Journal of Universal Computer Science, vol. 16, no. 2 (2010), 271-295
submitted: 19/10/09, accepted: 10/12/09, appeared: 28/1/10 © J.UCS

Embedding Hygiene-Compatible Macros in an Unhygienic Macro System

Pascal Costanza and Theo D'Hondt
(Vrije Universiteit Brussel, Belgium
Pascal.Costanza@vub.ac.be and Theo.D'Hondt@vub.ac.be)

Abstract: It is known that the essential ingredients of a Lisp-style unhygienic macro system can be expressed in terms of advanced hygienic macro systems. We show that the reverse is also true: We present a model of a core unhygienic macro system, on top of which a hygiene-compatible macro system can be built, without changing the internals of the core macro system and without using a code walker. To achieve this, the internal representation of source code as Lisp s-expressions does not need to be changed. The major discovery is the fact that symbol macros can be used in conjunction with local macro environments to bootstrap a hygiene-compatible macro system. We also discuss a proof-of-concept implementation in Common Lisp and give historical notes.
Key Words: Hygiene-compatible macro systems, Scheme, Common Lisp
Category: D.1.1, D.3.3, D.3.4

1 Introduction

Macros are local program transformations triggered explicitly in the source code of a program. Since their introduction into Lisp in 1963 [Hart 63], they have found their way into many Lisp dialects, including Common Lisp [ANSI 94], ISLISP [ISO 97] and Scheme [Kelsey et al. 98]. Lisp dialects are especially attractive for macros due to Lisp's 'homoiconic' nature: Lisp source code is constructed from s-expressions, that is lists, symbols and (literal) values, which are all core data types of Lisp itself [Kay 69, McIlroy 60]. Therefore it is straightforward to express Lisp macros as functions that simply map s-expressions to s-expressions. Compared to mere string replacement systems, like in C macros, s-expressions naturally represent the structure of nested language constructs. For this reason, Lisp-style macros are also sometimes called *structural macros*.

A Lisp compiler or interpreter recognizes macro invocations based on definitions of such macro functions which are in scope, calls such macro functions with s-expressions representing source code fragments, and uses the resulting s-expressions in place of the original ones for further processing. This process of replacing s-expressions with new ones is called macroexpansion.

Macros are useful for adding new language constructs, for expressing domain-specific languages and for controlling evaluation of arguments in an otherwise strict language, among other uses [Graham 93]. The operators one can define as macros are essentially 'just' syntactic sugar. Nevertheless, they are deemed as critical productivity enablers among users of Lisp dialects. Furthermore, the strictly local nature of macro expansion has led to a useful formal characterization of the expressive power of language constructs [Felleisen 91].

One of the issues related to macros that has been researched in depth is that of macro hygiene. Since the initial macro systems for Lisp have operated on 'raw' s-expressions, variable names that are introduced and/or referenced in the result of a macroexpansion are susceptible to inadvertent capture by introductions and/or references in the surrounding code of the macro invocation. Bawden and Rees introduce the following macro definitions in a hypothetical dialect of Scheme to illustrate inadvertent variable capture [Bawden and Rees 88]:

```
(define-macro (push obj-exp list-var)
  `(set! ,list-var (cons ,obj-exp ,list-var)))

(define-macro (or exp-1 exp-2)
  `(let ((temp ,exp-1))
     (if temp temp ,exp-2)))

(define-macro (catch body-exp)
  `(call-with-current-continuation
     (lambda (throw) ,body-exp)))
```

Here, the formal parameters in the macro definitions are bound to source code fragments represented as s-expressions, and the bodies of the macro definitions are functions that construct new s-expressions by way of quasiquotation. Quasiquote (`` ` ``) is similar to quote (`'`) in Lisp and Scheme, that is, it prevents the subsequent form from being evaluated. However, quasiquote enables marking subforms of the quasiquoted form to be evaluated and possibly spliced in the immediately surrounding form by way of comma (`,`) and comma-at (`,@`). See [Bawden 99] for more details about quasiquotation.[1]

Bawden and Rees continue to give the following two examples of uses of these macros.[2] The first one illustrates the inadvertent capture of free symbols in the result of a macroexpansion:

```
(let ((cons 5))
  (push 'foo stack))
```

...expands into...

```
(let ((cons 5))
  (set! stack (cons 'foo stack)))
```

[1] There are variations in the semantics of quasiquotation in different Lisp dialects, but since we use only simple examples of quasiquotation throughout this paper, these variations do not matter here. Quasiquote is historically also known as *backquote*.

[2] They actually give four examples, but the two examples we do not list here are mere variations of the same problems.

Here, the macro `push` expands into code that attempts to use the `cons` function for creating pairs, as predefined in Scheme. However, the code in which the invocation of `push` is macroexpanded creates a new binding for `cons` which is clearly not the one that `push` intends to use. Nevertheless, the reference to `cons` in the result of the macroexpansion inadvertently sees the new binding.

The second example illustrates the inadvertent capture of symbols passed as arguments to a macro:

```
(or (memq x y) temp)
```

...expands into...

```
(let ((temp (memq x y)))
  (if temp temp temp))
```

Here, the macro creates a binding for a temporary variable `temp`. However, the macro invocation itself attempts to use another variable `temp` that is presumably already defined in the surrounding code of the macro invocation. Due to the placement of the arguments to the macro invocation in the macroexpanded code, the latter reference to `temp` sees the binding created by the macro, which is clearly not the one that the user of the `or` macro intended.

The terms *free symbol capture* for the first kind of variable capture and *macro argument capture* for the second kind are due to Graham [Graham 93].

Macro argument capture is straightforward to prevent: A macro just has to make sure that variable names it introduces are unique and cannot be inadvertently captured by other code. For that purpose, most Lisp dialects provide a `gensym` function that generates symbols that are guaranteed to be unique: Such symbols cannot be accidentally typed in as regular source code tokens, and consecutive invocations of `gensym` are guaranteed to yield different symbols.[3]

So by providing the following new definition for the `or` macro above, the unintended capture of macro arguments can be avoided in this example. This generalizes to all kinds of macro argument capture in a straightforward way.

```
(define-macro (or exp-1 exp-2)
  (let ((temp (gensym)))
    `(let ((,temp ,exp-1))
       (if ,temp ,temp ,exp-2))))
```

However, traditionally the kind of macro system sketched so far does not provide a systematic solution for the case of free symbol capture.

[3] This can, for example, be achieved by generating symbol names that are not accepted by the parser, or by relying on the object identity of symbols for their comparison instead of their names. In the latter case, `gensym` generates fresh symbol objects on each invocation. A parser then has to ensure that, while reading source code, the same names are always mapped to the same symbols. Since macros are expanded only after parsing, `gensym` does not interfere with it. See [Graham 93] for more details.

Consider the following code fragment:

```
(let ((x 42))
  (macrolet (((foo) 'x))
    (let ((x 4711))
      (foo))))
```

Here, the local macro foo presumably wants to expand into a reference to the outer x variable. However, the invocation of foo in this code fragment will eventually expand into a reference of the inner x, making the overall code fragment evaluate to 4711. This is a very compact example of free symbol capture.

Some workarounds are suggested in the literature for inadvertent variable capture, like naming conventions, rearranging the results of macroexpansion, and so on. See [Graham 93] for a comprehensive overview. However, especially the solutions for free symbol capture are ad hoc and do not generalize well.

Finally, Bawden and Rees give the following example to illustrate that variable capture may be intentional:

```
(catch (+ 5 (throw 'x)))
```

... expands into ...

```
(call-with-current-continuation
  (lambda (throw) (+ 5 (throw 'x))))
```

Here, catch is a construct that provides an escape in a way similar to the throw/catch constructs in traditional Lisp dialects. It implicitly creates a binding for the variable throw, and that is the very purpose of this macro. This kind of intentional capture by macros is not uncommon.

The fact that free symbol capture does not have a straightforward solution, and that both unintentional variable capture should be avoided but intentional variable capture enabled, has led to extensive research on macro hygiene especially in the Scheme community.[4] There are two ways of achieving macro hygiene: *Hygiene-compatible macro systems* essentially stick to the kind of macro system sketched so far, but provide additional operators to help avoid free symbol capture manually [Bawden and Rees 88, Clinger 91a]. *Hygienic macro systems*, on the other hand, introduce different models of macroexpansion. They ensure that locally visible bindings are automatically respected - macro arguments refer to the bindings of the macro invocation site and free symbols refer to the bindings of the macro definition site - and add means for intentionally breaking macro hygiene [Dybvig et al. 92]. This eases expressing simple macros compared to traditional Lisp-style macro systems. However, more involved macros become more

[4] Macro hygiene has been considered a less important problem in the Common Lisp community mostly due to the separation of function and variable namespaces in Common Lisp, which avoids most of the practically occurring capture problems in macros. See [Gabriel and Pitman 88] for a detailed discussion.

complex, due to the fact that the latter approach differentiates between surface syntax, which is still represented as s-expressions, and internal representation of source code in terms of *syntax objects*. This leads to a system in which the 'homoiconicity' of traditional Lisp macros is lost and, in some cases, code fragments have to be manually mapped between the different representations in macro definitions, for example for the purpose of breaking macro hygiene. The hygienic macro system adopted in R5RS Scheme even goes as far as to disallow intentional breaking of macro hygiene [Kelsey et al. 98]. This removes the burden introduced by syntax objects, but makes it impossible to express certain kinds of macros [Dybvig et al. 92].

It has indeed been suggested that in order to support macro hygiene, the internal representation of source code has to be changed. For example, Rees discusses a few alternatives for implementing hygienic macro systems, which all rely on introducing new data types for the internal representation of source code that differ from the data types used for the surface syntax [Rees 93]. On the other hand, Clinger claims that "if a macro needs to refer to a global variable or function [...], then it is quite impossible to write that macro reliably using the Common Lisp macro system" [Clinger 91b]. Since Common Lisp's macro system is modelled after the traditional Lisp-style approach sketched above, this seems to suggest, in other words, that an unhygienic macro system cannot support macro hygiene for both macro argument capture and free symbol capture.
In this paper, we make the following contributions.

- It is known that the essential ingredients of an unhygienic macro system can be expressed in terms of advanced hygienic macro systems [Sperber et al. 07]. We show that the reverse is also true: The essential operators of hygiene-compatible macro systems, as discussed in the literature [Clinger 91a], can be expressed in terms of an advanced unhygienic macro system.

- We show that for this, the internal representation of source code in the form of s-expressions does not need to be changed. The major discovery is the fact that symbol macros can be used in conjunction with local macro environments to bootstrap a hygiene-compatible macro system.

- We present an implementation of our approach in Common Lisp that does not require a code walker and has a fully portable implementation.

On the other hand, we also show how the unhygienic and hygiene-compatible macro systems presented in this paper are incompatible with each other.

This paper is structured as follows. In Section 2, we present the essential elements of an advanced unhygienic macro system. In Section 3, we bootstrap a hygiene-compatible macro system on top of the unhygienic macro system of Section 2. The essential idea here is that each identifier gets both an external and an internal name, and higher-order macros are used to map from potentially ambiguous external names to guaranteed unique internal ones. This can be achieved without changing the internals of the unhygienic macro system of Section 2 and without involving a code walker. In Section 4, we discuss an integration of this paper's approach into Common Lisp. The latter is relatively straightforward because all ingredients described in Section 2 already exist there. In Section 5, we provide some historical remarks and discuss related work, before we conclude and present future work in Section 6.

2 An Unhygienic Macro System

In this paper, we use an effect-free[5] subset of Scheme for developing both a core unhygienic macro system, as well as the hygiene-compatible macro system built on top in the next section to back our claims. We use Scheme to be able to focus on the essential elements of our approach before discussing a more complete implementation in Common Lisp in Section 4. The following constructs from Scheme are sufficient for this paper:

- `define`, `let`, `let*` and `letrec` for introducing new variables.
- Functions with `lambda` for defining functions, function application, and `map` for mapping functions over lists.
- `begin` for sequencing expressions.
- Boolean values with operators `not`, `or` and `and`, and conditional expressions `if`, `cond` and `case`.
- Symbols with `symbol?` for testing for symbols, and `gensym` for creating unique symbols (see above).
- Pairs with `cons`, `list`, `append` and `reverse` for constructing pairs/lists; `car`, `cdr`, `cadr`, `cddr`, and `caddr` for accessing elements in pairs/lists; `pair?` for testing for pairs; `null?` for testing for empty lists; and `assoc` for treating lists as association lists.
- `eq?` for testing for object identity.
- `quote` for preventing a subsequent form from being evaluated.
- `eval` for evaluating a form. The latter is used in this paper only for converting lambda forms into functions.

[5] i.e., without assignments and first-class continuations

For most of the constructs used in this paper, the definitions given in any of the recent Scheme reports are sufficient, except for `gensym`, which is not part of any Scheme report, but has been characterized in Section 1 and is provided by many Scheme implementations.

Both our unhygienic and hygiene-compatible macro systems correctly expand the arguments to `set!` in user programs, but themselves do not use side effects in their expansion algorithms. We do not provide syntactic sugar for destructuring macro arguments and constructing resulting s-expressions, since this does not affect the core issues addressed in this paper. We do use (simple forms of) quasiquotation in our macro systems and in example macro definitions, but we consider this part of the metalanguage which is assumed to be manually translated into invocations of `list`, `cons` and `quote`. An integration of full quasiquotation is extensive but straightforward.

2.1 Required Elements

The macro system in this section provides the following elements which are required to build a hygiene-compatible macro system on top in Section 3.

List macros are regular macros, as for example illustrated in Section 1. They define the expansion of list forms, like in `(push 'foo stack)`, where `push` is the first, `'foo` is the second and `stack` is the third element of the given list. In the literature, they are typically called just *macros*, but we want to explicitly distinguish them here from symbol macros.

Symbol macros are macros that define the expansion of symbol forms. They are, for example, part of ANSI Common Lisp [ANSI 94] and R6RS Scheme (there called *identifier macros* [Sperber et al. 07]). Symbol macros are, for example, used in object-oriented extensions for Lisp and Scheme to introduce convenience syntax for accessing members of the implicit variables `this` or `super`. They typically expand into procedure calls, for example as in `name` expanding into `(person-name this)`.

Both list and symbol macros are introduced as *local macros*, which are affected by other local macros of the surrounding scope. To illustrate this, consider the following hypothetical code fragment, where a local list macro `foo` is defined:

```
(let ((x 42))
  (macrolet (((foo) (if (< x 50) '(print #t) '(print #f)))) ;; buggy
    (foo)))
```

In lexically scoped Lisp dialects, we would expect that the macro definition sees the variables from the surrounding code (like `x`). However, one important goal of macro systems is that macros can be fully expanded at compile time, before a program is executed. In other words, macro definitions cannot see runtime bindings of local variables, so the definition of `foo` above is invalid as is.

However, things are different for macro definitions in the surrounding code:

```
(macrolet (((x) 42))
  (macrolet (((foo) (if (< (x) 50) '(print #t) '(print #f)))) ;; correct
    (foo)))
```

Since the macro `x` in this version is also available at compile time, `foo` can indeed see and use it. The local invocation of `(foo)` thus expands into `(print #t)`.

Finally, we require low-level functions with which macros can be expanded explicitly, like Common Lisp's `macroexpand`. They are typically used for interactively testing macro definitions, but they also have uses in advanced macro programming. For example, consider a `with-lock` macro that is used in a hypothetical library for mulithreading, for locking an object during the extent of a block of code. However, if that block of code actually does not contain any references to that object, the potentially costly locking operation can be avoided:

```
(define-macro (with-lock variable block)
  (if (contains-reference? variable (expand-all block))
     (generate-costly-lock-operation variable block)
     block))
```

Here, macroexpansion of the `block` parameter (via `expand-all`) is necessary because `block` may contain macro invocations resulting in references to `variable` that would otherwise remain undetected. Due to local macro definitions, however, such low-level macroexpansion functions require representations of local macro environments to be passed. Consider the following example:

```
(define-macro (foo) '(display x))

(macrolet (((foo) '(display y)))
  (with-lock y (foo)))
```

Here, the invocation of `(foo)` will expand into code that references the variable `y`, due to the local redefinition of `foo`. This means that in the definition of the `with-lock` macro, we somehow need to capture a representation of the locally effective macro definitions and pass this to `expand-all` to ensure that it does not erroneously expand the global definition of `foo`.

To summarize, we require the following elements:

- List and symbol macros.
- Local macros, which are affected by surrounding local macros.
- Macro expansion functions which operate on local macro environments.

2.2 A Model of Macroexpansion

In our macro system, macro environments are represented as assocation lists that map macro names to tags and expansion functions. A tag is either the symbol `list-macro` or `symbol-macro`, indicating what kind of macro is bound to the respective macro name. Macro expansion functions take two parameters: the form to be expanded and a macro environment. Macros are introduced using an `expander-let` form. Consider the following local macro definition fragments:

```
(expander-let
  ((foo list-macro (lambda (form env) ... 1 ...)))
  (expander-let
    ((bar symbol-macro (lambda (form env) ... 2 ...)))
    (expander-let
      ((baz  list-macro  (lambda (form env) ... 3 ...)))
      ... enclosed code ...)))
```

These definitions create the following local macro environment:

```
((baz list-macro    ... function 3 ...)
 (bar symbol-macro  ... function 2 ...)
 (foo list-macro    ... function 1 ...))
```

Macro environments list inner before outer definitions, to aid `assoc` finding the innermost macro definition for a given name. Based on this data structure, we can now define the core macro system in Figure 1. It consists of three mutually recursive functions `expand-once`, `expand` and `expand-all`, and three helper functions `bind-expander`, `flatten-parameters` and `bind-variable`. Each of the three expansion functions takes a macro environment and a form to be expanded as parameters. For convenience, these functions are curried.

The function `expand-once` performs one step of macro expansion, based on the given macro environment, in case it successfully determines that the passed form is indeed a macro invocation of the appropriate kind (`symbol-macro` or `list-macro`). If it is not a macro invocation, or not applicable to the given form, the form is simply returned without change. The function `expand` repeatedly invokes `expand-once` on the passed macro environment and form until the consecutive forms yielded by `expand-once` are not changed anymore. This ensures that in the end, the resulting form is not a macro invocation anymore, but represents either a literal value or a core language construct. The function `expand-all` initially calls `expand` on the passed macro environment and form. It then analyzes the form to determine whether any of the subforms of the resulting form require further macroexpansion, in case the resulting form is a list.

```
(define expand-once
  (lambda (env)
    (lambda (form)
      (let ((binding (cond ((symbol? form) (assoc form env))
                           ((pair? form) (assoc (car form) env)))))
        (cond
          ((and binding
             (or (and (symbol? form) (eq? (cadr binding) 'symbol-macro))
                 (and (pair? form) (eq? (cadr binding) 'list-macro))))
           ((caddr binding) form env))
          (else form))))))

(define expand
  (lambda (env) (letrec ((local-expand
                           (lambda (form)
                             (let ((new-form ((expand-once env) form)))
                               (cond ((eq? form new-form) form)
                                     (else (local-expand new-form)))))))
                  local-expand)))

(define bind-expander
  (lambda (env)
    (lambda (spec) (list (car spec) (cadr spec)
                         (eval ((expand-all env) (caddr spec)))))))

(define flatten-parameters
  (lambda (spec)
    (cond ((null? spec) '())
          ((symbol? spec) (list spec))
          (else (cons (car spec) (flatten-parameters (cdr spec)))))))

(define bind-variable
  (lambda (spec) (list spec 'variable (lambda (form _) form))))

(define expand-all
  (lambda (env)
    (letrec
      ((local-expand-all
        (lambda (form)
          (let ((form ((expand env) form)))
            (cond ((pair? form)
                   (case (car form)
                     ((quote) form)
                     ((begin if set!)
                      `(,(car form) ,@(map local-expand-all (cdr form))))
                     ((expander-let)
                      (let ((new (map (bind-expander env) (cadr form))))
                        `(begin ,@(map (expand-all (append new env))
                                       (cddr form)))))
                     ((lambda)
                      (let* ((params (flatten-parameters (cadr form)))
                             (new (map bind-variable params)))
                        `(lambda ,(cadr form)
                           ,@(map (expand-all (append new env))
                                  (cddr form)))))
                     (else (map local-expand-all form))))
                   (else form))))))
      local-expand-all)))
```

Figure 1: The core unhygienic macro system of Section 2.

There are five cases:

- If the form is a quoted form, it is returned unchanged.
- If it is a sequence, conditional or assignment (`begin`, `if`, or `set!`), the remaining elements are further expanded.[6]
- If the form is an `expander-let`, a new local macro environment is created in which the new expanders are bound (see below), and the subforms of the `expander-let` form are then expanded in that new macro environment.
- If the form is a `lambda`, the parameter list of the lambda form remains unchanged. However, a new local macro environment is created in which the parameters are bound as variables (see below) to ensure that they properly shadow potential macro definitions in the old macro environment. The subforms of the lambda form are then expanded in that new macro environment.
- Otherwise, the form is a function application. In that case, each element of the list that represents the function application is further expanded.

Except for the `expander-let` and `lambda` cases, all local macro expansions are performed with the same environment as initially passed to `expand-all`.

The function `bind-expander` is used for creating an entry in a macro environment. It is passed an environment of the macros that are considered to be in scope for the macro definition in question, and a specification describing that macro definition. It is either of the form `(name list-macro (lambda ...))` or `(name symbol-macro (lambda ...))`. This specification is converted by fully expanding the respective lambda form in the passed macro environment using `expand-all`, and then using `eval` to convert it into a function.[7]

The `expander-let` case in `expand-all` uses `bind-expander` for creating new local macro environments and creates a new sequence form (with `begin`) that contains the subforms from the `expander-let` form covered by the new macro definitions, fully expanded in the newly created macro environment.

The function `flatten-parameters` takes a parameter list, as accepted by Scheme lambda expressions, and turns it into a flat list of parameter names. The function `bind-variable` takes a variable name as a parameter, and creates an entry for a macro environment which states that the entry is of kind `variable`, and whose expansion function returns the passed form unchanged.[8]

[6] Unlike required by some Scheme specifications [Kelsey et al. 98, Sperber et al. 07], we avoid the special treatment of `begin` for 'splicing' internal definitions here without loss of generality.

[7] Since the macro expansion function does not see local (runtime) variables, it is sufficient to evaluate the lambda form in a predefined global environment. A more robust implementation of our macro system would perform additional checks to ensure that what is evaluated here is indeed a correctly shaped lambda form.

[8] By convention, we use the underscore in this paper as a variable name to indicate that we are actually not interested in the corresponding parameter.

This adds a third kind for macro environments, namely `variable`, alongside the already mentioned `list-macro` and `symbol-macro`, so that macro definitions can be properly shadowed by variable definitions. The `lambda` case in `expand-all` uses `flatten-parameters` and `bind-variable` for creating a new local macro environment and creates a new lambda form that contains the subforms from the original lambda form, fully expanded in the modified macro environment.

As already pointed out, the macro system presented in this section is still unhygienic. Here are some definitions for macros shown in this paper so far using this new macro system:

```
(expander-let
  ((push list-macro (lambda (form _)
                      (let ((obj-exp (cadr form))
                            (list-var (caddr form)))
                        `(set! ,list-var (cons ,obj-exp ,list-var)))))
   (or list-macro (lambda (form _)
                    (let ((exp-1 (cadr form))
                          (exp-2 (caddr form))
                          (temp (gensym)))
                      `(let ((,temp ,exp-1))
                         (if ,temp ,temp ,exp-2)))))
   (catch list-macro (lambda (form _)
                       (let ((body-exp (cadr form)))
                         `(call-with-current-continuation
                           (lambda (throw) ,body-exp)))))
   (name symbol-macro (lambda _ '(person-name this))))
  ...)
```

3 Bootstrapping Support for Macro Hygiene

To illustrate the essential idea of how to build a hygiene-compatible macro system on top of the unhygienic macro system presented in the previous section, recall the example for free symbol capture from the introduction in Section 1, now expressed in the macro system of the previous section:

```
(let ((x 42))
  (expander-let ((foo list-macro (lambda _ 'x)))
    (let ((x 4711))
      (foo))))
```

We can actually solve it by simply renaming one of the variables manually to make the code fragment evaluate to 42:

```
; (1) Manual renaming.
(let ((y 42))
  (expander-let ((foo list-macro (lambda _ 'y)))
    (let ((x 4711))
      (foo))))
```

This is indeed one of the proposed workarounds for avoiding free symbol capture in unhygienic macro systems. However, this is unsatisfactory because we would like to be able to choose names freely everywhere in the code. What we actually need is an operator, say `alias`, that gives us a reference to a variable in the current lexical scope that cannot be inadvertently captured:

```
; (2) Using aliases.
(let ((x 42))
  (expander-let ((foo list-macro (lambda _ (alias x))))
    (let ((x 4711))
      (foo))))
```

The core idea of the hygiene-compatible macro system introduced in this section is indeed that whenever a variable is introduced by a programmer, this actually leads to the introduction of two variables: One 'external' symbol macro that has the original name chosen by the programmer, and one 'internal' variable that has a unique name, as generated by `gensym`, that carries the actual variable binding. The external symbol macro is defined such that each reference to the original variable name in scope expands into a reference to the correct variable. Additionally, we can introduce the desired `alias` operator which yields internal names to unambiguously refer to correct variable bindings. Effectively, our hygiene-compatible macro system works by automating the renaming shown in code example (1).

In order to make our hygiene-compatible macro system work, we introduce new operators `aexpander-let`, `alambda` and `alet` in the next subsection, as 'alias-aware' variants of `expander-let`, `lambda` and `let`, as well as the `alias` operator itself. To make our example work in the hygiene-compatible macro system of this section, it has to use the alias-aware operators as replacements for their 'unhygienic' counterparts, as follows:[9]

```
; (3) Using alias-aware operators.
(alet ((x 42))
  (aexpander-let ((foo list-macro (alambda _ (alias x))))
    (alet ((x 4711))
      (foo))))
```

Code example (2) now expands into something as follows (occurrences of variable names with the prefix `%` indicate unique symbols, as generated by `gensym`):[10]

```
; (4) Expanded form of example (3).
(let ((%sym01 42))
  (expander-let ((x symbol-macro (lambda _ '%sym01)))
    (expander-let ((foo list-macro (lambda _ '%sym01)))
      (let ((%sym02 4711))
        (expander-let ((x symbol-macro (lambda _ '%sym02)))
          %sym01)))))
```

[9] We discuss in Sections 3.2 and 4 how to avoid having to use new names here.

[10] `expander-let` introductions are actually removed after they have been 'consumed' in `expand-all` (see Figure 1). However, for clarity we have left them in this example.

3.1 Generating Aliases

Figure 2 shows the additional definitions that are needed on top of the unhygienic macro system from the previous section to make that automatic renaming work. It defines three helper functions: The function `make-alias-formals` takes a parameter list as used in Scheme lambda forms and generates a congruent list, where each occurrence of a variable name is replaced by a unique symbol generated by `gensym`. The function `make-lambda-aliases` takes two such parameter lists, one with external variable names and one with corresponding internal names as created by `make-alias-formals`, and creates binding forms suitable for being embedded in an `expander-let` form. Those binding forms map external variable names to lambda forms that ignore their parameters and simply return the quoted internal variable names.

The third helper function `make-expander-aliases` takes a list of bindings as used in `expander-let` forms and creates new binding forms that can be embedded again in an `expander-let` form. It transforms the binding forms in such a way that each macro definition introduced by a binding has its (external) name replaced by a unique symbol generated by `gensym`, which thus becomes its internal name. On top of that, for each such macro definition, an additional macro is inserted that maps the external macro name to a lambda form that returns a corresponding macro invocation using the internal macro name. In case of symbol macros, that lambda form simply ignores its parameters and returns the quoted internal macro name, just like for variable names in `make-lambda-aliases`. In case of list macros, that lambda form returns a new list macro invocation, where the first entry is replaced with the internal macro name, but the remaining entries are left unchanged from the original macro invocation.

Using these helper functions, we can now define the new macros `alias`, `alambda` and `aexpander-let`. The `alias` macro yields quoted internal names for external names by simply performing one step of macro expansion on its parameter. The `alambda` macro expands into a `lambda` form where the parameter list is replaced by the result of passing it to `make-alias-formals`, and the body is wrapped by an `expander-let` mapping external to internal names. The `aexpander-let` macro expands into an `expander-let` form where the bindings are replaced by the result of passing them to `make-expander-aliases`.

We have to ensure that these new macros can themselves be protected against inadvertent capture by user-defined macro definitions, so we have to provide the separation into internal and external names manually.[11] In Figure 2, we use `%alias`, `%alambda` and `%aexpander-let` for illustration as placeholders for actual internal names as generated by `gensym`.

[11] Since we cannot use `aexpander-let` for this purpose yet because it is just being defined, we have to manually simulate its result.

```
(define make-alias-formals
  (lambda (spec)
    (cond ((null? spec) '())
          ((symbol? spec) (gensym))
          (else (cons (gensym) (make-alias-formals (cdr spec)))))))

(define make-lambda-aliases
  (lambda (spec alias-spec)
    (cond
      ((null? spec) '())
      ((symbol? spec) (list (list spec 'symbol-macro
                                  `(lambda _ (quote ,alias-spec)))))
      (else (cons (list (car spec) 'symbol-macro
                        `(lambda _ (quote ,(car alias-spec))))
                  (make-lambda-aliases (cdr spec) (cdr alias-spec)))))))

(define make-expander-aliases
  (lambda (bindings)
    (cond
      ((null? bindings) '())
      (else (let* ((binding (car bindings))
                   (spec (car binding))
                   (alias (gensym))
                   (kind (cadr binding))
                   (form (caddr binding)))
              (cond ((eq? kind 'symbol-macro)
                     (cons (list alias 'symbol-macro form)
                       (cons (list spec 'symbol-macro
                                   `(lambda _ (quote ,alias)))
                         (make-expander-aliases (cdr bindings)))))
                    ((eq? kind 'list-macro)
                     (cons (list alias 'list-macro form)
                       (cons (list spec 'list-macro
                                   `(lambda (form _)
                                      (cons (quote ,alias) (cdr form))))
                         (make-expander-aliases (cdr bindings))))))))))

(expander-let
  ((%alias        list-macro (lambda (form env)
                               `(quote ,((expand-once env) (cadr form)))))
   (%alambda      list-macro
                  (lambda (form _)
                    (let* ((formals
                            (make-alias-formals (cadr form)))
                           (new-env
                            (make-lambda-aliases (cadr form) formals)))
                      `(lambda ,formals
                         (expander-let ,new-env ,@(cddr form))))))
   (%aexpander-let list-macro
                   (lambda (form _)
                     `(expander-let ,(make-expander-aliases (cadr form))
                        ,@(cddr form))))
   (alias          list-macro (lambda (form _) `(%alias ,@(cdr form))))
   (alambda        list-macro (lambda (form _) `(%alambda ,@(cdr form))))
   (aexpander-let list-macro
                  (lambda (form _) `(%aexpander-let ,@(cdr form)))))

   ...)
```

Figure 2: The hygiene-compatible macro system of Section 3.

New hygiene-compatible binding forms can now be expressed in terms of these macros. As an example, `alet` is defined in terms of `alambda` below in the usual way, where the use of `alambda` is protected against inadvertent capture:

```
(aexpander-let
  ((alet list-macro
         (alambda (form env)
           `((,(alias alambda) ,(map car (cadr form)) ,@(cddr form))
             ,@(map cadr (cadr form)))))
  ...)
```

We can now also express our example by embedding it in these macro definitions. This version indeed evaluates to 42:

```
(alet ((x 42))
  (aexpander-let ((foo list-macro (alambda _ (alias x))))
    (alet ((x 4711))
      (foo))))
```

3.2 Global Definitions

The macro systems presented in the previous sections handle only local identifiers. In a more realistic scenario, we also want to be able to use global identifiers (as for example introduced with top-level `define`s in Scheme) and/or identifiers exported from modules in a module system. In case programs are deployed in their source form, and macro expansion can be performed at load time, ensuring uniqueness of internal names is still feasible for such global/exported identifiers. However, one goal of module systems is to enable separate compilation, which requires user-defined external names to be present in the compiled files such that they can be used for identifying definitions at link or load time.

Module systems themselves suffer from potential name clashes when two independently developed modules happen to have the same names and export the same identifiers. The only way to completely avoid such accidental name clashes is by ensuring that module names and exported identifiers are globally unique, as is the case for example in Java, where internet domain names are used to ensure uniqueness [Gosling et al. 05], or in Microsoft COM, where globally unique identifiers (GUIDs) are used for similar purposes [Box 98]. An extension of our hygiene-compatible macro system where global/exported identifiers map to such globally unique names instead of internal names generated by `gensym` is relatively straightforward, and is discussed in more detail in Section 4 below.

A drawback of our approach is that the presented unhygienic and hygiene-compatible macro systems are incompatible: If the `lambda` and `expander-let` operators (or any derived binding forms) from the language recognized by the core unhygienic macro system are used in the same code where `alambda`, `alet` and especially the `alias` operator are used, the necessary mappings from external to internal names are not created anymore, and existing mappings are shadowed due to the handling of `lambda` and `expander-let` in the core unhygienic macro system. Consider the following example, under the assumption that `let` is expressed in terms of the core `lambda` operator in the usual way:

```
(let ((x 42))
  (aexpander-let
    ((foo list-macro (lambda _ (alias x))))
    (alet ((x 4711))
      (foo))))
```

Contrary to what one might expect, this form evaluates to 4711, because the introduction of the outer binding for `x` does not lead to a mapping from a symbol macro `x` to some internal symbol for the actual binding, so the invocation of `(alias x)` cannot detect such an internal mapping. Instead, the invocation of `(foo)` sees the inner definition of `x`. The fact that the inner definition is turned into a symbol macro, due to being introduced by an `alet`, does not have an effect here, `(foo)` expands into `x` and uses whatever `x` is locally defined.

However, in a realistic setting, it is desirable not to have to use new names for standard binding forms anyway (like `alambda`, `alet`, etc.), but rather to keep the original names (`lambda`, `let`, etc.), but with the added functionality described in the previous subsection. This also makes it easier to port existing code to the new hygiene-compatible macro system. Fortunately, advanced module and package systems, like provided by PLT Scheme [Flatt 07] and Common Lisp [ANSI 94], allow renaming and/or shadowing of imported identifiers in such a way that keeping the original names is also relatively straightforward to achieve. This also enables preventing the core unhygienic language from being used in conjunction with the hygiene-compatible language to avoid the problems discussed above. See Section 4 for a discussion how we achieve this in Common Lisp.

3.3 Additional High-Level Operations

Although the alias operators introduced above are sufficient for avoiding inadvertent free symbol capture, we can introduce additional high-level operators (see Figure 3), similar to the ones commonly available in traditional advanced hygienic and hygiene-compatible macro systems, like syntax-case [Dybvig et al. 92] and syntactic closures [Bawden and Rees 88].

With `free-symbol-identifier?`, we can determine whether a symbol names a bound or free identifier. We simply check whether that symbol expands to itself (`expand-once` has not found an expansion, or only an identity expansion as introduced by `bind-variable`) or into something else (it is directly or indirectly introduced by `alambda`, or by `aexpander-let` as a symbol macro).

With `rebound-symbol-identifier?`, we can determine whether a variable or a symbol macro has been rebound in an inner lexical scope. For that, we need an operator `current-env` to yield the current macro environment.

```
(aexpander-let
  ((free-identifier? list-macro
    (alambda (form env)
      (and (symbol? (cadr form))
           (alet ((expanded ((expand env) (cadr form))))
             (eq? (cadr form) expanded))))))
  (alet ((x 42))
    (display "x free? ")
    (display (free-identifier? x))
    (newline)
    (display "y free? ")
    (display (free-identifier? y))
    (newline)))

; displays:
; x free? #f
; y free? #t

(aexpander-let
  ((current-env list-macro
    (alambda (_ env) `(quote ,env)))
  (alet ((x 1) (y 2))
    (aexpander-let
      ((rebound-identifier? list-macro
        (alambda (form env)
          (and (symbol? (cadr form))
               (not (eq? ((expand env) (cadr form))
                         ((expand (current-env)) (cadr form))))))))
      (alet ((x 3))
        (display "x rebound? ")
        (display (rebound-identifier? x))
        (newline)
        (display "y rebound? ")
        (display (rebound-identifier? y))
        (newline)))))

; displays:
; x rebound? #t
; y rebound? #f

(aexpander-let
  ((or list-macro
    (alambda (form env)
      (alet ((exp-1 (cadr form))
             (exp-2 (caddr form)))
        `(,(alias alet) ((temp (expander-let ,(reverse env) ,exp-1)))
            (,(alias aif) temp temp
               (expander-let ,(reverse env) ,exp-2)))))))
  (alet ((temp 42))
    (or #f temp)))
```

Figure 3: Additional high-level operators on top of the macro system of Section 3.

Since we have access to all expanders which are in scope where a macro is invoked, we can also easily protect subforms from being captured. For example, we can define the `or` macro from Section 1 as in Figure 3.[12] This is similar to the use of syntactic closures, where forms can be closed over syntactic environments in a similar way [Bawden and Rees 88, Hanson 91]. Compare this to a version of the `or` macro that uses `gensym` for protecting against macro argument capture:

```
(aexpander-let
   ((or list-macro (alambda (form _)
                     (alet ((exp-1 (cadr form))
                            (exp-2 (caddr form))
                            (temp (gensym)))
                       `(,(alias alet) ((,temp ,exp-1))
                          (,(alias aif) ,temp ,temp ,exp-2))))))
   (alet ((temp 42))
     (or #f temp)))
```

3.4 Discussion

The required elements listed in Section 2.1 that are provided in the unhygienic macro system of the previous section are used to build the hygiene-compatible macro system in this section as follows:

- Symbol macros are used to map from external to internal names.
- Macro environments and low-level macro expansion functions enable `alias` to look up internal names for the respective scopes.
- Local macros are affected (expanded) by outer macros. This allows defining `alias` as well as the high-level operators in the previous subsection as higher-order macros to be expanded at compile time.

The hygiene-compatible macro system presented in this section is fully layered on top of the unhygienic macro system in the previous section. Especially, there is no need for walking the code embedded in a macro definition to ensure that external names are correctly mapped to internal ones. Consequently, the hygiene-compatible macro system does not need any knowledge about the core language that is processed by the core unhygienic macro system, but relies on the fact that the core macro system already correctly distinguishes core language constructs from macros. In contrast, traditional algorithms for supporting macro hygiene have to explicitly walk code embedded in macro definitions [Rees 93]. As a consequence, they have to be aware of the core constructs of the underlying language, so they have to be intimately tied to the compiler of the core language.

[12] The expanders can be reinstated by way of `expander-let` instead of `aexpander-let` here, since no new aliases need to be created.

Furthermore, our hygiene-compatible macro system does not require any additional data structures for representing identifiers and recording their syntactic levels, as is typically done in traditional hygienic and hygiene-compatible macro systems [Rees 93]. Instead, we exclusively use plain symbols for representing identifiers, which effectively leads to a 'flattening' of all identifiers in the result of `expand-all`, independent of whether an identifier is introduced by the programmer or by a macro, and independent of the stage at which an identifier is introduced. This is a consequence of the automation of the renaming solution for avoiding inadvertent free symbol capture that our approach is based on. This also means that we can still use plain unaliased symbols if we want to express macros that intentionally capture variable names.

4 Integration into Common Lisp

As a proof of concept, we have implemented a full version of a hygiene-compatible macro system in Common Lisp following the approach in Section 3.[13] To achieve this, all binding forms (`defvar`, `defun` `defmacro`, `let`, `let*`, `flet`, `macrolet`, and so on) have to be reimplemented in a way similar to `alambda` and `alet`, so that they can generate the necessary mappings from external to internal names. The 'internal' names for global definitions cannot be uninterned symbols, so they are symbols with the name of their respective package prepended and interned in a dedicated package: As long as that package is not manipulated by user code, it thus guarantees uniqueness for global names. To keep things manageable, we have not reimplemented all of Common Lisp, but restricted ourselves to ISLISP, which is mostly a small but non-trivial subset of Common Lisp [ISO 97].

On the one hand, this implementation is feasible since Common Lisp provides all of the required elements listed in Section 2, including 'list' macros and symbol macros, local macros affected by surrounding macros, and macro expansion functions which operate on local macro environments. Although macros are specified differently from the `expander-let` forms used in this paper (using `macrolet`), it is still also possible to access the local macro environment as part of the macro's parameter list via the `&environment` keyword.

On the other hand, we are faced with two additional technical challenges: Whereas Scheme uses a single namespace for values, Common Lisp and ISLISP provide different namespaces for variables, functions, block names, and so on. This requires different alias operators for the different namespaces that can be locally rebound, that is, `alias`, `function-alias`, and `block-alias`.[14] However, apart from minor differences, the approach for mapping external to internal names is always the same, like for `alias`.

[13] Download available at http://p-cos.net/core-lisp.html
[14] For example, classes and go tags cannot be locally rebound.

Secondly, although providing access to local macro environments, ANSI Common Lisp does not provide any operators for accessing their entries. However, it provides `macroexpand-1` and `macroexpand` (as equivalents to `expand-once` and `expand`) that take such macro environments as parameters. In order to provide mappings from external to internal names, we have to rebuild the macro environments as discussed in this paper on top of these low-level mechanisms.

In spite of these technical challenges, we have been able to preserve the characteristics of the hygiene-compatible macro system presented in this paper. Especially, it is a mere layer on top of Common Lisp's unhygienic macros, does not require a code walker and has a fully portable implementation, as confirmed by tests on several Common Lisp implementations. Additionally, Common Lisp's package system allows the reuse of the same names for binding forms as the original ones provided by Common Lisp, by defining our own package, shadowing the original binding forms, and reimplementing them as described above.

This esentially means that we have built our own Lisp dialect on top of Common Lisp (*HCL* for *Hygiene-compatible Lisp*). A question that arises is whether and how HCL can use Common Lisp libraries and vice versa in a safe way. To answer this question constructively, we make the following assumptions: ANSI Common Lisp specifies that redefining or lexically rebinding symbols exported from the `COMMON-LISP` package has undefined consequences (Section 11.1.2.1.2 in [ANSI 94]). We assume that Common Lisp libraries therefore indeed do not redefine or lexically rebind such symbols, which ensures that macros specified in ANSI Common Lisp, and therefore exported from the `COMMON-LISP` package, always see the correct bindings of predefined variables and functions. We furthermore assume that Common Lisp libraries do not redefine or lexically rebind symbols from any other packages either. In other words, we assume that programmers of Common Lisp libraries have indeed used the known workarounds and measures to protect their macro definitions from inadvertent capture.

In such a case, exporting definitions from packages implemented in HCL and importing them into Common Lisp code does not pose any problems: Symbols from HCL will not be redefined or rebound in Common Lisp code, so they will not be replaced with bindings without the necessary mappings from external to internal names that are necessary for HCL's aliasing operators to work correctly.

Definitions exported from Common Lisp and imported into HCL pose a more serious challenge: A HCL programmer expects to be able to use aliasing to protect against free symbol capture, but aliasing does not work on symbols imported from Common Lisp libraries, because they do not provide the necessary mappings from external to internal names. The solution is that HCL packages *never* import symbols from Common Lisp. Instead, we provide operators for importing *definitions* from Common Lisp packages, which define new symbols in HCL packages that map to original symbols in Common Lisp packages.

Consider the following example:

```
(import-variable pi common-lisp:pi)
```

This example expands into the following code:

```
(progn (define-symbol-macro pi common-lisp:pi)
  ...)
```

The omitted code contains the necessary actions to ensure that macro environments 'know' that `common-lisp:pi` is the 'internal' name for `pi`. HCL provides similar operators for importing functions, symbol macros and macros.

Another important case are Common Lisp macros that create new local bindings for some code body. For example, Common Lisp's `defmethod` macro creates the local function `call-next-method` whose name is a symbol exported from the `COMMON-LISP` package. To ensure that local HCL macros can alias such bindings introduced locally by Common Lisp macros, such symbols should not be imported from Common Lisp packages either. Instead, HCL provides operators `with-imported-variables`, `with-imported-functions`, and so on, to provide local mappings from HCL names to Common Lisp names, which are again considered 'internal' names for the purpose of the HCL macro system. So in the following example, `call-next-method` has a corresponding local mapping:

```
(defmethod foo ((x integer) (y integer) (z integer))
  (with-imported-functions
    ((call-next-method common-lisp:call-next-method))
    ...))
```

These import operators for both global and local definitions from Common Lisp libraries cover the most important cases when interoperating between HCL and Common Lisp. One case that is still not covered are Common Lisp macros that compute new names for automatically generated bindings (like the various functions generated by the `defstruct` macro). HCL does not provide a straightforward solution here. Instead, more effort is necessary in a separate library to define wrappers for such macros that ensure that the new names are interned in an external package, and then imported with the operators discussed above.

Another special case are keywords exported from Common Lisp's *keyword* package that are specified to evaluate to themselves: If used in HCL code, they retain their special status and should not be redefined or rebound. HCL's `nil` is even more special in that it loses its equivalence to `'nil` inside HCL code.

5 History and Related Work

Since the introduction of macros into Lisp in 1963 [Hart 63], macro systems have been continuously improved in various Lisp dialects. Pitman gives a summary of the then state of the art in an overview paper in 1980 [Pitman 80], shortly before the initial specification of Common Lisp was commenced.

Based on the good experiences with lexical scoping in Scheme, which set that language apart from other Lisp dialects that were typically dynamically scoped by default, one goal for Common Lisp was to define equally powerful lexically scoped constructs. Due to Common Lisp's different namespaces for different kinds of values, it was necessary to provide multiple constructs for lexically scoped definitions as well. For example, Common Lisp defines `flet` and `labels` for functions, `let` and `let*` for variables, `block` for blocks, and so on. One of the additions was a lexically scoped `macrolet` which, to the best of our knowledge, did not exist in previous Lisp dialects. The introduction of `macrolet` required a representation of local macro environments, as well as a change to `macroexpand` to accept such environments as an additional parameter. The function `macroexpand` itself already existed in previous Lisp dialects [Pitman 80]. According to the Common Lisp email history [White 84], the discussion about the modification of `macroexpand` started around August 1982.

Symbol macros, on the other hand, were introduced much later, as part of the Common Lisp Object System: It was considered desirable to enable access to fields in objects in a way similar to that of other object-oriented languages, without the need to mention the (implicit) `this` or `self` reference. In order to generalize this idea, `symbol-macrolet` was introduced in 1988 as part of the CLOS specification [Bobrow et al. 89].

Kohlbecker's seminal work started the research on macro hygiene in 1986 [Kohlbecker et al. 86] - after the introduction of `macrolet` and macro environments in Common Lisp, but before `symbol-macrolet`. Although hygienic macro systems were proposed for Common Lisp, they were not adopted, so research on macro hygiene continued almost exclusively in Scheme. Housel gives an overview of hygienic macro expansion in a series of usenet postings [Housel 93].

The algorithm in [Clinger and Rees 91] is a refinement of Kohlbecker's work. That algorithm performs hygienic macro expansion by way of renaming identifiers when they are newly introduced in macro definitions. This covers both identifiers used for new local bindings as well as new free identifiers that are supposed to refer to bindings in the lexical scope of the macro definition. A low-level, hygiene-compatible macro facility is described in [Clinger 91a], and was the basis for an implementation of the hygienic macro system described in [Clinger and Rees 91]. The hygiene-compatible macro system presented in this paper is very similar to the one described in [Clinger 91a]: In our system, 'external' potentially ambiguous identifiers can be turned into 'internal' unique ones by way of `alias`, whereas in their system, `rename` is used for the same purpose. However, in their system, the mapping from potentially ambiguous to guaranteed unique names is not created when bindings are introduced (like with `alambda` or derived forms in our system), but is generated by `rename` after the fact as soon as macros introduce new identifiers as part of macroexpansion. Macroexpanded code is further processed in a special lexical environment that maps the renamed identifiers back to the bindings of the original identifiers in the respective lexical environments of the macro definitions. The algorithm described in [Clinger and Rees 91] recognizes core language constructs of the underlying language in order to work correctly, and thus must be integrated with the compiler or interpreter for that language.

The fact that `symbol-macrolet` did not exist at the time when research on macro hygiene started may be a reason why the potential of using it for resolving macro hygiene issues was not recognized. Symbol macros have been explored in the context of Scheme much later, by Waddell in 1999 [Waddell and Dybvig 99], there called identifier macros, and have been adopted as part of R6RS Scheme only relatively recently [Sperber et al. 07]. The fact that symbol macros can be used in conjunction with local macro environments to bootstrap a hygiene-compatible macro system is the major discovery of this paper.

6 Conclusions and Future Work

Macro argument capture and free symbol capture are sometimes compared to the issues of dynamic scoping. In traditional Lisp dialects where variables are dynamically scoped by default, a new variable binding may inadvertently capture another one with the same name that is needed by a function to be evaluated further down the call chain. Lexical scoping is essential to ensure that closures can close over the variables visible at their definition sites. It is probably impossible to resolve such nameclashes otherwise without manually reimplementing lexical scoping. This paper shows that the case for macro hygiene is a different one, by constructing the essential ingredients of a hygiene-compatible macro system as a mere layer on top of an advanced unhygienic one. The difference is due to the fact that the different syntactic scopes are not needed in the fully macroexpanded code, but that macroexpansion can 'flatten' all identifiers, while separate lexical environments need to be maintained for closures at runtime.

It has been shown that hygienic macro systems can be implemented on top of syntactic closures [Hanson 91], but it remains future work to show that we can do this on top of our hygiene-compatible macro system as well. The hygiene-compatible macro system presented in this paper works because macros can expand into definitions of local symbol macros, and in this way control the further expansion of embedded code fragments. Macros implemented in expansion-passing style provide a different approach for controlling such further expansions [Dybvig et al. 88], and it would be interesting to see whether the approach presented here can be reimplemented using expansion-passing style.

References

[ANSI 94] ANSI/INCITS X3.226-1994. *American National Standard for Information Systems - Programming Language - Common Lisp*, 1994.

[Bawden 99] Alan Bawden, Quasiquotation in Lisp, *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM99)*, Technical report BRICS-NS-99-1, University of Aarhus, 1999.

[Bawden and Rees 88] Alan Bawden and Jonathan Rees, Syntactic Closures. *Conference on Lisp and Functional Programming*, July 1988, ACM Press.

[Bobrow et al. 89] Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, and David Moon, The Common Lisp Object System Specification. *Lisp and Symbolic Computation*, Vol. 1, No. 3-4, January 1989, Springer Verlag.

[Box 98] Don Box, *Essential COM*. Addison-Wesley, 1998.

[Clinger and Rees 91] William Clinger, Jonathan Rees, Macros That Work. POPL'91, ACM Press.

[Clinger 91a] William Clinger, Hygienic macros through explicit renaming. *Lisp Pointers* IV(4), December 1991, ACM Press.

[Clinger 91b] William Clinger, Macros in Scheme. *Lisp Pointers* IV(4), December 1991.

[Dybvig et al. 88] R. Kent Dybvig, Daniel Friedman, and Christopher Haynes, Expansion passing style: A general macro mechanism. *Lisp and Symbolic Computation*, Vol. 1, No. 1, June 1988, Springer Verlag.

[Dybvig et al. 92] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman, Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4), 1992, Springer Verlag.

[Felleisen 91] Matthias Felleisen, On the Expressive Power of Programming Languages. *Science of Computer Programming*, Vol. 17, No. 1-3, December 1991.

[Flatt 07] Matthew Flatt, PLT MzSchheme: Language Manual 370, May 2007.

[Gabriel and Pitman 88] Richard P. Gabriel and Kent M. Pitman, Technical Issues of Separation in Function Cells and Values Cells. *Lisp and Symbolic Computation*, Vol. 1, No. 1, June 1988, Springer Verlag.

[Gosling et al. 05] James Gosling, Bill Joy, Guy Steele Jr., and Gilad Bracha, *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[Graham 93] Paul Graham, *On Lisp*. Prentice-Hall, 1993.

[Hanson 91] Chris Hanson, A Syntactic Closures Macro Facility. *Lisp Pointers* IV(4), 9-16, December 1991, ACM Press.

[Hart 63] Timothy Hart, MACRO Definitions for LISP. AI Memo 57, MIT, 1963.

[Housel 93] Peter Housel, An introduction to macro expansion algorithms, parts 1-4. `http://www.cs.indiana.edu/pub/scheme-repository/doc/misc/`.

[ISO 97] ISO/IEC 13816:1997. *Programming Language ISLISP*, 1997.

[Kay 69] Alan Kay, *The Reactive Engine*, PhD thesis, University of Utah, 1969.

[Kelsey et al. 98] Richard Kelsey, William Clinger, Jonathan Rees (eds.). *Revised^5 Report on the Algorithmic Language Scheme*. Higher-Order and Symbolic Computation, Vol. 11, No. 1, September 1998, Springer Verlag.

[Kohlbecker et al. 86] Eugene Kohlbecker, Daniel Friedman, Matthias Felleisen, and Bruce Duba, Hygienic macro expansion. *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, ACM Press.

[McIlroy 60] M. Douglas McIlroy, Macro instruction extensions of compiler languages. *Communications of the ACM*, Vol. 3, No. 4, April 1960.

[Pitman 80] Kent Pitman, Special Forms in Lisp. *LISP Conference 1980*, ACM Press.

[Rees 93] Jonathan Rees, Implementing lexically scoped macros. *Lisp Pointers*, 1993.

[Sperber et al. 07] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). *Revised^6 Report on the Algorithmic Language Scheme*, 2007.

[Waddell and Dybvig 99] Oscar Waddell and R. Kent Dybvig, Extending the Scope of Syntactic Abstraction. POPL'99, ACM Press.

[White 84] JonL White, History of Common Lisp, `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/doc/history/cl.txt`.