Metalinguistic Abstraction in Smalltalk-80
(or: Making Smalltalk with a Lisp)

Ian Piumarta
Department of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL
England

Internet: `ikp@cs.man.ac.uk`
UUCP: `mcsun!uknet!man.cs!ikp`

Abstract

The Smalltalk-80 language provides the programmer with latent types, polymorphic procedures, first-class closures, and an immense amount of access to the runtime structures of the Smalltalk environment. As such, it can be an ideal vehicle for experiments in metalinguistic abstraction.

This paper describes some techniques for establishing embedded languages within the Smalltalk-80 system. These techniques are illustrated by applying source-level transformations to Lisp expressions. Procedure definitions are transformed into equivalent Smalltalk method definitions, and Lisp declarations into equivalent mutations of a localised Smalltalk environment in which those methods are executed.

Smalltalk can easily provide most features of Lisp systems, including the first-class continuations of Scheme and related languages, because of the liberal access provided by Smalltalk to the runtime data structures of the virtual machine. This liberality allows the implementation of additional language features beyond those intrinsically provided by Smalltalk, in Smalltalk itself.

Keywords: metalinguistic abstraction, embedded languages, compilation, source-level transformation, Lisp, Smalltalk.

1 Introduction

The notion of metalinguistic abstraction will be familiar to anyone who has read the excellent introductory computer science text "Structure and Interpretation of Computer Programs" [ASS85] in which both an interpreter and compiler for Scheme are developed, using Scheme as the implementation language.

Metalinguistic abstraction is the process of creating an 'embedded language' — a descriptive language that is built within the framework of a similar descriptive language. It applies to a particularly narrow range of 'translators' and 'evaluators' in which the semantic gap between the embedded and base languages is relatively narrow. Compiling (or evaluating expressions of) the embedded language should involve merely transforming its constructs into equivalent constructs of the base language without having to implement any significant features of the embedded language directly. The syntax of the embedded and base languages need not be particularly close, and the example of implementing Scheme in Scheme is misleading in this respect.

Exploratory languages are those in which data and procedures can be defined, modified and inspected incrementally by the programmer in an interactive environment. They do not impose the need to compile (or link) entire programs atomically and then restart them every time a change is made to the definition of some procedure or value. It is to this class of languages in particular that metalinguistic abstraction can be applied most successfully, providing the opportunity for very rapid experimentation with new languages, paradigms, execution models, and so on (see [Hop??] for a good example).

This paper is about Lisp and Smalltalk, languages which at a glance may appear to have very different semantics — but with a little thought it becomes apparent that this is not the case. Both symbolic processing languages, both have lexically-scoped mutable state with late binding of procedure names to procedure bodies, both provide anonymous (unbound) procedures as full closures, both use so-called "applicative-order" evaluation of procedure arguments, and both rely heavily on transparent management of dynamically allocated data structures. Of the two, Smalltalk has the more complex semantics [Wol88] providing equivalents for just about every feature found in Lisp systems and, although rather unnatural, it is entirely possible to write Smalltalk programs in a purely functional style. (The semantics of Scheme, a relatively small and 'clean' dialect of Lisp, are given in [StS??, section 7.2].)

2 Metalinguistic Abstraction in Smalltalk

There are many issues to be addressed when establishing an embedded language. From fine details about the representation of structured data and execution context in the embedded language, to pervasive organisational decisions such as whether to implement an evaluator that interprets or compiles.

2.1 Interpreter or Compiler?

Embedding some descriptive language within another ulimately requires one of two things: either an interpreter which performs the runtime actions of embedded programs on their behalf, or a compiler which transforms constructs of the embedded language into equivalent constructs of the base language. The choice of which of these to use depends largely on the gap between both the semantics and execution model of the base and embedded languages.

Whichever implementation strategy is adopted, some structured representation of the embedded program will be necessary. A parser for the embedded language will be required whose structured output will either be interpreted to implement the actions implied by the structure of the program, or traversed to generate an equivalent in the base language. Complexity is not an issue since an interpreter can be seen as a compiler which evaluates expressions as they are translated, discarding the translated form immediately thereafter rather than saving the translated form permanantly for re-evaluation on demand. Since exploratory languages generally provide easy access to their compiler from within programs, translated procedures can be installed in the environment as if they were hand written directly in the base language.

The only genuine reason for adopting an interpretive approach would seem to be the existence of a significantly large gap between either the semantics or execution model of the base and embedded languages. For example, if the semantics of function application were very different in the two languages (maybe the base language uses applicative-order and the embedded language normal-order evaluation of arguments) it might be necessary to implement the entire application mechanism in an interpreter written in the base language.

2.2 Data Types

Smalltalk provides a rich variety of classes for what would in other languages be considered data types. Common types such as Arrays are of course available, as well as many other kinds of Collection. Dynamically-sized data structures found in languages such as Algol can be modelled using existing Smalltalk classes (such as OrderedCollection). The structures (or records) and unions of languages like C and Pascal can be constructed using the class mechanism, and declarations for these in any language can be converted easily into equivalent class descriptions (with the attendant accessing methods). Such a class, when instantiated, will yield a datum of a type equivalent to that of the structure or union in the embedded language.[1]

'Primitive' data types, such as Lisp's cons cells, can be provided using either of these mechanisms. A trivial subclass of Link yields Cons, and a handful of iteration methods defined there can endow its instances with the usual collection-oriented behaviour associated with Lists composed of such `cons` cells. Alternatively, the implementation's representation of lists need not adhere strictly to the model used by the embedded language: an equally valid approach would be to have lists as a subclass of OrderedCollection, with the primitives of the embedded language providing transparent conversions between this representation of lists and the more usual cons cell model.

2.3 State, Context and Namespace

Most programming languages provide both global and local state. Local state is easily accommodated in Smalltalk, which provides statically scoped local variables within both methods (named procedures) and blocks (anonymous procedures). Global state, which should be visible in all procedures defined in the embedded language, needs more thought.

Most Smalltalk programmers keep only a few images (often just one) in which they do all their development work. It follows that any global variables defined in the embedded language

[1] This approach is used in ParcPlace's 'Objectkit\Smalltalk C Programming', which includes tools for generating classes based on structure definitions read from C header files.

should not interfere with the operation of the rest of the image. Smalltalk provides many different kinds of variable, with greatly varying ranges of scope. Mapping global state in the embedded language directly onto Smalltalk's global variables is unacceptable, since this would interfere with (and possibly even damage irrevocably) the rest of the image. Mapping it onto local variables is also impossible, since the Smalltalk code implementing the procedures of the embedded language could not be compiled (at least not by the standard Smalltalk compiler) within the local environment of a method or block. Fortunately, there are also class variables which are visible to any method or block defined in a particular class. A class can be created which will be responsible for evaluating programs in the embedded language. If the procedures of the embedded language are all compiled in the context of that class, their global state can reside in its class variables. Variables occurring free in the body of an expression or procedure will be resolved first in the class variables of the compilation context, and then in Smalltalk's global context (giving access to Smalltalk's own global variables and classes). Since class pools (which hold the class variables) in Smalltalk are simply collections of Associations (mapping identifiers onto values), the maintenance of the embedded global namespace is trivial.[2]

Maintaining the global context of our embedded language can be made easier by defining some convenience methods which provide the usual Collection protocol for the class pool. For the sake of illustration let's call our embedded language 'foo', and the class responsible for maintaining the global state 'FooContext':

FooContext class>>globalContext
    ↑classPool

FooContext class>>at: aSymbol ifAbsent: aBlock
    ↑classPool at: aSymbol ifAbsent: aBlock

Definitions for at:, at:put: and removeKey: (and maybe others that might come in handy) follow the same pattern as that given for at:ifAbsent:.

2.4 Functionality

Bound procedures in the embedded language are transformed into equivalent Smalltalk method definitions. Smalltalk provides a simple mechanism by which any class can be asked to compile and install a method. Sending any class the message compile: aString notifying: aRequestor ifFail: failBlock will compile aString as a method definition, in the context of the class receiving the message (and therefore with its class variables appearing global to the method), and then install it in the receiver's method dictionary.[3]

Anonymous procedures in the embedded language are transformed into equivalent block-valued Smalltalk expressions. Again, there is a simple interface to the compiler that allows these expressions to be evaluated in the context of any desired class. An instance of Compiler responds to evaluate: textOrStream in: aContext receiver: aReceiver notifying: aRequestor ifFail:

[2]Metaclass instance variables cannot be used since extending the global namespace of the embedded language would be tricky (to say the least), requiring the recompilation of all class methods every time the namespace changed (because class methods are metaclass instance methods). Variables bound in pool dictionaries have potentially too large a scope if every procedure in the embedded language is to be compiled in the context of the same class, and therefore have no inherent benefits over class variables.

[3]aRequestor is the agent from which the compilation was initiated — usually some kind of TextEditor. failBlock is executed if the compilation is aborted due to some kind of error.

failBlock with the value of the expression in the textOrStream, compiled in the context of aContext and aReceiver,[4] with the last two arguments having the same meanings as before.

Assuming we have built a class in which to evaluate programs in our embedded language, we can again define a couple of convenience methods in this class for procedure-related activities which will be useful no matter what language we are abstracting. The first takes a String containing a Smalltalk expression and returns the value of that expression evaluated in the global context of the embedded language. It will usually be handed a block expression in order to create an unbound procedure:

```
FooContext>>evaluateSmalltalk: aString
    ↑Compiler new
        evaluate: aString
        in: nil
        receiver: self
        notifying: nil
        ifFail: [self error: 'Foo compiler failed!']
```

The second takes a String containing a Smalltalk method definition which is compiled in the global context of the embedded language and then installed as a bound procedure:

```
FooContext>>function: aString
    self class
        compile: aString
        notifying: nil
        ifFail: [self error: 'Foo compiler failed!']
```

If the transformations from the embedded language to Smalltalk are correct, neither of these methods should ever raise an error.

Some attention must be paid to procedures in the embedded language which expect arguments. Smalltalk is probably unique in using an infix notation where method arguments are embedded within the message name. If the embedded language uses atomic procedure names which prefix a list of arguments (as most languages do), then some conversion between that representation and Smalltalk's is necessary.

Procedure names can be carried over from the embedded language to Smalltalk directly, for so-called "thunks" (procedures with no arguments). Appending a ':' to the procedure name caters for monadic procedures. Names for procedures of two or more arguments can be transformed into Smalltalk's infix notation by prepending a dummy keyword component '_:' before the second and subsequent arguments.[5] Some typical transformations would be as follows.

`newline()`	→	self newline
`nfibs(42)`	→	self nfibs: 42
`takeuchi(18, 12, 6)`	→	self takeuchi: 18 _: 12 _: 6

[4]aContext provides the class context, and aReceiver provides values for any instance variables named in textOrStream.

[5]This trick will not work in versions of Smalltalk earlier than Release 4.1.

The underscore is a good choice for the dummy keyword component since it is normally an illegal character and its promotion to alphabetic status in no way affects the operation of the rest of the Smalltalk system.[6]

2.5 Translation

There are two possible approaches, and the choice depends largely on the complexity of the embedded language and the familiarity of the implementor with Smalltalk's own compiler.

Parsers for languages with small grammars can easily be built by subclassing the Smalltalk compiler. The Smalltalk system includes several reusable compiler classes, including the Scanner and Parser used by the Smalltalk compiler itself. By subclassing these two classes, scanners and parsers for a wide variety of 'languages' can be constructed. In a similar manner, high-quality user interfaces (workspaces, file editors, and the like) can be subclassed from those used by the Smalltalk user interface, and by overriding a handful of methods versions specialised for languages other than Smalltalk can be built easily.

For languages with more complex grammars, or languages that have features that are difficult for Smalltalk's compiler to cope with, there are tools available that build scanners and parsers from token and grammar specifications in the manner of `lex` and `yacc`.[7] The most widely available of these is T-gen [Gra??], which follows the `lex` and `yacc` tradition very closely. It comes with ample documentation, and anybody familiar with the UNIX compiler construction tools should be able to construct a working translator for a small grammar in a matter of hours.[8]

Having built an abstract syntax tree (or equivalent structured representation of the source program) this tree can be walked in the usual fashion. Nodes representing variable declarations need only mutate the state of the embedded global context using messages like at:put sent to the context-holding class. Nodes representing particular constructs of the embedded language rewrite themselves as equivalent Smalltalk expressions, including the rewritten forms of their children where appropriate, and return their result to their parent as a contribution to its rewritten form. Near the top of the AST, nodes representing procedure definitions send messages such as compile: to the context class, passing it the rewritten forms of procedure definitions for compilation and installation.

Expressions written in the embedded language can be transformed, and the result evaluated using the evaluateSmalltalk: message described earlier.

3 Implementing Lisp in Smalltalk

As a concrete example of metalinguistic abstraction in Smalltalk, the remainder of this paper presents a simple implementation of Lisp built using the techniques described so far.

---

[6]In early versions of Smalltalk the underscore character was used to represent the assignment operator, and was displayed as '←'. When this usage was dropped in favour of :=, the underscore was made an illegal character. A trivial change to the Smalltalk Scanner allows its use as part of a keyword selector.

[7]Smalltalk has a recursive descent Parser which can easily cope with its LL(1-and-a-bit) grammar. Parsers for more complex grammars requiring two (or more) token lookahead, or fully-fledged backtracking, cannot be built by subclassing Smalltalk's Parser.

[8]T-gen is available via anonymous FTP from both the Manchester and UIUC archives. See section 7 for details.

Lisp has such as simple syntax that a scanner and parser can be built easily by subclassing the Smalltalk Scanner and Parser classes. Since data and programs in Lisp share the same structure, the LispParser need only be capable of reading a Lisp s-expression. A handful of new classes are needed to represent the data types of Lisp objects (symbols, lists, constants, and so on) and a structure based on these is the result of parsing an expression. Each of these classes understands the message generate, and when sent this message will return a string representing the equivalent Smalltalk expression (as described below). The most complicated generate method is defined in Cons which has to cope with rewriting all of Lisp's special forms

Most of the transformations are straightforward but there is one important choice to be made concerning the representation of Lisp procedures, the semantics of which are sufficiently different to those of Smalltalk's methods to warrant special attention.

3.1 Lisp Procedures: Blocks vs. Methods

Lisp provides a single model for all executable objects. A procedure definition is treated (conceptually, at least) as a lambda function bound to a global variable. This binding is dynamic in the sense that the variable can be rebound to a different procedure at any time, either globally or locally (by shadowing it with a variable of the same name bound to a different procedure). In Smalltalk the situation is quite different. The dynamic binding is similar in that the redefinition of a method will be felt globally, and immediately. However, Smalltalk's dynamic binding also overloads each message selector — the method that will be entered as the result of sending a message depends not only on the current bindings between message selectors and methods, but also on the class of the first argument (the receiver) of the message.

This has two implications for an embedded Lisp implementation. Smalltalk's more powerful two-dimensional view of dynamic binding must be flattened to conform to Lisp's one-dimensional model. Slightly more problematic is the lexical scoping of procedure bindings — a program must be able to bind or rebind identifiers to procedure bodies as easily as it would any other first-class value. The effects must be felt immediately and the bindings must obey lexical scoping rules; this behaviour has no parallel in Smalltalk's treatment of methods.

Smalltalk provides two kinds of executable object. Blocks are closely related to lambdas, and can be treated with the same first-class abandon. Bound methods behave quite differently, and it would take some stretching of the imagination (and/or some very unsanitary Smalltalk code) to call them first-class values.[9, 10]

The choice of which procedural representation to use is inevitably a compromise between the semantic integrity which can be provided by blocks and the better performance achieved when using compiled methods. The differences in the two mechanisms are contained entirely within the abstractions used for procedure definition and application, and it is not difficult to construct an evaluator which can switch between these representations easily. For this reason,

[9] Reaching inside a MethodDictionary and changing a binding by brute force may or may not work, depending on whether the method is currently in the virtual machine's translated method cache — the CompiledMethod in the image is only consulted when the method first enters this cache. The primitive flushVMmethodCacheEntriesFor: aSelector can be used (by the Smalltalk compiler, for example) to force changes to take effect immediately, but this still only rebinds a method in the *global* context; providing lexical scoping of methods in this manner is a much more difficult problem.

[10] Strictly speaking it is the binding mechanism used to associate selectors with methods which causes compiled methods, when regarded as the behavioural value associated with a message send, to lose their first-class status. A CompiledMethod can be activated anonymously using the performWithReceiver:arguments: primitive, which is to CompiledMethods what valueWithArguments: is to BlockClosures.

and because a comparison between the relative performance of the two approaches is in itself interesting, both strategies will be described.

3.1.1 Procedures as Methods

3.1.2 Procedures as Blocks

3.2 Transforming Lisp into Smalltalk

There are many dialects of Lisp available, but all of them share some common features. The following sections briefly describe the constructs that would be provided by any language claiming to be a dialect of Lisp, along with possible equivalents in Smalltalk.

The implied types of the variables used in the rewrite rules are as follows:

ε, ν	an arbitrarily complex expression
λ	a literal value
σ	any symbol
α	a symbol appearing in an argument position (a formal parameter name)

For the purposes of the rewrite rules these form a type hierarchy, with each type in the table containing all those below it.

The rewrite rules borrow some of the notation used by the denotational semanticians. $\mathcal{E}[\![x]\!]$ represents the rewrite (or 'denotation') of the Lisp expression x. Most denotations are defined recursively in terms of $\mathcal{E}[\![x_i]\!]$, where x_i are the subexpressions of x.

The operator $\mathcal{I}[\![x]\!]_\mathrm{s}$ is used to represent an imperative operation.[11] The Smalltalk argument x is evaluated at the time at which it is encountered during a rewrite. The denotation of $\mathcal{I}[\![x]\!]_\mathrm{s}$ is empty; it is used purely for its side effect, and makes no contribution to the denotation of any expression in which it occurs. This implies a temporal ordering of the rewrite rules which is easy to see intuitively from the rules, but is a significant departure from the usual interpretation of denotational expressions.

One additional operator $\mathcal{S}[\![x]\!]$ is used for quoted literals. The argument is a Lisp s-expression, and the result is a Smalltalk expression which evaluates to an equivalent literal. For example, a 'quoted' list might be rewritten as an Array:

$\mathcal{S}[\![$`(1 2 (foo 42) bar)`$]\!]$ = #(1 2 #(#foo 42) #bar)

The representation of literal data such as this depends on the strategy adopted in the primitives, and does not affect the structural transformations in any way.

Anything else occurring in the rewrite rules is interpreted literally. On the left hand side of a rule it will be matched directly against the expression being rewritten. On the right hand side of a rule, it will occur literally in the rewritten expression. The result of rewriting any Lisp expression is a semantically equivalent Smalltalk expression.

[11] The subscript is to indicate that the argument type is a Smalltalk expression, rather than Lisp.

3.2.1 Imperative Operations

For declarations it will be necessary to perform certain operations as a side-effect of rewriting a term. The imperative operator $\mathcal{I}[\![x]\!]_s$ represents the evaluation of the Smalltalk expression x, and makes no contribution to a rewritten term.

$$\mathcal{I}[\![x]\!]_s =$$

3.2.2 Literals and Symbols

Literals (mainly numbers) in Lisp and Smalltalk have the same representation:

$$\mathcal{E}[\![\lambda]\!] = \lambda$$

Symbols are also preserved unaltered:

$$\mathcal{E}[\![\sigma]\!] = \sigma$$

However, note that the semantic action associated with a symbol in Lisp is to replace it with its value. This behaviour is preserved since (for variables) Smalltalk does the same thing. Lisp does the same thing with a symbol appearing in the head position of a list (a procedure application), whereas Smalltalk treats such symbols as selectors rather than variables. A separate mechanism is employed to handle this, as will be explained in sections 3.2.3 and 3.2.8.

3.2.3 Global Definitions for Variables and Procedures

Global variables are bound (or rebound) in the class pool of LispContext. If a value is specified its rewritten form is evaluated in the context of LispContext and the result used, otherwise nil is used:

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(define }\sigma\texttt{)}]\!] = \\
&\quad \mathcal{I}[\![\text{LispContext at: } \sigma \text{ put: nil}]\!]_s \\
&\quad \sigma
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(define }\sigma\ \varepsilon\texttt{)}]\!] = \\
&\quad \mathcal{I}[\![\text{LispContext at: } \sigma \text{ put: } \mathcal{E}[\![\varepsilon]\!]]\!]_s \\
&\quad \sigma
\end{aligned}$$

Procedure definitions are slightly more complicated, since we must take care to cater for the use of procedures in two cases: they are applied due to either a symbol *or* a more complex expression in the head position of a list. In the former case a method, and in the latter a block closure, will be applied. This can be achieved by adding both an instance method to, and a binding of a variable of the same name in the class pool of, LispContext.

One solution is to compile the functionality into the method, and have the corresponding block simply invoke that method with a send to self:

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(define (}\sigma\ \alpha_1\ \alpha_2\ \ldots\ \alpha_i\texttt{)}\ \varepsilon_1\ \varepsilon_2\ \ldots\ \varepsilon_i\texttt{)}]\!] = \\
&\quad \mathcal{I}[\![\text{LispContext compile: '}\sigma\text{: } \alpha_1 \text{ _: } \alpha_2 \ldots \text{ _: } \alpha_i\ \mathcal{E}[\![\varepsilon_1]\!].\ \mathcal{E}[\![\varepsilon_2]\!].\ \ldots\ \uparrow\mathcal{E}[\![\varepsilon_i]\!]\text{'}]\!]_s \\
&\quad \mathcal{I}[\![\text{LispContext at: } \sigma \text{ put: (LispContext evaluateSmalltalk:} \\
&\qquad \text{'[:}\alpha_1 \text{ :}\alpha_2 \ldots \text{ :}\alpha_i \mid \text{self } \sigma\text{: } \alpha_1 \text{ _: } \alpha_2 \ldots \text{ _: } \alpha_i\text{]')}]\!]_s
\end{aligned}$$

σ

Alternatively, we can compile the functionality into the block and arrange for the corresponding method to invoke the block when activated:

$\mathcal{E}[\![$ `(define (`$\sigma\ \alpha_1\ \alpha_2\ \ldots\ \alpha_i$`)` $\varepsilon_1\ \varepsilon_2\ \ldots\ \varepsilon_i$`)` $]\!] =$
$\mathcal{I}[\![$LispContext compile:
'σ: α_1 _: α_2 … _: α_i $\uparrow\sigma$ value: α_1 value: α_2 … value: α_i'$]\!]_s$
$\mathcal{I}[\![$LispContext at: σ put: (LispContext evaluateSmalltalk:
'[:α_1 :α_2 … :α_i | $\mathcal{E}[\![\varepsilon_1]\!]$. $\mathcal{E}[\![\varepsilon_2]\!]$. … $\mathcal{E}[\![\varepsilon_i]\!]$]')$]\!]_s$
σ

3.2.4 Unbound (`lambda`) Functions

More recent versions of Smalltalk implement blocks as proper closures. This makes the implementation of `lambda` functions almost trivial:

$\mathcal{E}[\![$ `(lambda (`$\alpha_1\ \ldots\ \alpha_i$`)` $\varepsilon_1\ \varepsilon_2\ \ldots\ \varepsilon_i$`)` $]\!] =$
[:α_1 … :α_i | $\mathcal{E}[\![\varepsilon_1]\!]$. $\mathcal{E}[\![\varepsilon_2]\!]$. … $\mathcal{E}[\![\varepsilon_i]\!]$]

3.2.5 Local Definitions

Lisp provides the `let` and `setq` forms for creating and mutating local state. `let` can be modelled easily using a block defining the required local variables which is activated immediately with the initial values of those variables:

$\mathcal{E}[\![$ `(let ((`$\alpha_1\ \nu_1$`) ... (`$\alpha_i\ \nu_i$`))` $\varepsilon_1\ \varepsilon_2\ \ldots\ \varepsilon_i$`)` $]\!] =$
([:α_1 … :α_i | $\mathcal{E}[\![\varepsilon_1]\!]$. $\mathcal{E}[\![\varepsilon_2]\!]$. … $\mathcal{E}[\![\varepsilon_i]\!]$] value: $\mathcal{E}[\![\nu_1]\!]$ … value: $\mathcal{E}[\![\nu_i]\!]$)

`setq` maps trivially onto assignment:

$\mathcal{E}[\![$ `(setq` $\sigma\ \varepsilon$`)` $]\!] = (\sigma := \mathcal{E}[\![\varepsilon]\!])$

3.2.6 Special Forms for Conditional Execution

Lisp's `if` form has a direct equivalent in Smalltalk's ifTrue:ifFalse::

$\mathcal{E}[\![$ `(if` $\varepsilon\ \nu$`)` $]\!] =$ ($\mathcal{E}[\![\varepsilon]\!]$ ifTrue: [$\mathcal{E}[\![\nu]\!]$])

$\mathcal{E}[\![$ `(if` $\varepsilon\ \nu_1\ \nu_2$`)` $]\!] =$ ($\mathcal{E}[\![\varepsilon]\!]$ ifTrue: [$\mathcal{E}[\![\nu_1]\!]$] ifFalse: [$\mathcal{E}[\![\nu_2]\!]$])

The more complicated `cond` form can be modelled using a nested series of ifTrue:ifFalse: messages:

$\mathcal{E}[\![$ `(cond (`$\varepsilon_1\ \nu_{11}\ \nu_{12}\ \ldots\ \nu_{1j}$`) ... (`$\varepsilon_i\ \nu_{i1}\ \nu_{i2}\ \ldots\ \nu_{ij}$`))` $]\!] =$
($\mathcal{E}[\![\varepsilon_1]\!]$ ifTrue: [$\mathcal{E}[\![\nu_{11}]\!]$. $\mathcal{E}[\![\nu_{12}]\!]$. … $\mathcal{E}[\![\nu_{1j}]\!]$] ifFalse: [
…
$\mathcal{E}[\![\varepsilon_i]\!]$ ifTrue: [$\mathcal{E}[\![\nu_{i1}]\!]$. $\mathcal{E}[\![\nu_{i2}]\!]$. … $\mathcal{E}[\![\nu_{ij}]\!]$] …])

3.2.7 Special Forms for Boolean Operations

In a similar manner to cond, nested sequences of and: and or: can be used to realise the behaviour of the Lisp equivalents:

$$\mathcal{E}[\![\,\texttt{(and}\ \varepsilon_1\ \ \varepsilon_2\ \ldots\ \varepsilon_i\texttt{)}\,]\!] = (\mathcal{E}[\![\varepsilon_1]\!]\ \mathsf{and{:}}\ [\mathcal{E}[\![\varepsilon_2]\!]\ \mathsf{and{:}}\ [\ \ldots\ \mathcal{E}[\![\varepsilon_i]\!]]\ \ldots])$$

$$\mathcal{E}[\![\,\texttt{(or}\ \varepsilon_1\ \ \varepsilon_2\ \ldots\ \varepsilon_i\texttt{)}\,]\!] = (\mathcal{E}[\![\varepsilon_1]\!]\ \mathsf{or{:}}\ [\mathcal{E}[\![\varepsilon_2]\!]\ \mathsf{or{:}}\ [\ \ldots\ \mathcal{E}[\![\varepsilon_i]\!]]\ \ldots])$$

3.2.8 Function Application

Function application can take several forms, depending on the choice of implementation strategy for procedures. If procedures are compiled to blocks then arbitrarily complex expressions (including a literal symbol) can be evaluated and the resulting block invoked with a value message:

$$\mathcal{E}[\![\,\texttt{(}\varepsilon\texttt{)}\,]\!] = (\mathcal{E}[\![\varepsilon]\!]\ \mathsf{value})$$

$$\mathcal{E}[\![\,\texttt{(}\varepsilon\ \ v_1\ \ldots\ v_i\texttt{)}\,]\!] = (\mathcal{E}[\![\varepsilon]\!]\ \mathsf{value{:}}\ \mathcal{E}[\![v_1]\!]\ \ldots\ \mathsf{value{:}}\ \mathcal{E}[\![v_i]\!])$$

This mechanism must also be used if anything other than a symbol appears in the head position of a list.

If procedures are compiled into methods then a special case can be made of a symbol appearing in the head position of a list, and a message send (to self) used to effect the procedure call:

$$\mathcal{E}[\![\,\texttt{(}\sigma\texttt{)}\,]\!] = (\mathsf{self}\ \sigma)$$

$$\mathcal{E}[\![\,\texttt{(}\sigma\ \ v_1\ \ v_2\ \ldots\ v_i\texttt{)}\,]\!] = (\mathsf{self}\ \sigma{:}\ \mathcal{E}[\![v_1]\!]\ \_{:}\ \mathcal{E}[\![v_2]\!]\ \ldots\ \_{:}\ \mathcal{E}[\![v_i]\!])$$

3.2.9 Sequences

Sequences of statements are usually supported in Lisp using a special form like begin. Since such sequences have a value in Lisp they cannot be transformed into a simple sequence of Smalltalk statements. Instead, they must be wrapped in a block and then sent a value message immediately:

$$\mathcal{E}[\![\,\texttt{(begin}\ \varepsilon_1\ \ldots\ \varepsilon_i\texttt{)}\,]\!] = ([\mathcal{E}[\![\varepsilon_1]\!].\ \ldots\ \mathcal{E}[\![\varepsilon_i]\!]]\ \mathsf{value})$$

3.2.10 Quoting

A quoted s-expression is rewritten as an equivalent Smalltalk literal. If the literal is a list then it must be transformed into a Smalltalk structure that agrees with the definition of the primitives provided to access elements of a list.

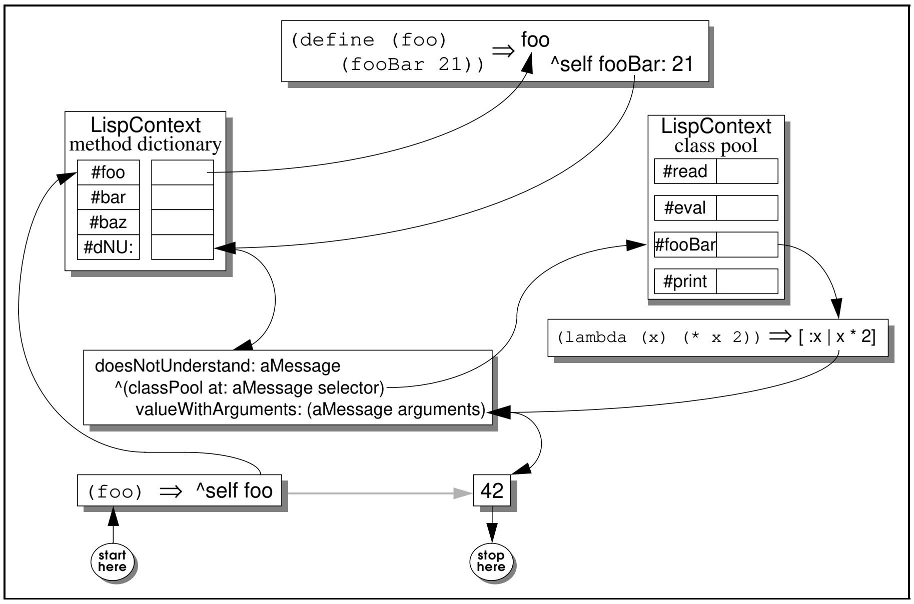
$$\mathcal{E}[\![\,\texttt{(quote}\ \varepsilon\texttt{)}\,]\!] = \mathcal{S}[\![\varepsilon]\!]$$

3.3 Lisp Procedures as Smalltalk Methods

3.4 Shortcomings

A major problem with this approach is that variable bindings can never shadow method definitions. If a method *x* is defined, a variable is bound to the corresponding block which performs the appropriate send. If this variable is then rebound to a `lambda` function, the effect will not be felt when applying the variable by name in the head position of a list, but *will* be felt if the `apply` procedure (or something similar) is used.

It is also possible to declare a variable, bind it to a `lambda` function, and then compile code that invokes it by name. In this case, a nonexistent method will be called, and the program will fail at runtime. This problem can be solved since the LispContext evaluating the program will receive doesNotUnderstand: aMessage. The errant selector can be extracted from aMessage, and the value of any variable of the same name extracted from LispContext's class pool. Sending a valueWithArguments: to this value should invoke the manually bound procedure, which is what the programmer was expecting.[12] Figure 3.4 shows how this works diagrammatically.



3.5 Compiling Lisp into Smalltalk Blocks

3.6 Shortcomings

This strategy does not suffer from the same broken semantics which were inherent when compiling procedures as methods. However, the price paid for the cleaner semantics is a significant extra overhead incurred by activating a block on every procedure application.

3.7 Some Examples

A couple of short examples should clarify how the rewriting process works. A Lisp function that calculates factorials can be written:

[12]This situation arises so infrequently in practice that the additional overhead of intercepting the doesNotUnderstand and redirecting execution to a block makes a negligible impact on performance.

```
(define (factorial n)
    (if (equal 0 n)
        1
        (times n (factorial (minus n 1)))))
```

Applying the rules for `define` and `if` results in:

```
factorial: n
    ↑((equal 0 n)
        ifTrue: [1]
        ifFalse: [(times n (factorial (minus n 1)))])
```

The rules for function application (for =, *, `factorial` and -) and literals (for 0, n and 1 variously) are applied to finish the job, resulting in:

```
factorial: n
    ↑((self equal: 0 _: n)
        ifTrue: [1]
        ifFalse: [(self times: n _: (self factorial: (self minus: n _: 1)))])
```

which is quite obviously (with suitable definitions of the primitives `equal`, `times` and `minus`) a Smalltalk definition of the same function.

A slightly more interesting example shows how blocks are used to provide local state, sequences, and lambda functions.

```
((lambda (x y)
        (let ((a (times x x)) (b (times y y)))
            (sqrt (plus a b))))
    3 4)
```

Rewriting first the outer `lambda` and then the inner `let` yields the following transformations:

```
( ([ :x :y |
        (let ((a (times x x)) (b (times y y)))
            (sqrt (plus a b))).])
    3 4)

( ([ :x :y |
    ([ :a :b | (sqrt (plus a b)).])
        value: (times x x)
        value: (times y y)).])
    3 4)
```

Applying the trivial transformations for the procedure applications and literal values gives the final Smalltalk equivalent:

```
(([ :x :y |
    ([ :a :b | (self sqrt: (self plus: a _: b )).]
        value: (self times: x _: x)
        value: (self times: y _: y)).])
    value: 3 value: 4)
```

4 Primitives

The example of section 3.7 demonstrates the need to provide some connectivity between procedures written in Lisp and 'primitives' defined in Smalltalk. These are provided by a special form which looks almost like that for `define`, but whose second argument is a string containing the body of the method in Smalltalk that implements the primitive:

$$\begin{array}{l}\mathcal{E}[\![\texttt{(primitive (}\sigma\ \alpha_1\ \alpha_2\ \ldots\ \alpha_i\texttt{)}\ \varepsilon\texttt{)}]\!] = \\ \quad \mathcal{I}[\![\text{LispContext compile: '}\sigma\text{: } \alpha_1 \text{ \_: } \alpha_2 \ldots \text{ \_: } \alpha_i\ \varepsilon\text{'}]\!]_s \\ \quad \mathcal{I}[\![\text{LispContext at: } \sigma \text{ put: (LispContext evaluateSmalltalk:} \\ \qquad \text{'[:}\alpha_1 \text{ :}\alpha_2 \ldots \text{ :}\alpha_i \text{ | self } \sigma\text{: } \alpha_1 \text{ \_: } \alpha_2 \ldots \text{ \_: } \alpha_i\text{]')}]\!]_s \\ \quad \sigma\end{array}$$

Using this facility, the primitive for `minus` in the examples could be written in a similar manner to a normal procedure definition:

```
(primitive (minus x y) "↑x - y")
```

resulting in a method being installed in LispContext

```
LispContext>>minus: x _: y
    ↑x - y
```

and the variable minus being bound to the block

```
[ :x :y | self minus: x _: y]
```

5 Performance

6 Metacircular Abstraction

Some dialects of Lisp provide extensible syntactic features, allowing the user to provide rewrite rules for additional linguistic constructs within Lisp itself. Given such a feature there is no reason why special forms for constructs such as `let` should be provided directly, since these can be expressed in terms of `lambda`. The rewrite rules given above for `let`, `begin` and so on do not need to be provided directly by the translator since identical results can be achieved by transforming them first into `lambda` expressions, and then using the 'built-in' transformation to convert the result from Lisp into Smalltalk.

The advantages for the linguistic experimentor of leaving as many special forms as possible to be defined 'soft' from within the embedded language itself are obvious. The only major additional feature that our evaluator requires is some way of constructing lists with a mixture of literal and evaluated data. Some dialects of Scheme provide the special forms `quasiquote` (abrreviated to ` in the same way as `quote` is abbreviated to `'`), `unquote` (abbreviated to `,`) and `unquote-splicing` (or `,@`) which do just this [Dyb??], and adding these to our translator is not too difficult.

We could define the `let` construct described in section 3.2.5 as follows, using a special form that works somewhat like `define`:

```
(define-syntax (let bindings . body)
   ((lambda
      ,(map car bindings)
      ,@body)
         ,@(map cadr bindings)))
```

Arranging for these rewrite 'definitions' to be applied recursively to the structure returned from our parser, *before* any translation into Smalltalk is performed, provides a very powerful mechanism for extending the abstraction of our embedded language from within the embedded language itself.[13]

7 Where to Buy One

The Lisp system described in this paper, metacircularities and all, is available via anonymous FTP from the Manchester Smalltalk archive. Connect to the host `mushroom.cs.man.ac.uk` (130.88.13.70) using the username "`anonymous`" and your full e-mail address as password. The Lisp system is in the directory "`/pub/goodies/manchester/4.1/lisp`", along with example Lisp dialects (Scheme and Common Lisp) and some example programs. For users in Canada and the United States, there is a mirror of this archive at UIUC on the host `st.cs.uiuc.edu` (128.174.241.10).

The author welcomes comments, and donations of Lisp dialects and programs written using this system, via e-mail at the address shown on the first page.

References

[ASS85] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge MA, 1985, ISBN 0-262-51036-7.

[Dyb??] ? Dybvig, *The Scheme Programming Language*, ???.

[Gra??] Justin Graver, *T-Gen User's Guide*, available via anonymous FTP with the T-Gen source, see section 7.

[Hop??] Trevor Hopkins, *SMACK — The Smalltalk Actor Kernel*, wherever.

[StS??] Guy Lewis Steele, Gerald Jay Sussman, et al, *The Revised^4 Report on the Algorithmic Programming Language Scheme*, ????.

[Wol88] Mario Wolczko, *Semantics of Object-Oriented Languages*, Ph.D. Thesis, Department of Computer Science, University of Manchester, UK, June 1988. Available as technical report UMCS-CSD-88-????.[14]

[13] This gives rise to a somewhat incestuous relationship. The embedded language is implemented in the base language, yet in order to translate from the former to the latter the base language has to 'call out' to procedures defined in the embedded language.

[14] These aren't cheap: if you're interested you'd be better advised to pick up the PostScript via anonymous FTP from `mushroom.cs.man.ac.uk` in the file "`/pub/mushroom/theses/wolczko.ps.Z`" and send it to your favourite laser printer.