Yet Another Lisp Interpreting Experiment

John O'Connor

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 create domain-specific mini-languages on top of a full Lisp 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." Python and Ruby languages have adopted many concepts from Smalltalk with great success. Though several object-oriented variants of Lisp have been developed, including the Common Lisp Object System, these have not achieved nearly the same widespread use. 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 both flexibility and 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 objects can greatly enhance Lisp's core strengths, but only if these concepts are introduced consistently throughout the entire language.

This is the goal of Yet Another Lisp Interpreting Experiment (or, "Yalie"). We present a variant of Lisp 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 practical description with examples, please see the accompanying README file, reprinted here in the appendices. 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 common are "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 in these languages inheritance is accomplished within the class hierarchy. Object-based languages omit this distinction, and instead allow objects to inherit directly from other objects and to produce their own clone or child objects. In practice, object-based languages often approximate classes by using certain objects

solely for inheritance. For this reason, and because of the wider popularity of class-based languages, some have argued that making classes an explicit part of a language is more convenient [2]. One of our primary purposes 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 that merely use objects as the fundamental unit of their semantics. Yalie is an object-*based* language, and it is not our intention to abandon the functional and procedural paradigms traditional in Lisp languages. OOP will of course be fairly natural in Yalie, but our goals of simplicity, consistency, and flexibility should benefit any programming style.

Message Passing

As a demonstration of the consistency possible with objects, we base all of the semantics of Yalie 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 (i.e. a series of cons objects with a nil object at the end) 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 to receive 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 a function or form in Yalie is just an ordinary object with a defined `call` method. We will construct a few predefined functions and forms in Yalie, and these will be ordinary objects whose `call` methods we define usefully.

Another key difference in this function is the way `+` is defined, which is not immediately visible. Rather than being a built-in operation that knows about integers, `+` simply invokes a method of its arguments. This method is also named `+`, though it resides in the integer method table rather than in the lexical scope as the `+` function does. One of the 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 designed to pass messages and invoke methods. When that `call` method is invoked, it evaluates its first argument, the 1 object, and invokes the method of that first argument named by the second argument, passing any further 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, which we call "pointers." We denote scopes by the variable φ, with the global scope—that is, the built-in 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 $[\sigma|\text{key} \rightarrow \text{val}]$. We will frequently refer to the pointers to certain built-in 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. Likewise, the symbol `foo` would be represented as

$$\langle \mathsf{SYMBOL}, \mu_0, \varphi_0, \text{“foo”} \rangle .$$

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 $\sigma(\omega)$. Similarly, we will use $p_{\sigma(\omega)}$ and $d_{\sigma(\omega)}$ to refer to the parent pointer and underlying data element of $\sigma(\omega)$. Note that the the root object does not have a parent, i.e. $p_{\Sigma(\mathsf{ROOT})} = \varnothing$.

We have already mentioned the function $\text{next}(\sigma)$, and we will introduce two more semantic functions here that we will need below. 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, its parent is queried recursively. Character strings representing method names are denoted by the variable m.

$$\text{lookup}(m, \sigma, \omega) = \begin{cases} \varphi_{\sigma(\omega)} & \text{if } m \in \text{Domain}(\varphi_{\sigma(\omega)}) \\ \text{lookup}(m, \sigma, p_{\sigma(\omega)}) & \text{else if } p_{\sigma(\omega)} \neq \varnothing \\ \textit{undefined} & \text{otherwise} \end{cases}$$

In practice, these character strings will either be supplied explicitly by a built-in method definition or extracted from the data field of a symbol object.

The second function we need to define at the moment is the semantic function for invoking methods. We mentioned the `msg` function 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 a pointer to the 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.

$$\text{invoke}(m, \sigma, \varphi, \langle a_1, \ldots a_n \rangle) = (\text{lookup}(m, \sigma, \omega))(\sigma, \varphi, \omega, \langle a_1, \ldots a_n \rangle)$$

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 does not denote an integer. Lists are written as sequences of whitespace-separated literals enclosed by matching parentheses. We define a parsing function that takes an abstract syntax expression and an initial store, and returns a tuple containing a pointer to the new object and the updated store. An integer n is thus parsed as

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

and a symbol s is parsed as

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

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

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

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.

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

where

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

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.

$$[\![-]\!] \; \Sigma \; \Phi = \text{invoke}(\text{“eval”}, \Sigma', \Phi, \omega, \langle\rangle),$$

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

$$\texttt{(a.b c...)} \to \texttt{(msg a b c...)}.$$

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

$$\texttt{(a.b.c d e)} \to \texttt{(msg (msg a b) c d e)}.$$

Another global special form that we define is the `quote` operator, which protects objects from evaluation. We use the 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 style 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 itself parses. Note also that the semicolon is used for splicing in Yalie, rather than for commenting, and it replaces the comma-at operator used for splicing in other Lisps. Comments in Yalie are denoted by a hash mark, as is common in scripting languages.

Built-in 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 define several methods inherited by every object and also a few prominent 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.

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

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 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

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

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

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

Another pair of important methods is the `get` and `set` methods, which query and modify member values. These are not used by any other built-in 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 `let` and `set` methods take two elements, and these are the first methods we have seen 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 the method call. The `let` method is used only when the member variable is currently bound, and `set` is used only to modify an existing binding. Thus, in the case where $d_{\sigma(a_1)}$ is not in the domain of $\varphi_{\sigma(\omega)}$, `let` is defined as

$$\mu_{\Sigma(\mathsf{ROOT})}(\text{“let”})(\sigma,\varphi,\omega,\langle a_1,a_2\rangle) = \left\langle [\sigma'|\omega \to \left\langle p_{\sigma(\omega)}, \mu_{\sigma(\omega)}, [\varphi_{\sigma(\omega)}|d_{\sigma(a_1)} \to e_1], d_{\sigma(\omega)}\right\rangle], \varphi', e_1\right\rangle .$$

where $\langle\sigma',\varphi',e_1\rangle = \mathrm{invoke}(\text{“eval”},\sigma,\varphi,a_2,\langle\rangle)$. In the case where $d_{\sigma(a_1)}$ is in the domain of $\varphi_{\sigma(\omega)}$, `let` is undefined. Likewise, `set` is defined only in that case, and its definition takes the same form as `let` immediately above. These two methods are defined this way in order to separate the creation of new bindings from the modification of existing ones. This is helpful for functional programming, and it also causes many bugs to raise an exception instead of passing silently.

The final method of the root object that we define here 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.

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

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.

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

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

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

where

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

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

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

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

Departures

Today Lisp is over fifty years old, and Common Lisp is already half that age. Aside from adding wholly new features, there is much room for improvement in the traditional syntax of the language. Some common names can be helpfully shortened (`fn` instead of `lambda`, `ls` instead of `list`), and there are several opportunities to reduce the sheer number of parentheses that a programmer needs to type [3]. Here we explain four such opportunities that we have included in Yalie.

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))
```

Yalie also extends the meaning of `let` in a different way. A `let` expression with a single binding and no body creates 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 from variable modification, so that typographical errors in variable assignments will raise an exception instead of silently creating a new variable. Yalie also follows the convention that any non-functional operations be marked with "set" or "!".

Finally, the functionality of the `not` operator is 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.

Objects: The Ultimate Lambda

As both an illustrative example and a demonstration of the flexibility of the object system, we detail the creation of a `lambda` operator. Yalie has such an operator built in for convenience (under the name `fn`), but we show how it can be defined within the language. As a second example, we will also create a factorial method for the integers.

We will need two methods that we did not formally define above, `def` and `deform`. These methods allow the user to define new methods and method forms in a given object. Recall that the difference between a function and a form (or "macro") is that a function implicitly evaluates its arguments while a form implicitly evaluates its return. Methods and method forms work in the same way, except that they are invoked by message passing. The syntax of the `def` and `deform` methods is identical to the syntax of the `define` operator in Scheme, and the return value of both methods is the calling object.

The definition of `lambda` thus reads

```
(let lambda Form.child)
(lambda.deform (call args (rest body))
  `(let (ret Function.child)
     (ret.def (call ;args)
       ;body)))
```

The first expression creates a child object from the object that is the parent of all forms and binds it to the name "lambda". We could have used any object as the parent, but this way the new operator will inherit whatever methods other forms inherit. The second expression defines a `call` method for the newly created object. Recall that `call` methods are invoked when S-expressions are evaluated, and so this method will contain the functionality of `lambda`. The method is a form, so its value is Yalie code to be executed at the end of the call. This code can be confusing, because it is creating a second child object—from the parent of all functions, this time—with its own call method. This new object is the anonymous function that `lambda` returns. Note that the arguments and the function body are just spliced into the call to the `def` method of the new object.

For a final example, the following code would define a factorial method named "!" for all integers. Note that methods and method forms implicitly bind the variable `self` to the calling object.

```
(Int.def (!)
  (if (<= self 0)
      1
      (* self (- self 1).!)))
```

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 visible benefit is that, despite having gutted the entire language to allow the ubiquitous use of objects, the only change the global namespace is the addition 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.

We have demonstrated the flexibility of the object-based approach, and the simplicity 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.

Acknowledgements

Thanks to Paul Hudak for advising this project and to Tristyn Bloom for helping to proofread it.

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>.

Appendix: README

This appendix contains the text of the README file that accompanies the Yalie interpreter.

```
Brief documentation for the Yalie language:

Invocation
----------
./yalie.py
./yalie.py code.y
./yalie.py < code.y

In addition to functions and methods built into
the Python code, Yalie automatically reads in
```

the file "builtins.y" in the current directory and executes the definitions there. This file is included with the Yalie source, and it contains many important functions. Make sure it is in the current directory when running Yalie, otherwise Yalie will abort.

Syntax

```
Integers: 1, 2, -5, 45984375394875945
Symbols: a, b, Foo, !!!<<&&
Lists: (a b c)
```

Note that inputting a symbol will raise an error unless that symbol is defined. (See 'let'.) Inputting a list will also raise an error unless the first element is a function. (See 'ls' and 'quote'.)

[#] begins a comment to the end of the line. ['] or [‘] quote an object to prevent it from being implicitly evaluated, and [,] and [;] are unquote and unquote-splice, respectively. See the "quote" function below. [.] is the method operator. It can operate as an infix without parens, or it can capture an S-expression if placed after the first element.

```
a.b --> (msg a b)
(a.b c d) --> (msg a b c d)
(a.b.c d e) --> (msg (msg a b) c d)
```

Reachable Objects

Root, (Int, Cons, Symbol, Nil, Operator), (Form, Function) All of the above are visible in the global scope. Root is the object at the very top of the hierarchy. Everything in the first tuple inherits directly from Root. The second tuple of objects inherits from Operator in the first tuple.

All of these objects can have methods added or redefined using the existing methods described below. This is the essence of the flexibility of object-orientation. We can change how built-in data types like Int behave.

Functions

```
(dir)
```

Returns a list of everything defined in the current scope. Call it to see the list of all predefined functions and objects.

```
(= a b)
(+ a b c ...)
(- a b c ...)
(* a b c ...)
(/ a b)
(% a b)
<, >, <=, >=, =
```

Arithmetic and comparison functions work as they do in Scheme or Common Lisp. The % function computes the remainder as in Python. All of these functions are defined in builtins.y in terms of the integer methods of the same name. Note that boolean values are just the integers 1 and 0 for True and False.

```
(and a b c...)
(or a b c...)
(not x)
(not f a b c...)
```

"and" and "or" work as they do in Scheme, taking an arbitrary number of arguments. "not" has the additional property that multiple arguments are treated like a function call, i.e. (not = 0 1) is syntactic sugar for (not (= 0 1)).

```
(cons x y)
(car a)
(cdr a)
(setcar a x)
(setcdr a x)
```

Operations on cons cells work as in Scheme. Again, these functions are just defined for convenience. The semantics are contained within the methods that these functions invoke.

```
(if bool conseq alt)
(if a b
```

```
    c d
    e f
      g)
```

As in Scheme. "if" can also accept more arguments, as shown in the second example, in which case it behaves like Scheme's "cond". The second example will return b if a, c if d, e if f, or else g. If all tests are false and no unpaired last element is supplied, nil is returned.

```
(while test
   body1
   body2
   ...)
(break)
(continue)
```

Runs a while loop as in Python. The (break) and (continue) functions will result in errors if called outside of a while context. However, they can be nested within other function calls, as long as a while loop is above them somewhere in the call stack when they are invoked.

```
(len list)
(ls 1 2 3 4)
(append '(1 2 3) '(4 5 6))
```

These list functions work as in Scheme, except that "list" has been renamed "ls". (Both for brevity and to free up the variable name.)

```
(map fn list1 list2...)
```
Returns the list of return values of "fn", with each call taking one arguments from each of the passed lists. Thus, the number of lists passed to "map" should be equal to the number of arguments "fn" takes, and all the lists should be of equal length.

```
(filter test list)
```

Returns the list of those elements of "list" that return true when passed as arguments to the function "test".

```
(in x list)
```

Returns 1 if x is an element of "list", otherwise 0.

```
(def (f x)
  body1
  body2
  ...)
(def (foo x y (rest z))
   x.print
   y.print
   z.print)
(deform (nor a b)
  '(and (not ,a) (not ,b)))
(deform (with var val (rest body))
  '(let (,var ,val)
     ;body))
  # like "let", but omits the initial
  # parens and only binds one variable
(def! ...)
(deform! ...)
```

Function and form (or "macro") definitions can be specified with an arbitrary number of arguments. If a variable number of arguments is desired, the final argument should be specified as (rest x), where x is any variable name. Forms don't implicitly evaluate their arguments, and instead their return values are implicitly evaluated. This allows users to construct new binding forms or control structures in terms of existing ones, as shown above. The anonymous counterparts to "def" and "deform" are "fn" and "form", respectively. "def" and "deform" are not allowed to modify existing bindings. Use "def!" and "deform!" for that.

```
(quote x)
```

When quote is evaluated, it returns x without evaluating x. This is helpful for passing symbols or lists as arguments since they do not evaluate to themselves. With the splice operators (immediately below), the quote tool is a fantastic way to put together the return value of a macro.

```
  `(a b c)  ==  (quote a b c)  -->  (a b c)
  `(a b ,(ls 1 2))
      == (quote a b (unquote (+ 1 1)))
```

```
        --> (a b (1 2))
  `(a b ;(ls 1 2))
      == (quote a b (unquote-splice (ls 1 2)))
         --> (a b 1 2))
```

```
(do  body1
     body2
     body3...)
```

Simply evaluates each argument and returns the last. Equivalent to Common Lisp's "progn".

```
(call f ...)
```

Calls the function "f" with a supplied list of arguments. If the final argument to "call" is a list (or nil), the contents of that argument are spliced into the list of passed arguments, rather than being passed as a single argument. "Call" is a function, so it's arguments are evaluated implicitly, but it suppresses the further implicit evaluation of arguments that would be carried out by "f". Several special forms evaluate some of their arguments implicitly and other arguments explicitly, and the distinction can be confusing. Special forms with implicitly evaluated arguments are all of the "let" and "set" forms and method forms (though not the body of a "let" form, if it has one), "if", "and", "or", "not" (in the single argument case only), "msg" (the recipient object), and "expand" (same as "msg"). Note in particular that evaluation done by "quote" and "while" forms is always considered explicit and never suppressed.

```
  (call + 1 2 3)      -->   6
  (call + (ls 1 2 3))   -->   6
  (call + 1 2 (ls 3))   -->   6
```

```
(let (a x
      b y
      c z)
  body1
  body2
  ....)
```

Works similarly to let* in Scheme, but omits the parentheses around assignment pairs.

```
(let a 9)
```

It is also possible to define a local variable that persists for the duration of the calling scope by omitting a body from let. This can only be done for one variable at a time, and the grouping parentheses are dropped.

```
(set a x)
```

Modifies an existing binding. In order to clearly distinguish functional from non-functional operators, (let a x) is not allowed to modify an existing binding if one already exists for 'a' in the current scope. (Though obviously, if provided a body, let will create its own scope without any problems.) Set should do that instead. Set can also be used, in conjunction with "fn" or "form", to modify function bindings. On the other side, set is not allowed to create a new binding. If no variable named 'a' exists in the previous example, set raises an exception. Separating declaration from modification is intended to eliminate silent bugs that result from misspelling, and to facilitate functional programming.

```
(msg obj message arg1 arg2...)
```

This function is how the user sends messages to objects. It is usually invoked using the period syntax.

```
(expand obj message arg1 arg2...)
```

"expand" is very similar to "msg", but it only works when the method specified by "obj" and "message" is a method form defined in Yalie. (Anything else raises an exception.) In that case, rather than evaluating the result of the method call, "expand" returns the code that would have been evaluated when the method returned, that is, the unevaluated return value of a method form. To see the expansion of a form defined with "deform", one would use something like (expand my-form call arg1 arg2...). For a simple example:

```
  (deform (let-x val) `(let x ,val))
  (let-x 5)
     --> 5 (and 5 is now bound to x)
  (expand let-x call 5)
     --> '(let x 5)
```

```
(error args...)
```

Raises an exception. The arguments don't actually do anything, but they show up in the resulting error traceback, so they are useful for commenting the error.

A few other functions like (eval, print, eq, is) are simply defined as wrappers around the appropriate method calls.

Methods

The most important methods are those of the root object, since all other objects inherit them.

```
x.child
x.copy
```

Return new objects that either inherit from or duplicate x, respectively

```
(x.get key)
(x.let key val)
(x.set key val)
```

Get returns the value of the member named key, or raises an error. Let binds a new member variable, and set changes the value of an existing member. NB: member and method scopes are separate, and members that happen to be functions cannot be invoked using method syntax.

```
(x.def (name args...) body...)
(x.deform (name args...) macro_body...)
```

These work similarly to the global "def" and "deform" constructs above, except that they bind methods to an object rather than functions to the local scope. These cannot be used to redefine local methods, i.e. those methods not inherited from a parent object. (See "def!", "deform!", and "dup" below). All user defined methods bind the variable "self" implicitly in their bodies. For example, to give all integers a factorial method called "!", we would write:

```
  (Int.def ! ()
    (if (= self 0)
        1
        (* self (- self 1).! )))
        # Now even literal integers
        # like "5" have a ! method.
```

```
(x.def! (name args...) body...)
(x.deform! (name args...) macro_body...)
```

As above, except that these methods can redefine an existing binding. The distinction is in place to facilitate functional programming: all non-functional operations are marked with either "set" or "!". See dup below.

```
(x.dup foo bar)
(x.dup! foo bar)
```

Copies a method within x named foo to bar. Note that even if foo is inherited, bar will be a local method of x. Yalie has no "super" keyword of any kind, so to redefine a method while continuing to invoke the old definition, first dup the old version to another name. "dup" cannot modify an existing binding, i.e. there can be no method named bar when dup is invoked. "dup!" is supplied for that purpose. Both functions return the calling object.

```
(x.isa y)
```

Returns a true value if x inherits from y, otherwise a false value. Again, booleans are just the integers 1 and 0.

```
x.parent
```

Returns the parent object of x. For Root, this returns an error.

```
(x.is y)
```

```
Returns true only if x and y are the same object
in memory.

(x.eq y)

For most objects this is the same as "is". For
symbols, integers, and lists, however, "eq" is
defined by value. In the case of lists, that
definition is recursive. Two lists are equal if
their contents are equal. Note that integers
have an "=" method that identical to their "eq"
method, except that "=" raises an exception when
"y" is not an integer. When comparing integers,
"=" can catch bugs that "eq" would miss.

x.print

Prints a representation of x to stdout, with a
newline.

x.members
x.methods
x.methods*

Return the lists of members or methods of x, as
a list of symbols. "methods*" lists only those
methods that are local to x, omitting inherited
methods.

x.eval

For most objects, this just returns x. For
symbols, this returns the value of that symbol
in the current scope. For lists, it evaluates
the first argument and then passes the rest of
the list to that argument's call method. (i.e. a
function call) The following produce the same
result, though the second takes an extra step.

  (+ 1 2 3)           # invokes +.call implicitly
  (msg + call 1 2 3)  # invokes +.call explicitly
                      # (by invoking msg.call
                      # implicitly)

(x.call a b c...)

Defined for things that can be called with
function syntax. Redefining this method changes
the behavior of the function. Evaluating a list
implicitly calls this method from the first
element of that list.

Other objects have specialized methods, and many
global functions are just wrappers around these
methods. Integers, for example, define methods
for all of the arithmetic operations and binary
relations. Cons objects define car, cdr, setcar,
and setcdr methods. If any of these methods are
modified, the change will be reflected in the
behavior of the wrapper functions using them. To
see a list of an objects methods, use the
"methods" method.
```

Appendix: builtins.py

This appendix contains the text of the file "builtins.py". This file defines any functions or forms not defined in the Python code.

```
### Anonymous function and form constructs ###
(deform (fn args (rest body))
 `(let ()
    (def (anon ;args)
     ;body)))

(deform (form args (rest body))
 `(let ()
    (deform (anon ;args)
     ;body)))

### Various useful operators ###
(def (eval x) x.eval)
(def (print x) x.print)
(def (eq a b) (a.eq b))
(def (is a b) (a.is b))

(def (do (rest rest))
     (if (rest.isa Nil) ()
         (rest.cdr.isa Nil) rest.car
         (car (while (not (rest.cdr.isa Nil))
                     (set rest (rest.cdr))))))

### List operators ###
(def (ls (rest rest)) rest)
(def (car x) x.car)
(def (cdr x) x.cdr)
(def (setcar x y) (x.setcar y))
(def (setcdr x y) (x.setcdr y))
```

```
(def (len x)
     (if x
         (1.+ (len (cdr x)))
         0))

(def (reverse list)
     (def (helper list rest)
          (if list
              (helper (cdr list)
                      (cons (car list) rest))
              rest))
     (helper list ()))

(def (append (rest rest))
     (if (rest.isa Nil) ()
         (rest.cdr.isa Nil) rest.car
         (rest.car.isa Nil) (call append
                                  rest.cdr)
         (cons rest.car.car
               (call append rest.car.cdr
                           rest.cdr))))

(def (map fn (rest arg-lists))
     (def (map-single fn list)
          (if list
              (cons (fn (car list))
                    (map-single fn
                                (cdr list)))))
     (if (not arg-lists)
         (error arguments must be provided))
     (if (car arg-lists)
         (let (args (map-single car
                                arg-lists)
               rests (map-single cdr
                                 arg-lists))
           (cons (call fn args)
                 (call map fn rests)))))

(def (filter fn list)
     (if list
         (if (fn (car list))
             (cons (car list)
                   (filter fn (cdr list)))
             (filter fn (cdr list)))))

(def (in obj list)
  (if (not list)         0
      (= obj (car list)) 1
      (in obj (cdr list))))
```

```
### Arithmetic operators ###
(def (= a b) (a.= b))
(def (< a b) (a.< b))
(def (<= a b) (or (< a b) (= a b)))
(def (> a b) (and (not < a b) (not = a b)))
(def (>= a b) (or (> a b) (= a b)))

(def (+ (rest rest))
     (if rest
         ((car rest).+ (call + (cdr rest)))
         0))

(def (* (rest rest))
     (if rest
         ((car rest).* (call * (cdr rest)))
         1))

(def (- x (rest rest))
     (if (not rest)
         (* -1 x)
         (x.- (call + rest))))

(def (/ a b) (a./ b))
(def (% a b) (a.% b))
```