Language Design and Implementation using Ruby and the Interpreter Pattern

Ariel Ortiz
Information Technology Department
Tecnológico de Monterrey, Campus Estado de México
Atizapán de Zaragoza, Estado de México, Mexico. 52926
ariel.ortiz@itesm.mx

ABSTRACT
In this paper, the S-expression Interpreter Framework (SIF) is presented as a tool for teaching language design and implementation. The SIF is based on the interpreter design pattern and is written in the Ruby programming language. Its core is quite small, but it can be easily extended by adding primitive procedures and special forms. The SIF can be used to demonstrate advanced language concepts (variable scopes, continuations, etc.) as well as different programming styles (functional, imperative, and object oriented).

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *interpreters, run-time environments.*

General Terms
Design, Languages.

Keywords
Interpreter, Ruby, S-expression, Design Patterns.

1. INTRODUCTION
Teaching programming language design concepts through the construction of interpreters has been an appealing approach used by others in the past [3][6]. Following this same direction, the author developed the *S-expression Interpreter Framework* (SIF). This framework has been effectively used as a teaching aid at the undergraduate Programming Languages course at the Tecnológico de Monterrey, Campus Estado de México. The framework is written in Ruby in order to have students learn an increasingly popular object-oriented dynamic language, while learning language design and implementations concepts at the same time.

The following subsections deal with some issues that are required in order to understand the design and inner workings of the SIF. The larger part of this paper, sections 2 and 3, is basically a tutorial on how to use and extend the framework so as to be able to explore different language concepts and programming styles. Section 4 describes briefly how the SIF has been used in class. The conclusions are found in section 5.

SIGCSE'08, March 12-15, 2008, Portland, Oregon, USA.

1.1 S-Expressions
S-expressions (symbolic expressions) are a parenthesized prefix notation, mainly used in the Lisp family languages. They were selected as the framework's source language notation because they offer important advantages:

- Both code and data can be easily represented. Adding a new language construct is pretty straightforward.
- The number of program elements is minimal, thus a parser is relatively easy to write.
- The arity of operators doesn't need to be fixed, so, for example, the + operator can be defined to accept any number of arguments.
- There are no problems regarding operator precedence or associativity, because every operator has its own pair of parentheses.

It's definitely easier to build an S-expression interpreter using an S-expression language like Common Lisp or Scheme. Indeed, plenty examples have been documented previously [1][2][5]. In most of these cases, a "meta-circular" approach is used, meaning that the existing facilities of the implementation language are directly applied to the source code being interpreted. Although this approach is very interesting as a way of demonstrating the power and expressiveness of the implementation language, the author has witnessed that some students get the impression that we are actually "cheating". This is because a lot of the source language features are typically mapped directly to the same features in the implementation language. The SIF, having Ruby as its implementation language, skips this common circular perception peril. Ruby offers the power and flexibility of a dynamic language, yet maintains a healthy distance from the source language being interpreted.

1.2 The Interpreter Pattern
The interpreter pattern is one of the classical Gang of Four (GoF) behavioral patterns [4]. A program is portrayed as a tree in which every construct or element in the source program is represented by a specific kind of node. Every node responds to a special operation called *interpret*, which is responsible for performing the expected behavior associated with the corresponding language element. The *interpret* operation receives a *context* as a parameter,

which is basically a symbol table where variable bindings can be established and queried during runtime.

The framework's job begins by reading a source program represented as a string. The Ruby regular expression API is used to scan the input, and a hand written recursive descent parser performs the syntactic analysis that transforms the S-expressions into the equivalent Ruby data values. These objects are then traversed in order to construct the tree required by the interpreter pattern.

1.3 The Ruby Language

Ruby is an interpreted, dynamically typed programming language. Its syntax borrows from Eiffel, Ada, and Perl, and it's fully object-oriented in the spirit of Smalltalk [7]. Several language features in Ruby simplify the construction of language interpreters, including:

- Built-in regular expressions, required to build a lexical analyzer or scanner.
- Garbage collection, which allows the interpreted languages to also benefit from automatic memory management.
- Built-in hashes can be used as symbol tables, while primitive symbol data values can be used as efficient hash keys.
- Open classes, which simplify using predefined Ruby classes to represent the interpreter's basic data types, since new methods can always be added to these classes if needed.
- First class continuations, which permit the implementation of special control-flow instructions such as arbitrary loop breaks and exception handlers.

The following subsubsections briefly describe a few Ruby language issues necessary to understand the code found in other sections of this paper.

1.3.1 Data Types

Everything in Ruby is an object, including: strings, numbers, arrays (lists), hashes (maps or dictionaries), symbols, procedures (closures), and even classes. Some examples (the # character denotes the start of a line comment):

```
'Hello Portland!'           # A string
42                          # A number
[1, [], 'A']                # An array
{ 'foo' => 5, 'bar' => 10 } # A hash
:foo                        # A symbol
Proc.new {|x, y| 2*(x+y)}   # A procedure
class MyClass; end          # A class
```

1.3.2 Naming Conventions

Local variables, method parameters, and method names should all start with a lowercase letter or with an underscore. Global variables are prefixed with a dollar sign ($), while instance variables begin with an "at" sign (@). Class variables start with two "at'" signs (@@). Finally, class names, module names, and constants should start with an uppercase letter [8].

1.3.3 Code Blocks

A code block is a "chunk" of code that can be associated with a method invocation, almost as if it was a parameter. That method can then invoke the block one or more times using Ruby's `yield` statement. The return value of a block is the result of its last expression. Code blocks are a very important and powerful feature of the Ruby language, and they are usually used to implement things like callbacks and iterators. The SIF implementation uses code blocks extensively.

Code blocks are enclosed within curly braces (or by the `do` and `end` reserved words), and they may include a comma-separated formal parameter list delimited by vertical bars (see the procedure object example in section 1.3.1). Although the code block would seem like an extra parameter, it's better to think of it as a coroutine that transfers control back and forth between itself and the method [8].

2. USING THE FRAMEWORK

The following subsections explain the basic usage of the SIF and demonstrate how to extend it.

2.1 Evaluation Rules

The core of the SIF is very simple. It only supports integers, symbols, lists, and procedures. These are evaluated as follows:

- A number evaluates to itself.
- A symbol is considered to be a variable reference, so it evaluates to the value that it's bound to. If the variable is unbound, a runtime error is raised.
- An empty list evaluates to itself.
- A non-empty list is considered a procedure application. The first element of the list must evaluate to a "callable" object. An object is callable if it contains a method named `call` (this is true also for standard Ruby `Proc` objects). A runtime error is raised if no callable object is found at the beginning of the list. The remainder of the list elements are evaluated and sent as arguments to the procedure being called. A runtime error is also raised if the number of actual and formal parameters doesn't match.

Non-empty lists may also be *special forms* with particular evaluation rules, but they must be explicitly defined. Section 2.4 explains how to do this. Additionally, there are no explicit boolean values. The SIF's convention is that an empty list is considered to be false, anything else is considered true.

2.2 Simple Usage

The `Interpreter.eval` class method is the heart of the SIF. This method receives a string and a context as inputs. The string should contain one or more valid S-expressions, while the context may be a hash object in which each key represents the name of a variable that's associated with some specific value. The hash keys are assumed to be Ruby `Symbol`s, and the hash itself should be mutable in order to allow the S-expressions to modify its contents if needed. This method parses the input string converting the S-expressions into their equivalent Ruby data values, and then proceeds with the evaluation using the interpreter pattern as described in section 1.2.

The following example shows how to use the SIF in its simplest form:

```
require 'sif'

my_context = {:foo => 5, :bar => 10}
result = Interpreter.eval('foo', my_context)
puts result.to_sexp
```

The first instruction loads the required framework code. The last instruction prints the result of the evaluation using the `to_sexp` method to obtain its S-expression textual representation. This example prints 5, because the expression to evaluate is only a

reference to variable `foo`, which is associated to the value 5 in the given context. Writing a simple REPL (Read-Eval-Print Loop) for the SIF is a pretty straightforward chore.

2.3 Defining Primitive Procedures

A procedure is considered to be "primitive" if it's written in Ruby. Continuing from the last example, the following code creates a primitive procedure and binds it to a variable in our previously defined context:

```
my_context[:max] = Proc.new do |args, ctx|
  (args[0] > args[1]) ? args[0] : args[1]
end
```

Primitive procedures for the SIF are usually defined as Ruby `Proc` objects. They receive two inputs: an array `args`, which contains the actual parameters sent to the procedure, and the current context `ctx`, which may be safely ignored most of the time (the current context in normally useful only when defining procedures like Lisp's `eval` or `apply`). The above code binds into `my_context` the variable `max` with a procedure that returns the larger of its two arguments (assuming they're comparable through the `>` operator). The `?` ternary operator used here works the same as in other C-family languages. The newly defined procedure can now be used as follows:

```
a_string = '(max (max foo bar) 7)'
result = Interpreter.eval(a_string, my_context)
puts result.to_sexp
```

This code prints 10.

The SIF provides a class called `Primitives` which provides a set of predefined primitive procedures (including operations for arithmetic, list processing, and type predicates) that can be readily used. The class method `Primitives.context` should be used to get a copy of the context containing the variable bindings to the primitive procedures. New variables bindings can be added afterwards, for example:

```
my_context = Primitives.context
my_context[:baz] = 20
```

New primitive procedures can be added to the `Primitives` class using the `Primitives.include` class method. The following code shows a possible implementation of an equality primitive procedure, including its input validation, and the way to integrate it into the `Primitives` class:

```
Primitives.include('equal?') do |args, ctx|
  if args.length != 2
    raise InterpreterError.new(
      'equal? expects two arguments!')
  else
    args[0] == args[1] ? :t : []
  end
end
```

The parameter received by the `include` method may be a string or a symbol. The associated code block gets converted to a `Proc` object and gets stored in a private hash contained within the `Primitives` class. Ruby's `true` and `false` values are not supported by the language being interpreted, so equivalent values should be explicitly returned (the symbol `:t` and the empty list `[]`, respectively).

2.4 Defining Special Forms

The SIF uses strict evaluation when calling a procedure. This means that all the arguments of a procedure are completely evaluated before being called. Sometimes a different behavior from the one described is required. If that's the case, a new special form can be defined.

Defining a special form requires creating a new class with the following characteristics:

- It must be a subclass of the `Node` class. Instances of this class will be used as nodes for the interpreter pattern tree.
- It must include a `special_form` statement to indicate the name that this special form will have when used as an S-expression. The name must be specified as a string, and it must be a valid S-expression symbol name.
- It must define an `initialize` method (similar to a constructor in other languages) receiving one array parameter that contains all the arguments (as Ruby data values) to this special form. This method has two purposes:
  1. Validate the correct syntax for this special form.
  2. Construct the subtrees for this special form by calling the `Node.build` class method. The resulting child nodes should be stored in instance variables. This step converts Ruby core data into the corresponding interpreter pattern tree.
- It must define the `interpret` method, which receives a context as a parameter. This is the central method in the interpreter pattern. Usually, the `interpret` method is called recursively for some or all of its child nodes, according to the semantics of this special form, which also establishes what the return value should be.

The previous description can be summarized as follows:

> *For any special form being defined, identify its **syntax** and **semantics**. The **syntax** must be checked in the* **`initialize`** *method, while the **semantics** must be implemented in the* **`interpret`** *method.*

For example, suppose we want to implement the `if` special form. Its syntax and semantics are as follows:

Syntax: **(if** ⟨*condition*⟩ ⟨*consequent*⟩ ⟨*alternative*⟩**)**

Semantics: Evaluate ⟨*condition*⟩, if the resulting value is not an empty list, evaluate and return ⟨*consequent*⟩, otherwise evaluate and return ⟨*alternative*⟩.

The following code takes into account all the stated requirements. This is the complete implementation of the `IfNode`:

```
class IfNode < Node
  special_form 'if'
  def initialize(args)
    if args.length == 3
      @condition = Node.build(args[0])
      @consequent = Node.build(args[1])
      @alternative = Node.build(args[2])
    else
      raise InterpreterError.new(
        'if takes three arguments!')
    end
  end
  def interpret(context)
    if @condition.interpret(context) != []
      return @consequent.interpret(context)
    else
      return @alternative.interpret(context)
    end
  end
end
```

No other part of the SIF has to be modified in order to extend it with this special form. The trick is done through the `special_form` statement, which adds a reference to this class into an internal table that is later used by the `Node.build` class method to create `IfNode` instances whenever the `if` symbol is found after an opening parenthesis.

The SIF is now able to evaluate an S-expression like this one:

```
(if (< (* 3 3) (+ 3 3)) (* 3 3) (+ 3 3)) ⇒ 6
```

In the previous `IfNode` class definition, the `if` special form was implemented using Ruby's `if` statement. Most of the time, this mapping from S-expressions to Ruby is not as straightforward as observed here, and that's when things start to get interesting.

3. ADVANCED LANGUAGE ISSUES

The SIF can be extended in several interesting ways. The next subsections document some possibilities that the author has explored together with his students.

3.1 Functional Programming

A small pure functional language interpreter can be built with the SIF core infrastructure plus the following special forms: `quote`, `define`, `if` and `fn`. These special forms behave mostly like their Scheme programming language counterparts [2]. Some examples:

```
(define composite
  (fn (f g)
    (fn (x) (f (g x)))))
(define f1 (fn (x) (+ x 3)))
(define f2 (fn (x) (* x 2)))
(define f3 (composite f1 f2))
(define f4 (composite f2 f1))
(f3 1)      ⇒ 5
(f4 1)      ⇒ 8
(define map
  (fn (proc lst)
    (if (null? lst) ()
        (cons (proc (first lst))
              (map proc (rest lst))))))
(map f2 (quote (4 -3 10 5))) ⇒ (8 -6 20 10)
```

The `fn` special form is used for creating procedures. Here's its implementation:

```
class FnNode < Node
  special_form 'fn'
  def initialize(args)
    # Validation code omitted
    @params = args[0]
    @body = Node.build(args[1])
  end
  def interpret(context)
    return StaticScopeProcedure.new(
      context, @params, @body)
  end
end
class StaticScopeProcedure
  def initialize(context, params, body)
    @context = context
    @params = params
    @body = body
  end
  def call(args, ctx)
    return @body.interpret(
      extend_context(@params, args, @context))
  end
end
```

The `StaticScopeProcedure` object returned by the `interpret` method of `FnNode` class allows having lexical closures. Instances of this class are callable (contain a `call` method), meaning that they can be used as the first element of a procedure application list. The `extend_context` method invoked inside the `call` method is responsible for creating a shallow copy of the stored context (`@context`) and adding to this copy the corresponding bindings between the formal (`@params`) and actual (`args`) parameters. This extended context is then used to evaluate the procedure's body (`@body`). Note that the context being cloned is the one available when the procedure was created and not when it was actually called. This subtle detail is the difference between static and dynamic scoping.

It's worth pointing out a couple of important issues with this implementation. Firstly, self and mutual recursions are supported thanks to the ways in which the contexts are used and cloned. Secondly, when a procedure gets called, Ruby's ordinary invocation mechanism is used. This means that there is no support for tail-recursion optimization, as it might be expected from a functional language. This could be considered a major shortcoming of the current SIF implementation.

3.2 Imperative Programming

The SIF allows imperative programming by adding two special forms: `set!` (assign a new value to a previously defined variable, and return `nil`) and `begin` (evaluate a sequence of expressions from left to right, and return the result of the last expression). A new class, called `Environment`, also needs to be defined. This class is used to represent contexts for imperative programs. Its interface is similar to a hash, but internally it uses a list of frames. Individual frames, which also happen to be hashes, represent a scoping level. Frames are shared within the environments of nested procedures, allowing every variable definition and assignment to occur inside the expected frame.

Special environment objects were not required in assignmentless programs because whenever a procedure was called, the hash used as the context was merely copied. Any new parameter whose name already existed in the hash copy was simply overwritten with a new value. Because the values in a context were never required to be updated (because there is no assignment), there could be multiple copies of the same variables in different hashes without this being a problem.

3.3 Using Continuations

Unlike many other languages, Ruby supports first class continuations. These can be used together with the SIF to implement special flow control structures such as exceptions or loop breaks. The following code illustrates how to use continuations to implement a `break` inside a `while`. This code uses the `callcc` method to capture the current continuation. Assuming that the code is running in a single execution thread, we can use a global stack in order to allow multiple nested `while` statements. The continuation is sent as a parameter to the associated block and is pushed into the stack referred by a class variable. When the `break` procedure is called, the continuation at the top of the stack is called. This causes the program execution flow to safely jump to the end of the corresponding `callcc` instruction, effectively ending the most nested `while` loop. When the `while` terminates (normally or through a `break` instruction) the continuation at the top of the stack is popped.

```
class WhileNode < Node
  special_form 'while'
  @@stack = []
  def initialize(args)
    # syntax validation code omitted
    @condition = Node.build(args[0])
    @body = Node.build(args[1])
  end
  def interpret(context)
    callcc do |continuation|
      @@stack.push(continuation)
      while @condition.interpret(context) != []
        @body.interpret(context)
      end
    end
    @@stack.pop
    return nil
  end
  def WhileNode.do_break
    @@stack.last.call if !@@stack.empty
  end
end

Primitives.include('break') do |args, ctx|
  WhileNode.do_break
end
```

This is how the `while` and `break` statements could be used:

```
(define i 1)
(define r 1)
(while (< i 11) (begin (set! r (* r i))
                       (if (= i 5) (break) ())
                       (set! i (+ i 1))))
r    ⇒ 120
i    ⇒ 5
```

4. CLASS EXPERIENCE

The author has used the S-expression Interpreter Framework in his Programming Language course several times since mid 2005. This course is mandatory for third year computer science undergraduate majors. Before taking the course, all students have had a fair amount of experience designing and programming in Java and C/C++. They've already taken courses on data structures and theory of computation, so they're familiar with stacks, queues, hash tables, regular expressions and context free grammars.

The course is taught during a sixteen week semester. It's mainly focused on teaching programming paradigms and language design issues. The SIF is presented to students during the last four weeks. At this point, they've been exposed already to Scheme and Ruby, so it's possible to thoroughly deal with the framework's inner workings. As extra-class activities, students are required to extend the SIF in order to implement some of the following features:

- Operations that deal directly with the primitive data types, for example arithmetic and type conversion operations.
- New syntactic elements based on others previously defined (syntactic sugar). This basically means using special forms as a *procedural* macro facility.
- Lazy (delayed) evaluation support, like short-circuit logical operations (C's `&&` and `||`) or infinite streams (Scheme's `delay` and `force`).
- Local variables with dynamic scope, as available in Common Lisp or Perl.
- Flow control statements, including conventional loops (C or Pascal), pattern matching (Haskell or Erlang), exception handlers (C++ or Java), coroutines (Modula-2), and iterators (Python).
- A simple object-oriented system, combining lexical closures and some syntactic sugar, as can be accomplished in Scheme [2].

The neat thing about the SIF is that all the exercises in the previous list can be solely accomplished by applying the techniques demonstrated in sections 2.3 and 2.4.

Once the course has concluded, most students demonstrate through formal assessment that they are able to:

- Extend the SIF by writing their own primitive procedures and special forms using Ruby and object-oriented design principles and techniques.
- Read and write S-expression source programs that use a certain programming style or apply a specific language concept.

Informal interviews with students at the end of the semester suggest that, in general, they enjoyed the SIF approach and that it helped them to have a better understanding of how programming languages work.

5. CONCLUSIONS

The S-expression Interpreter Framework (SIF) provides the basic building blocks that allow writing interpreters that can easily be studied, extended and modified. Particularly useful is the SIF's special form definition mechanism, which allows a clear distinction between the syntax and semantics of any language construct. The author has used the SIF in his Programming Languages course for several semesters now with positive results. The full source code for the SIF is freely available for anyone to use and modify under the GNU General Public License at the following Web site: *http://sif.arielortiz.com/*

6. REFERENCES

[1] Abelson, Harold, and Gerald Sussman. *Structure and Interpretation of Computer Program, Second Edition.* The MIT Press, Cambridge, MA, 1996.

[2] Dybvig, Kent. *The Scheme Programming Language, Third Edition.* The MIT Press, Cambridge, MA, 2003.

[3] Friedman, Daniel, Mitchell Wand, and Christopher Haynes. *Essentials of Programming Languages, Second Edition.* The MIT Press, Cambridge, MA, 2001.

[4] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[5] Hennessy, Wade. *Common Lisp.* McGraw-Hill College, New York, NY, 1989.

[6] Kamin, Samuel. *Programming Languages: An Interpreter Based Approach.* Addison-Wesley, Reading, MA, 1990.

[7] Matsumoto, Yukihiro. *What's Ruby.* http://www2.ruby-lang.org/en/20020101.html Accessed November 10, 2007.

[8] Thomas, Dave, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition.* Pragmatic Bookshelf, Raleigh, NC, 2004.