye.cs.brandeis.edu/silk/silkweb/README.html

[3] Appel, A. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[4] Booch, G., Rumbaugh, J. and Jacobson, I. *The Unified Modeling Language User Guide.* Addison Wesley, 1998.

[5] Bothner, P. *The Kawa Scheme System.* Available WWW: http://www.gnu.org/software/kawa/

[6] Friedman, D., Wand, M. and Haynes, C. *Essentials of Programming Languages.* MIT Press, 1992.

[7] Kamin, S. *Programming Languages An Interpreter-Based Approach.* Addison Wesley, 1990.

[8] Kelsey, R., Clinger, W. and Rees, J. eds. *Revised^5 Report on the Algorithmic Language Scheme.* 1998.

[9] Metamata. *JavaCC: The Java Parser Generator.* http://www.webgain.com/products/metamata/java_doc.html (downloaded April, 2001)