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 instructions to be executed. 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 (OOP) 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 formal description of the language implemented in that interpreter. For a more practical description with examples, please see the accompanying README file. This paper assumes the reader is familiar with a modern Lisp dialect such as Scheme or Common Lisp.

Concepts

Object-based Languages

Broadly speaking, there are two major categories of object-oriented languages[1]. The most familiar and widely used 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 is simplicity, however, so ours will be an object-based semantics. We will also draw an unconventional distinction between object-*oriented* languages, by which we mean those that encourage a programming style centered around objects and inheritance, and object-*based* languages, which merely use objects as the fundamental unit of their semantics. Yalie is an object-*based* language, and our intention is to replicate the functional and procedural paradigms traditional in Lisp languages. Obviously, an object-based semantics will make OOP fairly natural in Yalie, but our primary goals of simplicity, consistency, and flexibility should benefit any programming paradigm.

Message Passing

As a demonstration of the consistency possible with objects, all of the semantics of our language aside from literal syntax is based on message passing. This is very different from traditional Lisp semantics and requires some explanation. For an example, suppose we wish to evaluate the S-expression

```
(+ 1 2).
```

That is a list object (a series of cons objects, in fact) containing a symbol object and two integer objects. Evaluation begins by calling the `eval` method of the list. As with all Lisp dialects, S-expressions represent function calls, so the `eval` method of a list calls the `eval` method of its first element, expecting a function or special form object of some kind. The `eval` method of the `+` symbol returns the binding of that symbol in the calling scope, which is indeed a function. The list `eval` then invokes the `call` method of that object, passing the remainder of the list as arguments, and finally returns 3.

Traditionally, S-expressions represented function calls because a central evaluation loop interpreted them that way. Here they represent function calls because their `eval` method carries out a function call. Note also that functions and forms in Yalie are just ordinary objects with a defined `call` method. We will construct a few predefined global functions and forms in Yalie, and these will be just ordinary objects with `call` methods that we define usefully.

Another key difference in this function is the way `+` is defined, which is not immediately visible. Rather than being a builtin operation that knows about integers, `+` simply calls invokes a method of its arguments. This method is also named `+`, though it resides in method tables rather than the global scope as the `+` function does. One of the global functions we will define is `msg`, which allows the user to invoke methods explicitly. Expressed using `msg`, the S-expression we considered above can be rewritten as

```
(msg 1 + 2).
```

That is, adding 1 to 2 means sending the `+` message to the object 1 with the argument 2. In full detail, the `eval` method of that list evaluates the symbol `msg`, whose value is a special form object with a `call` method that passes that messages. When that `call` method is invoked, it evaluates its first argument, the 1 object, and invokes the method named by its second argument, passing the rest of its arguments to that method.

Something to keep in mind as we continue is that anything defined as a method can be invoked explicitly or even redefined by the user. Thus, the first S-expression could be rewritten again as

```
(msg + call 1 2),
```

and the second could even be rewritten

```
(msg msg call 1 + 2).
```

The latter is not particularly useful, but it illustrates what is going on "under the hood." We will discuss the facilities for defining and redefining new methods in later sections.

Semantic Machinery

The semantics of our language will be built from several different components. Persistent, often mutable objects will be the core of our language, so evaluation will use both a scope and a store. A scope is a map from names to positive integer values. We will refer to these integers suggestively as pointers. We denote scopes by the variable φ, with the global scope—that is, the builtin scope in which program evaluation begins—denoted by Φ. A store is then a map from these pointers to objects, and this indirection allows us to mutate objects by modifying the store. We denote a store by the variable σ, and the global store, analogous to the global scope, is denoted by Σ. An extension to a scope or a store is denoted by the expression $[\sigma|\text{key} \to \text{val}]$. We will frequently refer to the pointers to certain builtin objects using capitalized names like ROOT, the primary object at the top of the inheritance hierarchy, or INT, the object from which all integers inherit. The objects to which these pointers refer are then written $\Sigma(\mathsf{ROOT})$ and $\Sigma(\mathsf{INT})$. The actual values of these pointers are arbitrary, so long as they are constant and unique. The only other way we will refer to pointer values is the $\text{next}(\sigma)$ function, whose value is the smallest pointer not currently in the domain of σ. We will generally use the variable ω to denote pointers, though we will also use the variable a to refer to arguments to functions and e to refer to the results of evaluation.

An object has four important properties. It has a parent object from which it inherits method functions, a map of its own from names to method functions, a scope of member elements, and a field for underlying data such as an integer or a name. We will define objects as ordered 4-tuples, written

$$\langle p, \mu, \varphi, d \rangle \,.$$

For example, the integer five would be represented as

$$\langle \mathsf{INT}, \mu_0, \varphi_0, 5 \rangle \,,$$

where μ_0 is the empty function map, and φ_0 is the empty scope. We will frequently use the notation $\mu_{\sigma(\omega)}$ and $\varphi_{\sigma(\omega)}$ to refer to the function map or scope elements of the object $\sigma(\omega)$.

We have already mentioned the function $\text{next}(\sigma)$, but we should like to introduce two more semantic function here that we will use later. The first is our function for looking up methods, using the "message-passing" terminology of Simula and Smalltalk. When a method cannot be found in the method table of a given object, it's parent is queried recursively.

$$\text{lookup}(\text{“msg”}, \sigma, \omega) = \begin{cases} \varphi_{\sigma(\omega)} & \text{if “msg”} \in \text{Domain}(\varphi_{\sigma(\omega)}) \\ \text{lookup}(\text{“msg”}, \sigma, p_{\sigma(\omega)}) & \text{otherwise} \end{cases}$$

The second function we need to define at the moment is the semantic function for invoking methods. We mentioned the global `msg` object in the preceding section, and this semantic function will eventually form the core of that object's `call` method, in addition to being used explicitly by many other methods. The invocation of a method requires the current store and scope as well as pointers to invoking object and all the arguments of the method, the latter expressed as an ordered tuple. Method functions will be defined so that the value of an invocation is a 3-tuple containing the new store and scope and a pointer to the object returned by the method.

$$\begin{aligned} &\text{invoke}(\text{“msg”}, \sigma, \varphi, \langle a_1, \ldots a_n \rangle) = \\ &\qquad (\text{lookup}(\text{“msg”}, \sigma, \omega))(\sigma, \varphi, \omega, \langle a_1, \ldots a_n \rangle) \end{aligned}$$

Syntax

With the goal of unifying code and data, we will take a two-part approach to specifying the semantics of our language. First, we will describe the translation from syntax to literal objects. Then we will define the methods available to those objects. Invocation of the `eval` methods of our code will be responsible for most of the work done by a program, and ultimately our denotational semantic operator, $[\![-]\!]$, will be the simple combination of the parse and invoke operations with Σ and Φ.

Literal Objects

Yalie contains only three literal objects: integers, symbols, and lists. Integers are written as strings of decimal digits. 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 define a parsing function that takes as arguments an abstract syntax expression and an initial store, and returns a tuple containing a pointer to the new object and the updated store. Integers are thus parsed as

$$\text{parse}(\texttt{NUM}, \sigma) = \langle \text{next}(\sigma), [\sigma | \,\text{next}(\sigma) \to \left\langle \mathsf{INT}, \mu_0, \varphi_{\Sigma(\mathsf{INT})}, \texttt{NUM} \right\rangle] \rangle\,,$$

and symbols are parsed as

$$\text{parse}(\texttt{SYM}, \sigma) = \langle \text{next}(\sigma), [\sigma | \,\text{next}(\sigma) \to \left\langle \mathsf{SYMBOL}, \mu_0, \varphi_{\Sigma(\mathsf{SYMBOL})}, \texttt{SYM} \right\rangle] \rangle\,.$$

Note that the parent of each new integer or symbol object is the global integer or symbol object, respectively. These objects are created with an empty method table, meaning that they inherit all of their methods by default. They also copy the member scope of their parent, which is empty by default unless modified by the user.

Finally, lists are parsed into "cons" objects, as is traditional for Lisp languages. The underlying data field, d, of these cons objects will be an ordered pair of the form $\langle a, b \rangle$, and they will be chained together to form linked lists terminated in a "nil" object. List parsing will be defined in two steps. First, the empty list is parsed as

$$\text{parse}(\texttt{()}, \sigma) = \langle \text{next}(\sigma), [\sigma | \,\text{next}(\sigma) \to \left\langle \mathsf{NIL}, \mu_0, \varphi_{\Sigma(\mathsf{NIL})}, \varnothing \right\rangle] \rangle\,.$$

This is similar to the integer and symbol parsing operations above, though note that the data field of nil objects is ignored. List parsing for nonempty lists can now be defined recursively.

$$\text{parse}(\texttt{(a b ... z)}, \sigma) = \langle \text{next}(\sigma''), [\sigma'' | \,\text{next}(\sigma'') \to \left\langle \mathsf{CONS}, \mu_0, \varphi_{\Sigma(\mathsf{CONS})}, \langle F, R \rangle \right\rangle] \rangle$$

where

$$\langle F, \sigma' \rangle = \text{parse}(\texttt{a}, \sigma)$$
$$\langle R, \sigma'' \rangle = \text{parse}(\texttt{(b ... z)}, \sigma')$$

This expression can be slightly confusing. F denotes the pointer to the first parsed object in a list, and σ' is the store after parsing that object. R then denotes the pointer to the rest of the list, acquired recursively, and σ'' is the store after all that parsing is done. The whole list is finally assembled by adding to σ'' the cons cell at the head of the list.

As we mentioned above, we can now define the denotational semantic operator for Yalie expressions.

$$[\![-]\!]\ \Sigma\ \Phi = M(\text{"eval"}, \Sigma', \Phi, \omega, \langle\rangle),$$

where $\langle \Sigma', \Phi, \omega \rangle = \text{parse}(\,-\,, \Sigma)$.

Syntactic Sugar

In addition to these literals, we define some extra translational syntax for convenience. First we give a dot operator for message passing, and second we will provide a quote operator to protect objects from automatic evaluation.

As we mentioned above, message passing is invoked using the `msg` special form, in the manner

`(msg obj message [args...])`.

We supply the dot operator to avoid writing that entire S-expression for every method call. Without parentheses, `a.b` translates as

`a.b` → `(msg a b)`.

When placed after the first element of an S-expression, the dot operator subsumes the rest of the S-expression as arguments. Thus `(a.b c...)` translates as

`(a.b c...)` → `(msg a b c...)`.

Finally, when there are multiple consecutive infix operations 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 as

`(a.b.c d e)` → `(msg (msg a b) c d e)`.

Another global special form that we define is the `quote` operator, which protects objects from evaluation. We define the accompanying grave quote as a prefix operator that takes precedence over the dot, and we translate `'a` as

$$\texttt{'a} \to \texttt{(quote a)}.$$

Accompanying the quote syntax we provide unquote and splice syntax, in the manner of Common Lisp and Scheme. These translate as

$$\texttt{,a} \to \texttt{(unquote a)}$$
$$\texttt{;a} \to \texttt{(unquote-splice a)}.$$

Note that `unquote` and `unquote-splice` are not defined as separate operators but are rather ordinary symbols that the `quote` operator looks for and handles itself. Note also that the semicolon is used for splicing in Yalie, instead of the comma-at operator used in other Lisps. Comments in Yalie are denoted by a hash mark, as is common in scripting languages.

Builtin Objects and Methods

Yalie defines a number of objects that exist in the global scope and store when any program begins. We have already encountered the ROOT, INT, SYMBOL, CONS, and NIL objects. Other generic parent objects include FUNCTION, FORM, and their mutual parent, OPERATOR. Predefined function and form objects, which inherit from these, include MSG and QUOTE. Others we have not yet encountered include IF and WHILE. By convention, we give each such object in Σ a corresponding binding in Φ.

The meat of our language is not this hierarchy itself but rather the methods defined for these objects. Below we provide formal definitions of many of the most important methods implemented in the Yalie interpreter. We will begin with several methods inherited by every object and add to that description several important methods outside the root object.

The `parent` method returns the parent of a given object, and it is a good example with which to start. The definition reads

$$\mu_{\Sigma(\mathsf{ROOT})}(\text{“parent”})(\sigma, \varphi, \omega, \langle\rangle) = \left\langle [\sigma, \varphi, p_{\sigma(\omega)}] \right\rangle .$$

Recall that $\mu_{\Sigma(\mathsf{ROOT})}$ is the method table of the ROOT object in the global store, Σ. This method does not modify the store or the scope, so those are returned directly. As mentioned above, $p_{\sigma(\omega)}$ refers to the p element of the object $\sigma(\omega)$, which is the pointer to the parent of the calling object.

We can also introduce the default `eval` method, which most objects will inherit. This is just the identity method on the caller, so the definition is even simpler.

$$\mu_{\Sigma(\mathsf{ROOT})}(\text{“parent”})(\sigma, \varphi, \omega, \langle\rangle) = \langle [\sigma, \varphi, \omega] \rangle .$$

The next two methods we define are `copy` and `child`. These methods are fundamental to an object-based language. Apart from literal objects, copying is the simplest way to produce a new object. A copied object shares the same parent as its source, and it replicates its source's the method table, member scope, and underlying data. After copying the two objects are independent; changing one does not affect the other. The copy method is defined as

$$\begin{aligned}\mu_{\Sigma(\mathsf{ROOT})}&(\text{“copy”})(\sigma, \varphi, \omega, \langle\rangle) \\ &= \langle [\sigma|\, \mathrm{next}(\sigma) \to \sigma(\omega)], \varphi, \mathrm{next}(\sigma) \rangle .\end{aligned}$$

The other essential way to create a new object is to spawn a child object that inherits from a parent. Children also copy the member scopes and underlying data of their parent, but their method table is initially empty, and they inherit all their methods from the parent. Thus any changes made to the methods of the parent will be reflected in the child. The child method is defined as

$$\begin{aligned}\mu_{\Sigma(\mathsf{ROOT})}&(\text{“child”})(\sigma, \varphi, \omega, \langle\rangle) \\ &= \left\langle [\sigma|\, \mathrm{next}(\sigma) \to \left\langle \omega, \mu_0, \varphi_{\sigma(\omega)}, d_{\sigma(\omega)} \right\rangle], \right. \\ &\qquad\qquad \left. \varphi, \mathrm{next}(\sigma) \right\rangle .\end{aligned}$$

Another pair of important methods is the `get` and `set` methods, which query and modify member values. These are not used by any other builtin functions or methods, but they are an important facility for user-defined objects. The `get` method is the simpler of the two, and it is also the first method we have defined so far that takes an argument.

$$\mu_{\Sigma(\mathsf{ROOT})}(\text{“get”})(\sigma, \varphi, \omega, \langle a_1 \rangle) = \left\langle \sigma, \varphi, \varphi_{\sigma(\omega)}(d_{\sigma(a_1)}) \right\rangle .$$

That is, the `get` method receives a symbol, a_1, and references its data element in the member scope of the caller. The `set` method takes two elements, and it is the first method so far that will evaluate an argument. The first argument is a symbol giving the member name, and the second is evaluated to give the new value of the member and the return value of `set`.

$$\mu_{\Sigma(\mathsf{ROOT})}(\text{“set”})(\sigma, \varphi, \omega, \langle a_1, a_2\rangle) = \Big\langle [\sigma' | \omega \rightarrow \big\langle p_{\sigma(\omega)}, \mu_{\sigma(\omega)}, [\varphi_{\sigma(\omega)} | d_{\sigma(a_1)} \rightarrow e_1], d_{\sigma(\omega)} \big\rangle], \varphi', e_1 \Big\rangle .$$

where $\langle \sigma', \varphi', e_1\rangle = M(\text{“eval”}, \sigma, \varphi, a_2, \langle\rangle)$.

The final method of the root object that we define here (others can be found in the accompanying README) is `dup`, which serves in place of the “super” keyword from languages like C++. When a child object redefines an inherited method but wishes to invoke the inherited version, it must first duplicate that original method to another binding. The `dup` method thus takes two arguments, a method name and a name for the new binding. Note that even if the original binding is inherited, the new binding will be local to the calling object.

$$\mu_{\Sigma(\mathsf{ROOT})}(\text{“dup”})(\sigma, \varphi, \omega, \langle a_1, a_2\rangle) = \Big\langle [\sigma | \omega \rightarrow \big\langle p_{\sigma(\omega)}, [\mu_{\sigma(\omega)} | d_{\sigma(a_2)} \rightarrow \text{lookup}(d_{\sigma(a_1)}, \sigma, \omega)], \varphi_{\sigma(\omega)}, d_{\sigma(\omega)} \big\rangle], \varphi, \omega \Big\rangle .$$

One critical method outside of the root object is the `eval` method of symbols. Rather than evaluating to themselves, as most objects do, symbols evaluate to their binding in the current scope.

$$\mu_{\Sigma(\mathsf{SYMBOL})}(\text{“eval”})(\sigma, \varphi, \omega, \langle a_1\rangle) = \big\langle \sigma, \varphi, \varphi(d_{\sigma(\omega)}) \big\rangle .$$

The other nontrivial `eval` method is that of cons lists. As described in a previous section, the `eval` method of a list evaluates its first element and then invokes the `call` method of that value with the rest of the list passed as arguments. We denote this as

$$\mu_{\Sigma(\mathsf{CONS})}(\text{“eval”})(\sigma, \varphi, \omega, \langle\rangle) = M(\text{“call”}, \sigma', \varphi', e_1, \langle a_2, \ldots a_n\rangle).$$

where

$$\langle \sigma', \varphi', e_1\rangle = M(\text{“eval”}, \sigma, \varphi, a_1, \langle\rangle)$$

and $\langle a_1, a_2, \ldots a_n\rangle$ is the tuple representing the contents of $\sigma(\omega)$.

The final method we define here is the critical `call` method of the `msg` object. Recall that the `msg` form allows the user to invoke methods explicitly, and it is the `call` method of that form object that exposes this functionality

$$\mu_{\Sigma(\mathsf{MSG})}(\text{“call”})(\sigma, \varphi, \omega, \langle a_1, \ldots a_n\rangle) = M(d_{\sigma'(a_2)}, \sigma', \varphi', e_1, \langle a_3, \ldots a_n\rangle),$$

where $\langle \sigma', \varphi', e_1\rangle = M(\text{“eval”}, \sigma, \varphi, a_1, \langle\rangle)$.

Additional Features

Today Lisp is over fifty years old, and Common Lisp is almost half that age as well. Apart from adding wholly new features, there is much room for improvement in the traditional syntax of the language, in particular by reducing the sheer number of parentheses that the user needs to type. We list here four such attempts that we have included in Yalie. The first two are inspired by Paul Graham's work on the Arc dialect of Lisp[3], and the others are our own. Each is described in the accompanying README file.

First, the functionality of the `cond` expression is incorporated into `if` without changing the basic structure of an `if` call. In particular, `if` can be made to take an arbitrary number of test-consequence pairs, with an optional unpaired expression interpreted as an alternative. For example:

```
(if test1 a
    test2 b
    test3 c
          d)
```

Second, parentheses inside the bindings of a `let` expression are simply dropped. This gives the cleaner syntax:

```
(let (a 1
      b 2)
  (+ a b))
```

The meaning of `let` is also radically extended in a different way. A `let` expression with a single binding and no body can create a local binding in the calling scope that persists until the end of that scope. This allows the programmer to create a local variable without adding a level of nesting and indentation, which can greatly improve readability. When doing this, a further pair of parentheses is omitted. Thus, the following is valid:

```
(let a 1)
(+ a 2)
```

As implemented in Yalie, `let` can create a new binding in this way but cannot modify an existing one. (The `set` function exists for that.) The purpose of the distinction is to separate variable declaration for variable modification, so that most typographical errors in variable names will lead to a program error instead of a silent bug.

Finally, the functionality of the `not` operator is be extended to include multiple arguments. When receiving more than one argument, not interprets all its arguments as a function call and returns that call's negation. For instance, the following returns a true value:

```
(not = 0 1)
```

In many common cases, this both increases readability and saves the user two parentheses.

Conclusions

Several different interpreters were created at different stages in this project, but the "call" and "eval" semantics as described in this paper were only conceived for 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-based 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 facilities are 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 accompanying README) demonstrates the flexibility of the object-based 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.

[3] P. Graham. *Arc at 3 Weeks.* November 2001. Web. 8 May 2009. <http://www.paulgraham.com/arcll1.html>.