Embedded Lisps in Smalltalk-80

This package allows the adventurous programmer to experiment with dialects of Lisp. The idea is that a minimal translator and runtime support for Lisp are provided in Smalltalk, which in themselves do not provide much of a useful language but are sufficient to bootstrap an implementation of Lisp written in itself.

Only a handful of special forms are provided. These "intrinsic forms" are so central to Lisp that it would be unthinkable to provide a "Lisp" that did not have them, or their equivalents.

One of the intrinsic forms provides a mechanism by which primitives can be defined (as Smalltalk embedded in Lisp "procedure" definitions). Usually one of the first primitives to be defined is one that allows user-defined forms to be added to the translator's syntax table — using this primitive the special forms of any Lisp dialect can be defined as rewrites in Lisp.

User-defined forms have precedence over the intrinsic forms during translation, and so even those can be redefined to have different semantics if necessary.

> A NOTE FOR THE IMPATIENT EXPLORER
> You should at least read the sections "Filing It In" and "Running The Examples" before going off to play with it.

Contents

1 Brief Release Notes

There are bound to be many bugs and deficiencies, and probably even some methods, artifacts of earlier implementation strategies, which have not been spotted and removed. The testing has been limited to making the two extended examples work; this has the unfortunate corollary, true of any translation system, that it works just fine for *correct* programs, but probably fails miserably for some *incorrect* programs. Most of the effort in developing a translator for any language is devoted to error detection and recovery — the severely limited amount of time I had available in which to do this work has had a detectable effect on the quality of the latter. On the other hand since the intrinsic features are so few they pervade the implementation of even the tiniest dialect of Lisp and so should be almost correct by now.

The most serious deficiency is the lack of rigor in the checking of arguments in the intrinsic forms — although the obvious mistakes will be spotted, some obscure things may well just generate an error notifier from the guts of the translator (the most common will likely be nil being sent a message understood only by lists).

2 Packing List

`.`	this directory: run your image from here!
`README`	this document
`README.dvi`	this document as a TEX3 DVI file
`README.ps`	this document in PostScript
`doc/`	miscellanous documents
`doc/metalinguistic.ps`	draft paper on metalinguistic abstraction
`lisp/`	example Lisp dialects
`lisp/scheme.lsp`	essential+some Scheme as defined by [1]
`lisp/scheme-chez.lsp`	a couple of the Chez-Scheme extensions
`lisp/interface.lsp`	confirmers, prompters, file browsers, &c…
`lisp/common-lisp.lsp`	a subset of Common Lisp
`lisp/tak.lsp`	support for the performance analysis
`examples/`	example programs
`examples/continuations.lsp`	see [2], page 81
`examples/word-count.lsp`	see [2], section 9.2
`examples/text-formatter.lsp`	see [3], section 9.7
`examples/countable.text`	test input for `word-count.lsp`
`examples/example.text`	test input for `text-formatter.lsp`
`source/`	source directory
`source/FILE_ME_IN.st`	expression to file-in Lisp stuff
`source/Lisp-continuations.st`	stand-alone support for continuations
`source/Lisp-misc.st`	changes to system classes
`source/Lisp-Parser.st`	the Lisp parser and scanner
`source/Lisp-Runtime.st`	the runtime system
`source/Lisp-Tools.st`	Lisp launcher, browsers, file lists, &c…
`source/Lisp-Translators.st`	the two Lisp → Smalltalk translators

3 Filing It In

While filing the various bits and pieces in, you should make sure that you are running your image out of the directory *immediately above* the directory `source`. You can file it into any 4.1 or similar image; it should even work with VisualWorks release 1.

It is vitally important that you file the various bits into your image in the correct order. The file called `FILE_ME_IN.st` contains an expression which will do this for you.

Changes to system classes are restricted to a few additions and two insignificant changes. The additions are:

Object	isList
OrderedCollection	isList
Array	isList
SystemDictionary	takeuchi:with:with:
Integer	takeuchiWith:and:

The latter two are used to compare the performance of Lisp and Smalltalk. A very minor change is made to the following method, which makes underscore ($_) behave as an "extended alphabetic" character:

Character class	initClassificationTable

Finally, the classes Character and Scanner are re-initialized to effect the change in status of the underscore character.

If you choose to file-in the continuations support, `Lisp-continuations.st`, then the following additions will be made to system classes:

BlockClosure	copyStack
Context	copyStack

One additional class, Continuation, will also be added to the category Kernel-Methods. The implementation of continuations is not dependent on any of the other Lisp stuff, and `Lisp-continuations.st` can be used independently. There is an example in class Continuation demonstrating how continuations can be used from within Smalltalk programs.

4 The User Interface

After filing in the sources, `FILE_ME_IN.st` opens a Launcher with an additional menu item called **Lisp Utilities** (any customization of the other menu items will not be affected). This menu supports the following utilities:

- **lisp file list**
  Like a Smalltalk file list, but with the yellow button menu customized for Lisp. The list of files in the upper half of the file list supports the following additional operations:
  - **read**
    reads the selected file as a Lisp program, and opens an inspector on the corresponding structured representation;

- **translate**
  reads the selected file, translates it into Smalltalk, and opens a CodeView on the result;
- **eval**
  reads, translates and then evaluates the program in the selected file (this is the one you'll use most often);
- **inspect**
  just like **eval**, but opens an inspector on the result of the last expression evaluated from the selected file.

The lower code view is customized in a similar manner. The operate menu includes:

- **read**
  read the current selection and open an inspector on the structured representation of the expression;
- **translate**
  open a CodeView on the Smalltalk equivalent of the selected expression;
- **eval**
  evaluate the current selection (the lisp equivalent of **do it**);
- **print**
  evaluate the current selection and paste the result (the lisp equivalent of **print it**);
- **inspect**
  evaluate the current selection and inspect the result.

- **lisp file editor on...**
  The same as the Smalltalk version, but with the code view modified to support the same operations as the **lisp file list**.

- **lisp workspace**
  The same as the Smalltalk equivalent, but with the code view modified to support the same operations as the **lisp file list**.

- **advanced...**
  A collection of some useful things that I left lying around permanently in a workspace while developing all this stuff, and which eventually became so useful and so permanent that they warranted a menu of their own:

  - **flush variables**
    removes all the variable bindings in Lisp's top-level context;
  - **flush functions**
    does the same for functions;
  - **flush symbol table**
    empties Lisp's cached map of Lisp symbols → Smalltalk method selectors;
  - **flush syntax table**
    empties Lisp's table of user-defined syntactic forms (the intrinsic forms are not affected);

- **generate methods**
  causes procedures to be translated into methods (flushes the entire Lisp context);
- **generate blocks**
  causes procedures to be translated into blocks (flushes the entire Lisp context);
- **browse defined functions**
  opens a method list on the Smalltalk methods corresponding to the currently defined Lisp procedures;
- **browse defined variables**
  opens an inspector on the Dictionary containing the variables currently bound in Lisp's top-level context;
- **performance analysis**
  runs `takeuchi(18,12,6)` in both Smalltalk and Lisp, and opens a window containing a report of the relative performance;
- **file out lisp system**
  files out the entire Lisp system into a subdirectory called `source`, and generates the file `FILE_ME_IN.st` (flushes the entire Lisp context before doing so).

5 Running The Examples

Two example dialects (Scheme and a subset of Common Lisp) are provided, as well as three example programs that use them: a trivial demonstration of continuations, plus two not-so-trivial examples taken from books on Scheme and Common Lisp which were used as "proofs of concept".

The following instructions seem utterly trivial, but then I sometimes have difficulty identifying what others may find esoteric in what I take for granted.

For the examples to work properly, you should be running your image out of the directory immediately above the directories `examples` and `lisp`. If not, it will all go horribly wrong.

5.1 Scheme

How to run the Scheme examples, and what should happen when you run them…

5.1.1 `continuations.lsp`

Fist the trivial example: `continuations.lsp`. (You must have filed in the optional continuation support along with the rest of the Lisp stuff. If you haven't, then it won't hurt to file-in `source/Lisp-continuations.st`, on its own, right now.)

Open a **lisp file list** on `examples/continuations.st`. This example is the odd one out: you *won't* see anything interesting if you just **eval** it in one go from the file list. Instead, select from the top of the file down to the end of the definition of `factorial`. **eval** this from within the code view. The Transcript will make some noise while `scheme.lsp` does its stuff.

Once you get the normal cursor back, select and **inspect** each of the last three lines (in the order that they appear) one at a time. The first expression calculates `(factorial 4)`, saving a continuation at the point where the base case (n < 1) is calculated. This continuation can be restarted later with a different base case.

The second and third expressions invoke that continuation, restarting the factorial calculation performed in the first expression with a different base case (specified by the argument to the continuation).

5.1.2 `word-count.lsp`

Find the file `examples/word-count.lsp` using the **lisp file list**. For this example, it *is* safe to **eval** it all in one go from the upper "list view" of the file list.

Again, the Transcript will rattle a bit while the example loads the essential Scheme, the Chez-Scheme printing and structure extensions, and the support for the dialogs that will pop up later. If all goes well, a confirmer should appear asking you if you want to run the example. Answer "yes" and use the default input file name `examples/countable.text`. You can choose a different output filename if you like. Once the frequency counting program has done its stuff, you'll be asked if you want to inspect the result. Answer "yes" again and a TextView on the resulting table should pop up.

5.2 Common Lisp

Unlike the Scheme examples, this one *does* require at least one word of warning.

5.2.1 `text-formatter.lsp`

As before, find `examples/text-formatter.lsp` in a **lisp file list**. **eval** it from the upper list view. The Transcript will go crazy complaining about redeclared block args and undeclared variables: *this is normal*. However, the mess is slightly too great for Lisp to sort out in one pass, so unfortunately you'll have to answer "no" when it invites you to run the example. **eval** the whole thing a *second* time which will ensure certain things are untangled sufficiently, then answer "yes" to the invitiation and proceed as with the previous example.

The final TextView that pops open uses a fixed-width font to show that the formatted text really has been filled properly.

5.3 Performance Analysis

This is probably the easiest of the lot: just select **performance analysis** from the **advanced...** section of the **Lisp Utilities** menu in the Launcher. For this to work, the file `lisp/tak.lsp` *must* be present (it contains the Lisp definition of Takeuchi's function which will be used to meter the performance, and should be present by default). You'll get a confirmer asking if it's okay to trash the Lisp environment: just say "yes".

After a short time (somewhere between 10 and 100 seconds, depending on how expensive your workstation was) a report window will pop up. This shows the ratio of

(Lisp performance) / (Smalltalk performance)

for `takeuchi(18,12,6)` for all four combinations of:

- lisp compiled into methods
- lisp compiled into blocks

×

- statically-bound Smalltalk (the receiver is essentially a constant for every message send)
- dynamically-bound Smalltalk (the receiver is the first argument in the message send).

Interestingly, the results vary considerably between platforms. For the ratio of Lisp compiled to methods against dynamically-bound Smalltalk I get:

platform	*ratio*
Sun ELC (Sparc)	1.53
HP9000/750 (PA-RISC)	3.55

(You *aren't* misunderstanding this: the Lisp really *is* almost four times faster than the equivalent Smalltalk on an HP workstation.)

I'd be very grateful to receive copies of the report window for results obtained on other platforms.

5.4 Important Note

Don't forget: all this stuff was done for fun. I wasn't trying to build a production Lisp implementation and so the Lisp dialects are not as rigorous as they might be (the Common Lisp in particular is significantly wide of the mark in a number of places, and the example program was modified slightly to compensate; the Scheme is not too bad, to the extent that the example Scheme programs are quoted absolutely unmodified from [2][1]).

6 Intrinsic Forms

A handful of syntactic forms is provided. These form a "syntactically complete" set, and from them any other syntactic form can be synthesized by defining it in Lisp. These intrinsic forms are as follows:[2]

`(primitive (`⟨proc-name⟩ ⟨args...⟩`) "`⟨method-definition⟩`")`

acts a bit like `(define...)` except that the body of the procedure is bound to the Smalltalk ⟨method-definition⟩ given as a string in the second argument. ⟨args...⟩ are bound to the actual arguments within the Smalltalk ⟨method-definition⟩ (the ⟨method-definition⟩ should *not* contain the method pattern: this is provided automatically by the translator). Returns the closure representing the primitive.

`(define` ⟨var-name⟩ `[`⟨expression⟩`])`

appearing *externally* this evaluates ⟨expression⟩ in the top-level context, and binds the result to the top-level variable named ⟨var-name⟩. The result is the symbol ⟨var-name⟩.

Appearing *internally* to any expression sequence, this creates a variable local to the current environment called ⟨var-name⟩, evaluates ⟨expression⟩ in the context of that environment, and binds the result to ⟨var-name⟩. Internal definitions are evaluated in-order,

[1] Well, almost. In the `word-count` program I added a single gratuitous `unwind-protect` to close any open streams in the event of an error or user interrupt. I consider its omission in the original to be a bug.

[2] The stuff in `courier` is literal Lisp; things in ⟨angles⟩ are syntactic variables; things in [brackets] are optional; things followed by an ellipsis are repeated an arbitrary number of times…

before any real statements in the sequence are executed. They are therefore evaluated in the context of any preceding, but not in the context of any succeeding, definitions. The result of an internal definition (which will rarely be useful!) is the newly bound value.

If ⟨expression⟩ is missing, the ⟨var-name⟩ is bound to `nil`.

`(define (`⟨proc-name⟩ ⟨args...⟩`)` ⟨body...⟩`)`
: must appear *externally* and binds ⟨proc-name⟩ to a procedure with arguments ⟨args...⟩. The procedure expects an exact match between formal and actual argument counts, and its overall result on application will be the value of the last expression in ⟨body...⟩.

  This is syntactic sugar for "`(define` ⟨proc-name⟩ `(lambda (`⟨args...⟩`)` ⟨body...⟩`))`", and returns the newly bound closure as its result.

`(setq` ⟨var-name⟩ ⟨expression⟩`)`
: rebinds ⟨var-name⟩ to the result of evaluating ⟨expression⟩. It is (obviously) an error to try to rebind an unbound variable. Returns the newly bound value.

`(`⟨expression⟩ ⟨args...⟩`)` (function application)
: applies ⟨expression⟩ to ⟨args⟩. See the explanation of lists in section 8 for details.

`(lambda` ⟨arglist⟩ ⟨body...⟩`)`
: returns a closure that can be applied as a function whose action is to perform the expression sequence ⟨body...⟩ in the context that was in effect at the time of definition, extended by binding the variables named in ⟨arglist⟩ to the values supplied at each time of use. If ⟨arglist⟩ is `nil`, no arguments will be expected.

`(and` ⟨expression...⟩`)`, `(or` ⟨expression...⟩`)`
: the usual boolean combinations. `and` evaluates its arguments left-to-right returning `false` immediately if any argument evaluates to `false`, or returning `true` if they all evaluate to `true`. `or` does the same thing, with `true` and `false` reversed.

  Note that the set-theoretic interpretations apply, where `(and)` is `true` and `(or)` is `false`.[3]

`(if` ⟨expression⟩ ⟨consequent⟩ [⟨alternate⟩]`)`
: if ⟨expression⟩ evaluates to `true` then the result of evaluating ⟨consequent⟩ is returned, otherwise ⟨alternate⟩ is evaluated and its value returned. (If ⟨alternate⟩ is missing, the result is `nil`.)

`(quote` ⟨expression⟩`)`
: protects ⟨expression⟩ from evaluation. The result is ⟨expression⟩ itself, unevaluated.

All other syntactic forms (`let`, `cond`, and their friends and relatives) can be defined easily in terms of the above. To support the definition of syntactic rewrite rules for user-defined forms, the following are also provided:

`(quasiquote` ⟨expression⟩`)`
: protects ⟨expression⟩ from evaluation in the same way that `(quote` ⟨expression⟩`)` does, except that subcomponents of ⟨expression⟩ can be "unprotected" selectively using one of the following...

[3]Try thinking of `and` and `or` as the universal and existential quantifiers (respectively) to understand why this should be so. (Understanding the isomorphism between quantifiers and logical operations is left as an exercise for the interested reader! `;-)`

`(unquote` ⟨expression⟩`)`
: when appearing inside a quasiquoted expression this is replaced by the result of evaluating ⟨expression⟩.

`(unquotesplicing` ⟨expression⟩`)`
: when appearing inside a quasiquoted *list*, ⟨expression⟩ is evaluated (and must result in a list); the entire `unquotesplicing` expression (a single element in the enclosing list) is replaced by the arbitrary number of elements of the list-valued result of ⟨expression⟩,[4] thus:

If you're as confused as I was when I first saw quasiquoting, here's a quick example:

```
(quasiquote
   (a b (unquote (* 14 3))
    c d (unquotesplicing (map car (quote ((1 2) (3 4)))))
    ))
⇒ (a b 42 c d 1 3)
```

The usual shorthand icing anointments atop the quoting cake are provided by the parser:

cake	*icing*
`(quote` ⟨expression⟩`)`	`'`⟨expression⟩
`(quasiquote` ⟨expression⟩`)`	`` ` ``⟨expression⟩
`(unquote` ⟨expression⟩`)`	`,`⟨expression⟩
`(unquotesplicing` ⟨expression⟩`)`	`,@`⟨expression⟩

Using this notation, the above example becomes:

```
`(a b ,(* 14 3) c d ,@(map car '((1 2) (3 4))))
⇒ (a b 42 c d 1 3)
```

6.1 Example Of User-Defined Syntactic Forms

A very quick example of how additional special forms are defined using the intrinsic forms (especially the quasiquoting forms) as rewrites within Lisp itself:[5]

```
;;; primitive definitions for (car ⟨expr⟩) etc. are assumed...

;; define a primitive for extending the syntax table

(primitive (add-syntax _name _rewrite)
   "↑LispTranslator addSyntax: _name calling: _rewrite")

;; define (let ((var_i expr_i)...) body...) in terms of the intrinsics
```

[4]This isn't a particularly lucid explanation: for an expert description of `unquotesplicing`, see [1], section 4.2.6.

[5]For reasons of pure convenience, a very weak version of the `let` form is provided intrinsically. Both the Scheme and Common Lisp dialects are quick to redefine it more powerfully, though. (In the case of Scheme, it is extended significantly to cope with both "named" and "unnamed" let.)

```
(add-syntax 'let (lambda (x)
   (define bindings (car x))
   (define body (cdr x))
   `((lambda ,(map car bindings) ,@body)
      ,@(map cadr bindings))))
```

In the above example, `add-sntax` is defined to take two arguments: a symbol (the name of a syntactic form) and a closure (called to rewrite the arguments of this syntactic form). During parsing, the translator's syntax table is consulted whenever a symbol is encountered as the first element of a list *L*. If this symbol is present in the syntax table the tail of the list is passed to the corresponding closure as a list. The result returned by this closure replaces *L* in the structured representation of source program. The example rule for `let` shown above would cause the following rewrite during translation:

```
(let ((a 14) (b 3)) (* a b))
⇒ ((lambda (a b) (* a b)) 14 3)
```

7 Intrinsic Primitives

There are no predefined primitives at all. Any primitives that are required are defined by the user in Smalltalk. (This includes the primitive that connects Lisp to the syntax-extension mechanism.)

The syntactic forms of any Lisp govern the flow of control and data through a program, but not the computation performed on the data manipulated by that program. All data manipulation is controlled by the primitives, so changes in the way *anything* is represented merely requires that the relevant primitives be redefined to use the new representation. The example dialects of Scheme and Common Lisp are replete with examples of primitive definitions.

8 Intrinsic Data Types

Atoms
`true`, `false`, and `nil` are provided intrinsically. Lisp and Smalltalk use identically the same objects to represent these values.

Numbers
The usual Smalltalk mechanisms for arithmetic are preserved, however the parser currently only recognises integers. The additions required to recognise floats are not too difficult. If you file-in any additional number classes (such as the implementation of `Complex` from the goodies) these will be assimilated into Lisp's arithmetic. All numbers are self-evaluating.

Characters
Supported with the same $-prefix syntax as Smalltalk. All characters are self-evaluating.

Symbols
All identifiers are translated into Smalltalk Symbols by the parser. A quoted identifier will yield the corresponding symbol when evaluated. Symbols evaluate to their bound value in the current environment.

Strings
Strings are delimited by "double quotes" rather than single quotes. A closing " followed immediately by an opening " will be converted into a single " character by the scanner. All strings are self-evaluating.

Lists
Any sequence, delimited by parentheses, of types described in this list. If compiling procedures into closures (or in the cases where the first element is not a symbol), Lists are evaluated as follows:

- the first element is evaluated (it must evaluate to a closure) to yield a procedure;
- the rest of the elements are evaluated left-to-right to form a list of arguments;
- the procedure is applied to the arguments to give the final result of evaluating the list.

If compiling procedures into methods, lists are evaluated as follows:

- the second and subsequent elements are evaluated left-to-right to form a list of arguments;
- a message send, using the arguments just evaluated, is performed to invoke the relevant CompiledMethod in LispContext to yield the result of evaluating the list.

Argument evaluation is therefore so-called "applicative order" (as opposed to "normal order").

Lists are represented internally in one of two forms, depending on how they are being used. Lists that are unlikely to change in length are represented as Arrays, and lists that are undergoing construction work are represented as OrderedCollections.[6]

9 Suggestions For The Really Adventurous

- Try extending the translator to cope with improper lists, and then implement "optional" and "rest" arguments. The mechanisms are not impossible to design, but are an interesting implementation challenge if "well-behaved" procedures are still to run reasonably efficiently.
- Try re-implementing the parser so that it is called via an intrinsic primitive defined in the top-level context. Bootstrap a dialect of Lisp and write a `read` procedure in it, then rebind the reader so that the translator uses the one defined in Lisp. The syntactic extension mechanism will have to be implemented in this reader too.

[6] A trivial subclass of OrderedCollection called CompactList has a useful **instance creation** method called readFrom: aCollection (which converts aCollection, and recursively all of its subcollections, into CompactLists), and overrides printOn: to provide an indented representation of itself. The combination of these two methods is particularly useful during debugging.

- Do the same with the translator. If you get this far then you'll have, apart from the necessary invisible runtime support, a Lisp that's implemented completely in itself: open a beer, sit back, and breathe a deep sigh of aesthetic appreciation.
- Implement decent debugging facilities. Subclass the Smalltalk Debugger and modify it to differentiate between Smalltalk "primitives" and Lisp procedures. For lisp procedures, show the Lisp rather than the Smalltalk. Maybe even add a third file to the SourceFileManager holding Lisp definitions.
- Implement a proper macro facility. This should be possible entirely within Lisp by sugaring the existing syntax-extension mechanism. Real wizards might like to try making macros hygienic and referentially transparent, maybe following the guidelines set out in the appendix on macros in [1]; this will require changes to the parser, translators, and possibly the runtime support too.

10 The Future (And Fractional Divisions Thereof)

This was my second attempt at embedding an extensible Lisp in Smalltalk, and the lessons have been many and varied. "Round 3" has just commenced, and will address squarely many of the serious deficiencies remaining in this second attempt. The most important improvement will be support for improper lists and the provision of "rest" arguments in procedures. Initial experiments have already shown that:

```
(let loop ((a 10000))
   (if (= a 0)
      #f
      (begin
         (map * (list 1 2 3) (list 4 5 6))
         (loop (- a 1)))))
```

(where `map`, `*` and `list` are defined to take an arbitrary number of arguments) achieves about 70% the performance (in time) of MIT Scheme release 7.1.3, when compiled into Smalltalk on the same platform.

11 Address For Correspondance

The author welcomes comments, suggestions, criticisms, bug reports, bug fixes, and donations of code written using this system. These should preferably be communicated via email to:

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

For those without email access, send Royal Mail to:

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

In extreme circumstances (maybe you ran out of postage stamps, or have a lot of money to give away to academics with a predilection for daydreaming about language implementations in their spare time) ’phone:

(061) 275 6145	(UK)
+44-61-275 6145	(anywhere else)

12 References

[1] William Clinger and Jonathan Rees (*eds.*), “The Revised^4 Report on the Algorithmic Language Scheme”, MIT AI Lab, November 1991.[7]

[2] R. Kent Dybvig, “The SCHEME Programming Language”, Prentice-Hall, 1987. ISBN 0-13-791864-X.

[3] Deborah G. Tatar, “A Programmer’s Guide to COMMON LISP”, Digital Press, 1987. ISBN 0-932376-87-8.

[7] DVI and PostScript versions of this document are available by anonymous FTP from `export.lcs.mit.edu` as part of the MIT Scheme distribution.