Yet Another Lisp Interpreting Experiment

John O'Connor, Paul Hudak

Introduction

One of the historical attractions of the Lisp family of programming languages has been their flexibility. Lisp was one of the first languages to include dynamic typing, and Lisp macros allow the programmer to create new binding forms and control structures. This flexibility is one of the reasons for Lisp's popularity in the AI community. Programmers can implement domain-specific mini-languages without losing the infrastructure of a full language implementation.

Another strength of Lisp is its elegance. The S-expression syntax allows Lisp to treat lists as code. Thus Lisp macros, by constructing and returning lists of symbols, actually return code. This is one of the more confusing aspects of Lisp for newcomers, to be sure, but it also simplifies the language in an important sense. Where most languages have code and data, Lisp has only data. In the long run, fewer "moving parts" means fewer distinct concepts that a programmer needs to keep in mind to understand his language.

More recent developments, however, have seen Lisp fall behind. In particular, the Smalltalk language demonstrated that it is possible to treat everything in a language as an "object." The popular Python and Ruby languages have adopted many concepts from Smalltalk with great success. Several object-oriented variants of Lisp have been developed, including the Common Lisp Object System, but these have not achieved nearly the same popularity. We wish to suggest that one of the failings of various Lisp object systems is that they are built for the wrong reasons.

Through object-orientation, Smalltalk was able to achieve greater flexibility and greater simplicity. These goals are essentially orthogonal to the goal of facilitating an object-oriented programming style. Lisp object systems have attempted the latter, and as a result they have generally made their languages more complicated and less consistent. We believe that object-orientation can greatly enhance Lisp's core strengths, but only if these concepts are introduced consistently throughout the entire language.

This is the intention of Yet Another Lisp Interpreting Experiment (or, "Yalie"). Our goal is to demonstrate a Lisp variant in which everything is an object and all function calls are based on message passing. In doing this, we hope to make the language less complicated, rather than more so. An interactive interpreter for Yalie, written in Python, accompanies this paper. What follows is a partial formal description of the language implemented in that interpreter. For a practical description, please see the accompanying README file. This paper assumes basic familiarity with a modern Lisp dialect such as Scheme or Common Lisp.

Concepts

Broadly speaking, there are two major categories of object-oriented languages.[1] The most familiar and popular are the "class-based" languages, including C++, Java, and Smalltalk. The other category is "object-based" or "prototype-based" languages, like JavaScript and Lua. Class-based languages distinguish classes from the objects that belong to them, and inheritance is accomplished within the class hierarchy. Object-based languages omit this distinction, and instead allow objects to inherit from one another and produce their own clone or child objects. In practice, object-based languages often approximate classes by using certain objects solely for inheritance, and for that reason—and because of the wider popularity of class-based languages—some have argued that making classes an explicit part of the language is more convenient.[2] Our primary purpose here, though, is simplicity and consistency, so ours will be an object-based semantics.

We will also be taking a less conventional approach to specifying the semantics of our language. With the goal of unifying code and data, we will first describe the translation from syntax to literal objects and then separately define the methods available to those objects. This is also intended to highlight the fact that the vast majority of the functionality of Yalie is contained in predefined methods for our objects, rather than in the syntax of our language.

The semantics of our language will be built from three components: objects, the scope, and the store. Objects will be treated as simple tuples, of the form (p, ϕ_1, ϕ_2, d), containing a reference to a parent object, a scope of methods, a scope of member objects, and a data field for builtin data like integers. Scopes—denoted by ϕ—are maps from names to natural numbers or (in the case of method scopes) functions, and the store—denoted by σ—is a map from natural numbers to objects. In this way, the store is intended to simulate a persistent memory of mutable objects, and for our purposes we can understand the store as unique (in contrast to scopes, of which there will be many). The extension of a scope or store is represented by the expression "$[\phi|key \to val]$". The smallest currently unassigned integer in a store is denoted by "next(σ)". The empty scope and the empty store are denoted by ϕ_0 and σ_0, respectively.

We have put Yalie together so that externally it resembles a traditional Lisp, but the underlying semantics are very different. In particular, functions and special forms are just objects with a predefined "call" method. The mechanism of a function call is that the "eval" method of a list implicitly evaluates its first element and passes the rest of the list as arguments to the first element's call method. The "msg" special form is then provided with a call method that allows the user to pass arbitrary messages.

Syntax

Yalie contains only three literal objects: integers, symbols, and lists. Integers are written literally. Symbols are written as any string of characters (other than those defined as punctuation in this section) that is not an integer. Lists are written as a sequence of whitespace-separated literals enclosed by matching parentheses. We can define the parsing function as taking for arguments a code snippet and an initial store, and returning an updated store and the number referring to the newly created object in that store.

Integers are parsed as

$$\text{parse}(i \in \mathbb{Z}, \sigma) = (n, \sigma'),$$

where $\sigma' = [\sigma|\,\text{next}(\sigma) \to (\text{INT}, \phi_0, \phi_{2\text{INT}}, i)]$ and $n = \text{next}(\sigma)$. "INT" is the reference number of the integer superobject, which we will define later, and $\phi_{2\text{INT}}$ is its member scope.

Symbols are parsed as

$$\text{parse}(s \in \mathbb{S}, \sigma) = (n, \sigma'),$$

where

$$\sigma' = [\sigma|\,\text{next}(\sigma) \to (\text{SYMBOL}, \phi_0, \phi_{2\text{SYMBOL}}, i)]$$

and $n = \text{next}(\sigma)$. "SYMBOL" is the reference number of the symbol superobject, which we will again define later, and $\phi_{2\text{SYMBOL}}$ is its member scope, again as before.

Finally, lists are parsed into "cons" objects, as is traditional for Lisp languages. The data element of these cons objects will be an ordered pair of the form (a, b), and they will be linked together to form linked lists terminated in a "nil" object.

$$\text{parse}(\text{"()"}, \sigma) = (n, \sigma'),$$

where $\sigma' = [\sigma|\,\text{next}(\sigma) \to (\text{NIL}, \phi_0, \phi_{2\text{NIL}}, \varnothing)]$ and $n = \text{next}(\sigma)$. Also,

$$\text{parse}(\text{"(a, b,\ldots z)"}, \sigma) = (n, \sigma'),$$

where

$$\sigma' = [\sigma|\,\text{next}(\sigma) \to (\text{CONS}, \phi_0, \phi_{2\text{CONS}}, \text{parse}(\text{"(b,\ldots z)"}, \sigma')]$$

and $n = \text{next}(\sigma)$.

In addition to these literals, we define some extra translational syntax for convenience. Message passing is invoked by the expression `(msg foo msgForFoo [args...])`, so we allow the period operator to translate into that S-expression. We let `a.b` translate to `(msg a b)` and `(a.b c d...)` translate to `(msg a b c d...)`. When there are multiple consecutive infixed objects at the front of an S-expression, we evaluate from left to right and allow the final infix to capture the expression. Thus `(a.b.c d e)` translates to `(msg (msg a b) c d e)`. We also define the single quote as a prefix operator that takes precedence over the period, and we translate `'a` to `(quote a)`. Both `msg` and `quote` will be defined below.

Builtin Objects and Methods

The meat of the Yalie language is in its builtin objects and methods. As we have seen, object literals all inherit from parent objects that will be exposed to the user for potential modification. These parent objects will have methods defined for their children to inherit.

We will formally specify only a few methods here. A short description of all the methods used by the interpreter can be found in the accompanying README file.

In order to specify these methods, we will need a few helper functions. The lookup function, L, takes a store, an object reference, and a method name and returns the method of that name belonging to the closest ancestor of the object.

$$L(\sigma, o, \text{“name”}) = \phi_1(\text{“name”}) \text{ if “name”} \in \phi_1 \quad \text{else } L(\sigma, p, \text{“name”}),$$

where $\sigma(o) = (p, \phi_1, \phi_2, d)$.

The method invoker, M, returns the value of a method call on a given object under a given scope and store. That is, it invokes functions which return an object reference, a new scope, and a new store.

$$M(\phi, \sigma, o, \text{“name”}, \langle a_1, \dots a_n \rangle) = f(\phi, \sigma, o, \langle a_1, \dots a_n \rangle)$$

where $f = L(\sigma, o, \text{“name”})$.

Lastly, our wrapper function, W, takes a function of some arguments and returns a function that implicitly evaluates those arguments before beginning. This is necessary because the “call” function will need to circumvent implicit evaluation of arguments. Evaluation is accomplished by the “eval” method, which each object will define.

$$W(f)(\phi, \sigma, o, \langle a_1, \dots a_n \rangle) = f(\phi, \sigma, o, \langle e_1, \dots e_n \rangle)$$

where $e_i = M(\phi, \sigma, a_i, \text{“eval”}, \langle\rangle)$.

Now we can define some of the important methods. The basic `eval` function simply returns the value of the caller.

$$\phi_{\text{ROOT}}(\text{“eval”})(\phi, \sigma, o, \langle\rangle) = (o, \phi, \sigma)$$

where σ_{ROOT} is the method scope of the root object. Most objects inherit this method, with the exception of symbols and lists. The symbol “eval” method is

$$\phi_{\text{SYMBOL}}(\text{“eval”})(\phi, \sigma, o, \langle\rangle) = (\phi(d_{\sigma(o)}), \phi, \sigma).$$

Finally, for lists,

$$\phi_{\text{CONS}}(\text{“eval”})(\phi, \sigma, o, \langle\rangle) = M(\phi', \sigma', e_1, \text{“call”}, \langle a_2, \dots a_n \rangle)$$

where $\langle a_1, \dots a_n \rangle$ are the contents of the list, and $(e_1, \phi', \sigma') = M(\phi, \sigma, a_1, \text{“eval”}, \langle\rangle)$.

For an example of the use of the W wrapper, the method for addition over the integers is

$$\phi_{\text{INT}}(\text{“eval”}) = W\lambda(\phi, \sigma, o, \langle e_1 \rangle).(n, \phi, \sigma')$$

where $n = (\text{INT}, \phi_0, \phi_{2\text{INT}}, d_{e_1} + d_{\sigma(o)})$ and $\sigma' = [\sigma |\, \text{next}(\sigma) \to n]$.

Finally, here is the critical “call” method for the `msg` function object.

$$\phi_{\text{MSG}}(\text{“call”})(\phi, \sigma, o, \langle a_1, a_2, \dots a_n \rangle) = M(\phi, \sigma, a_1, d_{a_2}, \langle a_3, \dots a_n \rangle).$$

Conclusions

Several different interpreters were created at different stages in this project, but the “call” and “eval” semantics were only included in the final version. The ease of coding these semantics relative to the more standard approach (using a central eval loop) was striking, both in reduced programming time and in reduced debugging time. These semantics are also far easier to modify than any previous.

The language is still very small, so some of the benefits of the object-oriented approach are not yet apparent. One tangible benefit is that, despite having gutted the entire language to allow the ubiquitous use of objects, the only difference visible in the global namespace is the existence of the `msg` function. All other object-oriented functionality is confined within the method spaces. If we were to further extend the language—with strings and file handles for example—we would see the same benefits repeated, with new functionality mostly confined to method spaces. Not only does this approach keep the global namespace free of clutter, it also provides helpful documentation in the form of method lists.

The example of adding a factorial method to the entire set of integers (see README) demonstrates the flexibility of the object-oriented approach, and though the language is still small, the improved consistency of the semantics has already made itself felt during the implementation process. We look forward to finding new ways of leveraging the object abstraction, both by itself and in conjunction with Lisp's macro system.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* New York: Springer, 1996.

[2] K. B. Bruce. *Foundations of Object-Oriented Languages.* Cambridge, Massachusetts: MIT Press, 2002.