

Compiling Emacs Lisp to Scheme

Eric Knauel

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13, 72076 Tübingen, Germany
`knauel@informatik.uni-tuebingen.de`

Abstract We present a metacircular compiler written in Scheme that compiles Emacs Lisp to Scheme code. The compiler either emits source text or a Scheme function immediately useable at the prompt of the Scheme implementation running the compiler. This characteristic makes the compiler especially useful in an interactive setting. Thus, our compiler provides a basis for running unchanged Emacs Lisp code in a Emacs editor using Scheme as its extension language.

In this paper we show how to compile Emacs Lisp particularities such as functions with complex signatures and symbols to efficient Scheme code.

1 Introduction

The GNU Emacs and XEmacs editors are among the most popular text editors. One reason for their popularity is the possibility to extend and to change their features and behavior using a built-in programming language: Emacs Lisp [19, 12]. Since the introduction of Emacs Lisp users have written more than one million lines of Emacs Lisp code. Originally intended for writing new editor modes, Emacs users started writing complex applications like web browsers, and mail and NetNews-clients in Emacs Lisp — although Emacs Lisp lacks almost all concepts that are desirable for programming large-scale applications. Moreover, the implementations of Emacs Lisp are slow compared to implementations of other Lisp dialects.

To overcome these problems, the GNU Emacs developers plan to replace Emacs Lisp [17] and its interpreter by Guile, an implementation of Scheme [8] specifically suited for integration as an *extension language*. Independent of these efforts of the GNU Emacs project, Sperber and Neubauer showed that an automatic migration of Emacs Lisp to maintainable Scheme code is feasible [15]. Even in this setting, it is essential that a Scheme-driven Emacs is able to run unchanged Emacs Lisp code. Running the current Emacs Lisp implementation in parallel with a Scheme implementation is a far too complex solution. We present a compiler written in Scheme that compiles Emacs Lisp to Scheme code. Thus, our compiler provides a basis for running unchanged Emacs Lisp code in a Scheme-driven Emacs.

The compiler may generate the target code in two different formats. First, it is possible to emit the Scheme source as text and, secondly, the target code may

be returned in form of a Scheme function, which may be applied directly in the Scheme implementation running the compiler. The latter feature is especially useful in an interactive system like the Emacs editors where the user may add or redefine functions and variables anytime using the *Lisp interaction mode*.

We choose to implement the compiler using Scheme 48 [11], a well-structured and well-maintained Scheme implementation especially suited for interactive use. Scheme 48 endues a byte-code compiler, which compiles the function issued by our compiler to an efficient byte-code representation immediately. The current development version also offers a byte-code optimizer and native-code compiler. Thus, the function issued by the Emacs Lisp compiler may be compiled to an efficient native code version, immediately useable at the Scheme 48 prompt.

Overview Section 2 describes the model of compilation and discusses the compilation of some Emacs Lisp particularities. Section 3 gives an overview of the run-time system needed for running compiled Emacs Lisp code. Section 4 discusses primitive functions: functions written in C accessible from Emacs Lisp. We give an overview on related work in Section 5 and end with a discussion on future work.

2 The compiler

In this section we describe the compilation method used by our compiler and discuss the compilation of some Emacs Lisp particularities. The compiler starts out as an interpreter and evolves into a compiler suitable for the interactive use within Emacs, but also as a source code generating compiler.

2.1 Metacircular compilation

A *metacircular interpreter* is an interpreter that expresses constructs found in the source language in terms of the language the interpreter is written in. Figure 1 shows a metacircular interpreter for a minimal subset of a dynamically-scoped Emacs Lisp like language. The interpreter is written in Scheme. The source language consists only of numbers, variables, functions with one argument and function calls. The `eval` function evaluates an expression of this language given in form of a quoted S-expression in a metacircular fashion. Note that Scheme is a lexically-scoped language, which means we can not fall back on Scheme's variable lookup mechanism for implementing this aspect of the source language. Instead, we pass an environment `env` that maps variables to their values.

`Eval` evaluates all constructs of the source language to *implementation functions* with an uniform signature. An implementation function gets applied to an environment and yields the value of a construct. The metacircular nature of this interpreter may be observed by looking at the implementation of functions, function calls and numbers, these constructs fall back on the accordant constructs in the interpreter language.

```

(define (lambda? e)
  (and (list? e) (eq? (car e) 'lambda)))
(define lambda-arg caddr)
(define lambda-body caddr)
(define (funcall? e)
  (and (list? e) (not (lambda? e))))
(define funcall-op car)
(define funcall-arg cadr)

(define (eval e)
  (cond
    ((number? e)
     (lambda (env) e))
    ((symbol? e)
     (lambda (env)
       (cdr (assoc e env))))
    ((lambda? e)
     (lambda (env)
       (lambda (a)
         ((eval (lambda-body e)) (cons (cons (lambda-arg e) a) env))))))
    ((funcall? e)
     (lambda (env)
       ((eval (funcall-op e)) env) ((eval (funcall-arg e)) env))))))

```

Figure1. A metacircular interpreter

Feeley shows how to turn an interpreter like this into a compiler [5]. The idea is to separate the syntactical analysis from the evaluation of a construct. We use this approach to turn our interpreter into a compiler. A closer examination of the interpreter shown in Figure 1 yields the observation that the implementation functions contain calls to functions that belong to the domain of syntactical analysis. For example, each time the implementation function for a certain `lambda` expression is applied, `lambda-body` and `lambda-arg` are called to extract body and argument from the source code representation `e`—a syntactical analysis of the same part of source code repeats over and over again. This characteristic distinguishes interpreter from a compiler; in a compiler the syntactical analysis happens only once—during compilation.

To turn our interpreter into a compiler, we assure that the syntactical analysis is computed only once. Figure 2 shows the compiler that emanates from our interpreter. The results of the syntactical analysis is bound in the closure of the implementation function, thus occurs only once when the implementation function is created.

We used the compilation method outlined in this section to write a compiler that handles almost the complete Emacs Lisp language. In place of the naive implementation of dynamic binding presented in the examples of this section we use a more efficient technique described in Section 3.2. The implementation

```

(define (compile e)
  (cond
    ((number? e)
     (lambda (env) e))
    ((symbol? e)
     (lambda (env)
       (cdr (assoc e env))))
    ((lambda? e)
     (let ((arg (lambda-arg e))
           (body (compile (lambda-body e))))
       (lambda (env)
         (lambda (a)
           (body (cons (cons arg a) env)))))))
    ((funcall? e)
     (let ((op (compile (funcall-op e)))
           (arg (compile (funcall-arg e))))
       (lambda (env)
         ((op env) (arg env)))))))

```

Figure2. A metacircular compiler

functions are written in continuation passing style to facilitate the task of implementing an exception system and constructs related to control like `while` and `cond`.

We have also implemented a compiler front-end consisting of a reader and a simple recursive descent parser that is capable of handling the various syntactic notions that have been added to Emacs Lisp over the time. The parser generates an abstract syntax tree represented by records [7].

2.2 Controlling the compiler's output

The output of our compiler is available in two different forms: Scheme source text or as a value, which means a Scheme function directly useable at the prompt of the Scheme system running the compiler. In an interactive setting a representation as a Scheme value is more appropriate than a textual representation, since reading and parsing the Scheme source text can be omitted completely. On the other hand the generated code in text form is useful for debugging the compiler or running the code on other Scheme implementations. In this section we introduce `compilation-form`, a macro that allows us to produce both output forms from one implementation function. We exemplify the functionality of `compilation-form` with the compilation function for `if`.

Figure 3 shows the compilation function for `if` without the usage of `compilation-form`. Compilation functions like `analyze-if` usually have two arguments, a compile time environment `cte` that holds information about the compilation process, such as the symbol table, and a node `exp` of the abstract syntax tree which represents the code to be compiled. The nodes in the abstract syntax tree

```

(define (analyze-if cte exp)
  (let ((test (analyze cte (ast-el-if-condition exp)))
        (consequent (analyze cte (ast-el-if-then-form exp)))
        (alternative (analyze cte (ast-el-if-else-forms exp))))
    (lambda (rte k)
      (test rte
        (lambda (bool)
          (if (el-truish? bool)
              (consequent rte k)
              (alternative rte k)))))))

```

Figure3. Compiling if.

are represented as records; function names beginning with `ast-el-` are selectors for these records. The function `analyze` compiles a node of the abstract syntax tree to the accordant implementation function (thus it serves the same purpose as `compile` in Figure 2). An `if` expression contains three subexpressions: a test, an expression evaluated if the test evaluates to `nil`, and an expression that is evaluated otherwise. For each subexpression we call `analyze`, thus create implementation functions. These implementation functions are stored in the closure of the implementation function for `if`.

The compilation function in figure 3 evaluates to a Scheme function. To turn this compilation function into a source text generating compilation function, we use quasiquotation. Wrapping the implementation function in Scheme's `quasiquote` macro, results in quoted list instead of a function. In fact, we have to change every implementation function this way. Now, converting the quoted list to a text representation is straightforward.

Of course, this method causes a problem: The source text issued references `test`, `consequent` and `alternative`—originally these variables were implementation functions for subexpressions bound in the closure of the implementation function. Since the compilation function returns a list of symbols, no closure is created, thus these variables are suddenly free. To overcome this problem we need to `unquote` the occurrences of these variables. Assuming that we quoted all implementation functions in our compiler, we thereby insert the code of the implementation functions in place of the reference of a variable.

Since the original plan was to have a compiler that emits both formats, we introduce a macro called `compilation-form` to abstract over the modification just presented. By providing two implementations of this macro, one that wraps implementation functions in `quasiquote` and one that does nothing, we may control the compiler's output format. Figure 4 shows the compilation function for `if` using `compilation-form`.

In the case of generating source text, `quasiquote` and `unquote` provide a notion for controlling at which point in time a computation happens. Unquoting an expression in a quasiquoted implementation function is a synonym for evaluation at compile time. Accordingly, there is also a need to distinguish `compile` and

```

(define (analyze-if cte exp)
  (compilation-form
   (lambda (rte k)
     ((compile-time (analyze cte (ast-el-if-condition exp)))
      rte
      (lambda (bool)
        (if (el-truish? bool)
            ((compile-time
              (analyze cte (ast-el-if-then-form exp)))
             rte k)
            ((compile-time
              (analyze cte (ast-el-if-else-forms exp)))
             rte k)))))))

```

Figure 4. Compiling if to multiple output formats.

evaluation time in an implementation function written with `compilation-form`. This differentiation is done by the special marker `compile-time` which is available in the scope of a `compilation-form`.

The implementation of `compilation-form` for source code generation leads to a re-implementation of `quasiquote`. In the other case, when generating functions as Scheme values, the compile-time values need to be stored in closure of the implementation function. Therefore, `compilation-form` binds each compile-time value to a fresh variable name in the closure of the implementation function. An implementation of `compilation-form` for generating source code is included in appendix A. We used Clinger's explicit renaming [4] low level macro system to implement `compilation-form`.

2.3 Functions and function calls

Section 2.1 shows how functions and function calls of the source language may be expressed metacircularly through functions and function calls in the implementation language. This was especially concise because the source language only had functions with a fixed arity of one to its disposal. The signature of Emacs Lisp functions are considerably more complicated, thus require a refinement of our compilation technique.

The argument list of an Emacs Lisp function consists of an arbitrary fixed number of mandatory arguments followed by an arbitrary fixed number of optional arguments and a optional rest list argument as described by the following grammar:

$$\begin{aligned}
 m, o, r &\in \mathbf{VarNames} \\
 \langle \textit{argument - list} \rangle &::= m^* (\&\textit{optional } o^+)^? (\&\textit{rest } r)^?
 \end{aligned}$$

The set `VarNames` contains all valid variable names, *m* stands for mandatory arguments, *o* for optional arguments and *r* for the rest-list argument. In

```

(lambda (rte k)
  (k (let ((ctv-4 (let ((ctv-3 (el-symbol-value %y))
                       (ctv-2 (el-symbol-value %x))
                       (ctv-1 (el-symbol-function %fun))))
          (lambda (rte k)
            (ctv-1 rte stop-k ctv-2 ctv-3))))
      (ctv-3 (%z))
      (ctv-2 %y)
      (ctv-1 %x))
    (lambda (rte k a0 a1 . more)
      (let ((scope (current-dynamic-scope)))
        (bind-dynamic-scope! rte ctv-1 a0)
        (bind-dynamic-scope! rte ctv-2 a1)
        (bind-optional-params! rte ctv-3 #f more)
        (ctv-4 rte
          (lambda (return-value)
            (install-dynamic-scope! rte scope)
            (k return-value))))))))))

```

Figure 5. Code generated for `(lambda (x y &optional z) (fun x y))`.

comparison with Emacs Lisp, the signatures of Scheme functions are more compact. Scheme functions either have a fixed number of mandatory arguments or a fixed number of mandatory arguments plus a rest list argument.

A simple method for implementing Emacs Lisp function calls in Scheme is to create a Scheme list containing all arguments and pass this list as the only argument to the function. The function traverses this list to match parameters with argument values. However, this solution is quite inefficient—function calls are very common in Emacs Lisp programs and creating and traversing an argument list is rather expensive.

Therefore, our compiler generates a n -ary Scheme function for an Emacs Lisp function with n mandatory arguments. If an Emacs Lisp function has optional arguments or uses a rest list argument, we add a rest list argument to the Scheme function. Hence, the optional arguments are passed as the rest list argument of the Scheme function.

Figure 5 shows the implementation function for an Emacs Lisp function with two mandatory arguments `x` and `y` and one optional argument `z`. Implementation functions have two additional arguments; `rte` is a reference to current run-time environment (see Section 3 for details) and `k` is the continuation. Variables with names beginning with `ctv-` come from the `compilation-form` mechanism introduced in Section 2.2.

Besides the two standard arguments `rte` and `k`, the implementation function in our example function has the additional arguments `a0` and `a1` and a rest-list argument `more`. `a0` and `a1` are used to pass the mandatory arguments `x` and `y`, the optional argument `z` is passed via the rest-list argument `more`. Binding an argument to a value in Emacs Lisp actually means setting the value slot of symbol

to the value. (Section 3.1 describes Emacs Lisp symbols and their representation at run-time in detail.) This work is done by `bind-dynamic-scope!` which additionally does some bookkeeping needed for implementing dynamic binding (see Section 3.2 for details). The function `bind-optional-params!` traverses the list of optional argument values given by `more` and binds them to the accordant optional arguments.

The above compilation method for function calls is more efficient than using an ephemeral list to pass the argument values. However, it is harder to implement. For compiling a n -ary Emacs Lisp function the compiler needs to create an implementation function with $n + 2$ arguments. In the case of a compiler that emits source text solely, this is straightforward, since the compiler basically writes the function definition in text form, thus adding an argument is no problem. However, in our setting the compiler creates the implementation function in form of a Scheme value. This changes the situation: The arity of the implementation function which the compiler has to return is first known at the compiler's run-time—too late to change anything, since the arity of `lambda` expressions in Scheme is fixed at compile time. In this case this is the compile time of our compiler.

There are two possible solutions for this problem. Writing all functions and function calls in a Curry notation is one solution. Generating a series of functions with ascending number of arguments up to an arbitrary number in advance at the compiler's compile time is a second solution.

Compiling curried functions and applications efficiently is a delicate task, that generally speaking, assumes that it is known which function is called at a specific function call. Since the Scheme 48 byte-code compiler does not carry out the analysis required for acquiring this knowledge, the ephemeral functions introduced by currying the code will cause the creation of an accordant number of closures at run-time. In comparison with the metacircular translation presented in this section where no ephemeral closures need to be created, this is rather slow and especially in the common case of function applications not acceptable.

Therefore, we generate the implementation functions up to an arbitrary maximum number of arguments at the compiler's compile time. The following simplified excerpt is taken from `compile-function`, the part of our compiler that compiles functions:

```
(case (length required-params)
  ...
  ((1) (cond
        (have-optionals?
         ...
         (lambda (rte k a0 . more) ...))
        (else
         ...
         (lambda (rte k a0) ...))))
  ((2) (cond
        (have-optionals?
```

```

      ...
      (lambda (rte k a0 a1 . more) ...)
    (else
      ...
      (lambda (rte k a0 a1) ...)))
  (else (lambda arglist ...)))

```

In this case the compiler has implementation functions up to a maximum number of two arguments to it's disposal. At run-time of the compiler the `case` expression checks the count of mandatory arguments of the Emacs Lisp function to compile. A local variable `have-optionals?` indicates whether the signature contains optional arguments. Based on this, the compiler returns the appropriate implementation function. If the argument count of an Emacs Lisp function exceeds the maximum number of arguments, the protocol for passing arguments changes. In this case the arguments are passed as a list (see above), therefore the implementation function returned is a Scheme function accepting an arbitrary number of functions.

We generate the `case` expression up to an arbitrary number of branches using a macro. Thus, generating implementation functions in advance up to a limit sufficiently high is no extra work. We used Clinger's explicit renaming low-level macro system [4] for implementing the macro, since this macro system lightens the task of generating fresh identifiers.

3 Run-time system

Some features and characteristics of Emacs Lisp, such as variable lookup, symbol magic and dynamic binding require the presence of a run-time system. In practice, the border between the run-time system needed for implementing Emacs Lisp as a general purpose programming language and the internal state of Emacs as an editor is blurry. We focus on implementing the language specific part of the run-time system. The part of the run-time system reflecting the editor's state is currently implemented in a minimalist way. In the long run this part of the run-time system is to interface the internal state of a real Emacs editor. This section outlines the situations where a run-time system is required and the tasks the run-time systems fulfills.

3.1 Symbols

Symbols are objects with an unique name and are of fundamental concern for Emacs Lisp [19]. This section summarizes the semantics of symbols and describes our implementation approach.

Primarily, symbols are used as variables and function names. A symbol is composed of four *slots* which serve different purposes. The *value slot* holds the value of the symbol as a variable. The *function slot* holds a function. The *property list* saves a list of pairs with information on the symbol, such as a documentation

string for instance. The *print name* slot stores a string with the name used for reading and printing the symbol.

The separate slots for functions and values point up the separate namespaces for functions and values in Emacs Lisp. For example, the expression `(f f)` applies the function stored in the function slot of the symbol `f` to the value from its value slot.

Symbol magic adds extra complexity to the picture. Function slot and value slot may independently act as aliases, thus reference a second symbol's function or value slot. A lookup operation has to chase the indirections until it encounters a value or function. Furthermore, it is possible to make a variable *buffer-local*, that is to say, the value stored in the value slot depends on the currently active buffer of the Emacs editor. GNU Emacs also features *frame-local* variables, where the value depends on the active editor window. By *interning*, a symbol may be referenced or added given a string: `(intern "foo")` returns the existing symbol named `foo` or creates a new symbol if there is no symbol with such name.

Having to regard all these features, variable lookup in general becomes a fairly complex operation. We implement symbols as records with a field for each slot plus an extra boolean field that indicates whether magic has been used on the symbol. Therefore a lookup operation in the common case (no magic has been used on the symbol) only needs to access two fields. The compiler generates a global variable bound to such a record for each symbol used in the source program. In addition the run-time system maintains a *symbol table* that maps symbol names as strings to the global variables representing the symbols. Thus, interning a symbol corresponds to a lookup or extension in the symbol table.

To implement buffer-local variables, the run-time system maintains a list of buffers (as a Scheme list) and a table of buffer-local symbols. Each entry in this table stores an association list that maps a buffer name to the symbol's value for this buffer. At the moment, there is no support for frame-local variables in our run-time system.

3.2 Implementing dynamic binding

Emacs Lisp uses dynamic binding [18] as the default binding scheme for all variables. In a language with dynamic binding, a variable always refers to the most recent binding at run-time. Most modern programming languages implement lexical binding as the default binding scheme. In a few contexts, such as exception handling or parameterization, a careful and limited use of dynamic binding is considered convenient. Thus, modern programming languages support dynamically scoped variables as a library [9] or using a special syntactical notion [16, 13]. However, dynamic binding as the default binding scheme is disadvantageous; it restricts modularity and limits the ability to build abstractions [14].

There are several ways to implement dynamic binding. The metacircular interpreter shown in figure 1 of section 2.1 uses *deep binding*. However, our run-time systems uses *shallow binding* [1] to implement dynamic binding.

4 Primitive functions

Besides code written in Emacs Lisp the GNU Emacs and XEmacs code base also contains about 1800 *primitive functions* implemented in C. Using a foreign function interface, Emacs Lisp programs call primitive functions like any Lisp function. Primitive functions present a major challenge for a successful migration. We divide primitive functions into three classes:

- true primitive functions
- primitive functions rewritten in C for performance reasons
- stubs for foreign function calls

A *true primitive function* implements a functionality that can not be implemented without access to the compiler's run-time representation of values; `car` and `cdr` are examples for this class. The Emacs Lisp interpreter inside Emacs represents Emacs Lisp values as C structs, since the interpreter itself is written in C. Primitive functions of this class need to be re-implemented to work with run-time representation as Scheme values in our compiler.

In the course of time, functions implemented in Emacs Lisp have been rewritten to C code to bypass the poor performance of the Emacs Lisp implementation. A typical example for a function of this class is `nth`, which returns the object at a given index of a list. This function could be easily be implemented in Emacs Lisp using recursion and the true primitive `cdr`. In this case, the lack of proper tail calls and the overhead necessary to re-root the dynamic environment are to blame for the poor performance of the Emacs Lisp version. So far, we have re-implemented a small subset of functions belonging to this class in Scheme. This work is also eased by the fact that many primitive functions have a counterpart available in a Scheme library. Compiling still available Emacs Lisp implementations of the primitive functions is also an option.

The Emacs foreign function interface has also been used to implement bindings to several external C libraries such as a LDAP client library for example. In this case, the primitive functions serve as *stubs for foreign function calls*, thus, provide only a possibility for calling the external C function from Emacs Lisp. This approach is also known to many Scheme implementations. In fact, many Scheme implementations provide bindings for those libraries in a very similar fashion [10] which may be re-used. However, a practical solution for this class of primitive functions is part of future work.

5 Related work

There have been several attempts to develop an Emacs editor with a new extension language. Edwin [2] is a general-purpose text editor that uses a Scheme dialect as its extension language and offers an interpreter for running Emacs Lisp code. JEmacs [3] is the prototype of a re-implementation of the GNU Emacs in Java using Emacs Lisp and Scheme as its extension languages.

6 Conclusion and future work

Currently our compiler handles the core Emacs Lisp constructs. To complete the compiler we plan to add a macro expander and add support for exceptions. For a compiler suitable for practical use, macro expansion is a crucial feature. We plan to integrate the macro expander with the parser directly.

The primitive functions discussed in Section 4 remain a special problem. Though, it is clear how it is possible to proceed to make primitive functions available to our compiler, this imposes a tremendous amount of work. We have to hope for support by the Emacs developer communities here.

A version of GNU Emacs or XEmacs that replaces its Emacs Lisp interpreter by the compiler we proposed will only be accepted by users if Emacs Lisp code runs at acceptable speed. So far, we have done a few micro-benchmarks, which look promising. However, we have no measurements with meaningful benchmarks, such as the Gabriel's benchmark suite [6]. Currently, the lack of a macro expander prevents the compilation of the Common Lisp macros which is a prerequisite for compiling the Gabriel benchmarks.

Finally, the Emacs Lisp interpreter inside XEmacs needs to be replaced by the Scheme 48 virtual machine, a task that probably will be accompanied by pragmatical problems.

Acknowledgments I thank Martin Gasbichler and Michael Sperber for the helpful discussions about my work leading to many improvements and insights.

References

1. Henry G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978.
2. Matthew Birkholz. Emacs Lisp in Edwin Scheme. Artificial Intelligence Lab Report TR-1451, MIT, September 1993.
3. Per Bothner. JEmacs-the java/scheme-based emacs. In *Proceedings of the 2000 Usenix conference*, pages 271–278, June 2000.
4. William Clinger. Hygienic macros through explicit renaming. (from the Scheme repository).
5. Marc Feeley. Deux approches à l’implantation du langage Scheme. Master’s thesis, Université de Montréal, 1986.
6. Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, MA, 1985.
7. Richard Kelsey. SRFI 9: Defining record types. <http://srfi.schemers.org/srfi-9/>, September 1999.
8. Richard Kelsey, William Clinger, and Jonathan Rees. Revised⁵ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.
9. Richard Kelsey and Jonathan Rees. *Scheme 48 Reference Manual*, 2002. Part of the Scheme 48 distribution at <http://www.s48.org/>.
10. Richard Kelsey and Michael Sperber. SRFI 50: Mixing scheme and c. <http://srfi.schemers.org/srfi-50/>, December 2003.
11. Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.
12. Bil Lewis, Dan LaLiberte, Richard Stallman, and the GNU Manual Group. GNU Emacs Lisp reference manual. <http://www.gnu.org/manual/elisp-manual-20-2.5/elisp.html>, 1785.
13. Jeffery R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: Dynamic scoping with static types. In Tom Reps, editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, pages 108–118, Boston, MA, USA, January 2000. ACM Press.
14. Matthias Neubauer. Dynamic scope analysis for Emacs Lisp. Master’s thesis, Eberhard-Karls-Universität Tübingen, December 2000. <http://www.informatik.uni-freiburg.de/~neubauer/diplom.ps.gz>.
15. Matthias Neubauer and Michael Sperber. Down with Emacs Lisp: Dynamic scope analysis. In Xavier Leroy, editor, *Proceedings of the 2001 International Conference on Functional Programming*, pages 38–49, Florence, Italy, September 2001. ACM Press, New York.
16. Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1994.
17. Richard Stallman. GNU extension language plans. Usenet article, October 1994.
18. Guy L. Steele Jr. and Richard P. Gabriel. The evolution of Lisp. In Jean E. Sammet, editor, *History of Programming Languages II*, pages 231–270. ACM, New York, April 1993. SIGPLAN Notices 3(28).
19. Ben Wing. XEmacs Lisp Reference Manual. <ftp://ftp.xemacs.org/pub/xemacs/docs/a4/lispref-a4.pdf>.gz, May 1999. Version 3.4.

A A macro for emitting source code

```
(define-syntax compilation-form
  (lambda (e r c)
    (let ((%quasiquote (r 'quasiquote))
          (%unquote (r 'unquote)))

      (define fresh-name
        (let ((count 0))
          (lambda ()
            (set! count (+ 1 count))
            (r (symbol-from-number "ctv-" count))))))

      (define (transform arg static-bindings)
        (cond
          ((pair? arg)
           (if (c (car arg) (r 'compile-time))
               (let ((temp (fresh-name)))
                 (values temp
                         '((,temp (,%unquote ,(cadr arg))) .
                          ,static-bindings)))
               (call-with-values
                (lambda ()
                  (transform (car arg) static-bindings))
                (lambda (car-code car-static-bindings)
                  (call-with-values
                   (lambda ()
                     (transform (cdr arg) car-static-bindings))
                   (lambda (cdr-code car+cdr-static-bindings)
                     (values (cons car-code cdr-code)
                             car+cdr-static-bindings)))))))
           (else
            (values arg static-bindings))))

      (call-with-values
       (lambda ()
         (transform (cadr e) '()))
       (lambda (code static-bindings)
         '(%quasiquote (let ,static-bindings ,code))))))
```