, , 1–28 ()

Quasiquotation in Lisp

ALAN BAWDEN *
Brandeis University
bawden@cs.brandeis.edu

Abstract. Quasiquotation is the technology commonly used in Lisp to write program-generating programs. This paper explains how quasiquotation works, why it works well, and what its limitations are. A brief history of quasiquotation is included.

Keywords: Quasiquote, Backquote, Unquote, Comma, Lisp, Scheme

1. Introduction

Quasiquotation is a parameterized version of ordinary quotation, where instead of specifying a value *exactly*, some holes are left to be filled in later. A quasiquotation is a "template." Quasiquotation appears in various Lisp dialects, including Scheme [9] and Common Lisp [18], where it is used to write syntactic extensions ("macros") and other program-generating programs.

A typical use of quasiquotation in a macro definition looks like:

```
(define-macro (push expr var)
  `(set! ,var (cons ,expr ,var)))
```

The body of this `define-macro` uses quasiquote to construct a `set!`-expression. The backquote character (`` ` ``) introduces a quasiquotation, just as the ordinary quote character (`'`) introduces an ordinary quotation. Inside the quasiquotation, the comma character (`,`) marks expressions whose values are to be substituted into the result. This is almost everything a Lisp programmer needs to know in order to use quasiquotation—the basic idea is very simple.

But a simple idea can be the start of a complicated story. This paper introduces quasiquote in its simplicity, and then explores the following aspects of its story:

- It is powerful. The combination of quasiquotation with Lisp's S-expression data structures yields a remarkably effective technology for program construction and manipulation.

- It has complicated ramifications. In particular, when a quasiquotation builds another quasiquotation (i.e., when quasiquotes are "nested") the notation becomes rather inscrutable.

- It has an interesting history. Despite the power to be gained by combining quasiquotation with S-expressions, it took almost twenty years for quasiquotation to become an accepted and well-understood part of most Lisp dialects.

* This work was supported by NSF grant EIA-9806718.

- It has some limitations. Program construction usually involves building expressions that are subject to variable scoping rules. Lisp's quasiquote is blind to these rules.

Section 2 motivates and introduces basic quasiquotation as it is used in Lisp. Section 3 explains the more advanced features of quasiquote and how they interact with each other. Section 4 demonstrates how quasiquotation can be used to help turn an interpreter into a compiler. Section 5 discusses quasiquote's blindness to variable scoping and surveys the techniques used to cope with this limitation. Section 6 gives a brief history of quasiquotation. Section 7 shows how quasiquote can be used to solve a famous puzzle. And finally, section 8 summarizes our conclusions.

In this paper, the word "Lisp" means primarily the Lisp dialects Common Lisp and Scheme. Examples of Lisp code will be written in Scheme, although a few examples will not work in most Scheme implementations (for reasons revealed in appendix B).

2. Why Quasiquotation?

Before looking at how quasiquotation is used to write program-generating programs in Lisp, let us consider what happens when a program-generating program must be written in C. Seeing what can and cannot be easily accomplished in C will help clarify what a truly useful and well-integrated quasiquotation technology should look like.

2.1. Quasiquotation in C

The most straightforward way to write a C program that generates another C program is to textually construct the output program via string manipulation. The program will probably contain many statements that look like:

```
fprintf(out, "for (i = 0; i < %s; i++)
                %s[i] = %s;\n",
        array_size,
        array_name,
        init_val);
```

C's `fprintf` procedure is a convenient way to generate the desired C code. Without `fprintf`, the programmer would have to write:

```
fputs("for (i = 0; i < ", out);
fputs(array_size, out);
fputs("; i++) ", out);
fputs(array_name, out);
fputs("[i] = ", out);
fputs(init_val, out);
fputs(";\n", out);
```

It is clear that the `fprintf` statement generates a syntactically legal C statement. Determining that the sequence of calls to `fputs` generates legal C requires that it be read carefully.

Using `fprintf` achieves the central goal of quasiquotation: It allows the programmer to write expressions that look as much like the desired output as possible. He can write down what he wants the output to look like, and then modify it only slightly in order to parameterize it with escape sequences such as "`%s`".

Although `fprintf` makes it *easier* for a programmer to write a C program that generates a C program, two problems with this technology will soon become clear to anyone who has to use it for very long:

- The parameters are associated with their values positionally. One must count arguments and occurrences of "`%s`" to figure out which matches up with which. If there are a large number of parameters, errors can easily occur.

- The string substitution that underlies this technology has *no* understanding of the syntactic structure of the programming language being generated. As a result, unusual values for any of the parameters can change the meaning of the resulting code fragment in unexpected ways. (Consider what would happen if `array_name` were "`*x`": C's operator precedence rules cause the resulting code to be parsed as "`*(x[i])`" rather than the presumably intended "`(*x)[i]`".)

The first problem could be addressed by somehow moving the parameter expressions into the template—something like:

```
subst("for (i = 0; i < $array_size; i++)
        $array_name[i] = $init_val;");
```

But even if this could be made to work in C, the second problem would remain: flat character strings are not a good way to represent recursive structures such as expressions. Indeed, programmers typically adopt some convention for inserting extra parentheses into such strings to ensure that they parse as intended. (This technique is often employed by users of the C preprocessor for the very same reason.)

This analysis suggests the following three goals for a successful implementation of quasiquotation:

- Quasiquotation should enable programmers to write down what they want the output to look like, modified only slightly in order to parameterize it.

- The parameter expressions should appear inside the template, in the positions where their values will be inserted.

- The underlying data structures manipulated by quasiquotation should be rich enough to represent recursively-defined structures such as expressions.

The achievement of this last goal is where Lisp really shines.

2.2. Quasiquotation in Lisp

Now consider the case of writing a Lisp program that generates another Lisp program. It would be highly unnatural for a Lisp program to accomplish such a task by working with character strings, as the C code in the previous section did, or even to work with tokens, as the C preprocessor does. The natural way for a Lisp program to generate Lisp code is to work with Lisp's "S-expression" data structures: lists, symbols, numbers, etc. So suppose the goal is to generate a Lisp expression such as:

```
(do ((i 0 (+ 1 i)))
    ((>= i array-size))
  (vector-set! array-name i init-value))
```

The primitive Lisp code to construct such an S-expression is:

```
(list 'do '((i 0 (+ 1 i)))
      (list (list '>= 'i array-size))
      (list 'vector-set! array-name 'i init-val))
```

It is an open question whether this code is more or less readable than the C code in the previous section that used repeated calls to `fputs` instead of calling `fprintf`. But Lisp's quasiquotation facility lets one write instead:

```
`(do ((i 0 (+ 1 i)))
     ((>= i ,array-size))
   (vector-set! ,array-name i ,init-val))
```

A backquote character (`` ` ``) precedes the entire template, and a comma character (`,`) precedes each parameter expression inside the template. (The comma is sometimes described as meaning "unquote" because it turns off the quotation that backquote turns on.)

It is clear what this backquote notation is trying to *express*, but how does a Lisp implementation actually make this *work*? The underlying technology for C's `fprintf` is an interpreter for a simple output formatting language. What is the underlying technology for this backquote notation?

The answer is that the two expressions above are actually *identical* S-expressions! That is, they are identical in the same sense that

```
(A B . (C D E . ()))
```

and

```
(A B C D E)
```

are identical. Lisp's S-expression parser (traditionally called "`read`") expands a backquote followed by a template into Lisp code that constructs the desired S-expression. So we can write:

```
`(let ((,x ',v)) ,body)
```

and it will be exactly as if we had written:

```
(list 'let
      (list (list x (list 'quote v)))
      body)
```

Backquote expressions are just a handy notation for writing complicated combinations of calls to list constructors. The exact expansion of a backquote expression is not specified—`read` is allowed to build any code that constructs the desired result.[1] (One possible expansion algorithm is described in appendix A.) So the backquote notation does not change the fact that a program-generating Lisp program works by manipulating Lisp's list structures.

Clearly this backquote notation achieves at least the first two of the three goals for quasiquotation: the code closely resembles the desired output and the parameter expressions appear directly where their values will be inserted. The third goal for a quasiquotation technology was that the underlying data structures it manipulates should be appropriate for working with programming-language expressions. It is not immediately clear that we have achieved that goal.

List structure is not quite as stark a representation as character strings, but it is still pretty low-level. Perhaps we would be happier if, instead of manipulating lists, our quasiquotation technology manipulated objects from a set of abstract data types that were designed specifically for each of the various different syntactic constructs in our language (variables, expressions, definitions, `cond`-clauses, etc.). After abandoning character strings as too low-level, it seems very natural to keep moving towards even higher-level representations that capture even more of the features of the given domain.

We might, for example, design a set of abstract data types for various programming-language constructs—something like the "abstract syntax" often employed when analyzing programming languages. Such a representation might prevent us from accidentally using quasiquotation to construct programs with illegal syntax. Or perhaps it would protect us from unintended variable captures (a problem discussed below in section 5). This additional safety might make the additional complexity worthwhile. Or perhaps a higher-level representation would enable us to do more powerful operations on programming-language fragments beyond simply plugging them into quasiquotation templates. This additional functionality might also offset the increased complexity.

Despite these possibilities, no superior representation has appeared during the last twenty years. Although recently, some clouds have started to gather on the horizon: The variable capture problems discussed below in section 5 have received a lot of attention. An improved quasiquotation may someday emerge from the research in this area.

But for now, there do not seem to be any compelling reasons to complicate matters by moving up to an even higher-level representation. S-expression-based quasiquotation achieves our third goal of using an appropriate representation for programming-language expressions.

2.3. Synergy

In fact, there is a wonderful synergy between quasiquotation and S-expressions. They work together to yield a technology that is more powerful than the sum of the two ideas taken separately.

As we saw in the last section, Lisp code that constructs non-trivial S-expressions by directly calling Lisp's list constructing procedures tends to be extremely unreadable. The most experienced Lisp programmers will have trouble decoding the following expression:

```
(cons 'cond
      (cons (list (list 'eq? var (list 'quote val))
                  expr)
            more-clauses))
```

But even a novice can see what the equivalent quasiquotation does:

```
`(cond ((eq? ,var ',val) ,expr)
       . ,more-clauses)
```

(The appearance of "dot notation" in this example points towards an inadequacy in quasiquote as presented thus far. We will return to this example in section 3.1.)

S-expressions were at the core of McCarthy's original version of Lisp [13]. The ability to manipulate programs as data has always been an important part of what Lisp is all about. But without quasiquotation, actually working with S-expressions can be painful. Quasiquotation corrects an important inadequacy in Lisp's original S-expression toolkit.

The benefits flow the other way as well. As we have seen, character string based quasiquotation is an untidy way to represent recursive data structures such as expressions. But if our data is represented using S-expressions, substitution in quasiquotation templates works cleanly. So S-expressions correct an inadequacy in string-based quasiquotation.

Quasiquotation and S-expressions compensate for each other's weaknesses. Together they form a remarkably effective and flexible technology for manipulating and generating programs. So it is not surprising that although quasiquotation did not become an official feature of any Lisp dialect until twenty years after the invention of Lisp, it was in common use by Lisp programmers for many years before.

This close relationship between quasiquotation and S-expressions has led some programmers to adopt a style where quasiquote is *only* used for generating programs. In this style, lists that are used as data are always constructed by explicitly calling list structure primitives such as `list` and `cons`, while lists that are used to represent programs (i.e., S-expressions) are always constructed using quasiquote. Because data lists tend to be uncomplicated, very little is lost by refraining from using quasiquote to construct them. In return for this small sacrifice, a large gain in readability results from clearly separating the construction of data lists from the construction of S-expressions. In this paper, we will adopt this style.

3. Embellishments

Having presented the case that quasiquotation in Lisp is a powerful and useful technology, we proceed to fill in the rest of the picture. Two important points about quasiquotation and how it is used will be presented. First, we introduce an additional feature, called "splicing." Second, we take a look at what happens when quasiquotations are nested.

3.1. Splicing

We brushed very close to needing splicing in section 2.3. Recall:

```
`(cond ((eq? ,var ',val) ,expr)
       . ,more-clauses)
```

The value of the variable `more-clauses` is presumably a list of additional `cond`-clauses to be built into the `cond`-expression we are constructing. Suppose we know (for some reason) that that list does not include an `else`-clause, and we want to supply one. We can always write:

```
`(cond ((eq? ,var ',val) ,expr)
       . ,(append more-clauses
                  '((else #T))))
```

But calls to things like `append` are exactly what quasiquotation is supposed to help us avoid.

The backquote notation seems to suggest that we should be able to write instead:

```
`(cond ((eq? ,var ',val) ,expr)
       . ,more-clauses
       (else #T))
```

Unfortunately, this abuse of Lisp's "dot notation" will be rejected by the Lisp parser, `read`. Fortunately, this is a common enough thing to want to do that the backquote notation allows us to achieve our goal by writing:

```
`(cond ((eq? ,var ',val) ,expr)
       ,@more-clauses
       (else #T))
```

This new two-character prefix, comma-atsign (`,@`), is similar to the plain comma prefix, except the following expression should return a *list* of values to be "spliced" into the containing list. (Given the example, the reader may well wonder why the prefix comma-period (`,.`) was not chosen instead. Section 6 answers this question.)

The expanded code might read:

```
(cons 'cond
      (cons (list (list 'eq? var (list 'quote val))
```

```
                 expr)
            (append more-clauses
                    '((else #T)))))
```

This is certainly not very readable, but a simple example should make everything clear: If the value of `X` is `(1 2 3)`, then the value of

```
`(normal= ,X splicing= ,@X see?)
```

is

```
(normal= (1 2 3) splicing= 1 2 3 see?)
```

Splicing comes in handy in many situations. A look at the BNF for any Lisp dialect will reveal many kinds of expressions where a sequence of sub-parts occur: the arguments in a function call, the variables in a `lambda`-expression, the clauses in a `cond`-expression, the variable binding pairs in a `let`-expression, etc. When generating code that uses any of these kinds of expressions, splicing may prove useful.

The distinction between ordinary substitution (`,`) and splicing substitution (`,@`) exists because S-expressions are trees and not flat sequences. Character string based quasiquotation has no way to make this distinction—the user is forced to make the distinction in other ways, perhaps by explicitly inserting parentheses into the string.

3.2. Nesting

Sometimes a program-generating program actually generates another program-generating program. In this situation, it will often be the case that a quasiquotation will be used to construct another quasiquotation. Quasiquotations will be nested.

Nested quasiquotation sounds like the kind of highly esoteric construction that would only be needed by the most wizardly compiler-writers, but in point of fact, even fairly ordinary Lisp programmers can easily find themselves in situations where they need to nest quasiquotations. This happens because Lisp's "macro" facility works by writing Lisp macros in Lisp itself. (Most other programming languages with a macro facility use a different language to write the macros—e.g., the C preprocessor.) Once a programmer starts writing any macros at all, it is only a matter of time before he notices a situation where he has written a bunch of similar macro definitions. Clearly his next step is to design a macro-defining macro that he can use to generate all those similar looking definitions for him. In order to do this he needs nested quasiquotations.

To illustrate this, imagine that the programmer wrote the following macro definition: (This is not how macros are defined in any actual Lisp dialect.)

```
(define-macro (catch var expr)
  `(call-with-current-continuation
     (lambda (,var) ,expr)))
```

This defines `catch` as a macro so that the call

```
(catch escape
  (loop (car x) escape))
```

is expanded by binding `var` to the symbol `escape` and `expr` to the list `(loop (car x) escape)` and executing the body of the macro definition. In this example, the definition's body is a quasiquotation that returns:

```
(call-with-current-continuation
  (lambda (escape)
    (loop (car x) escape)))
```

which is then used in place of the original `catch`-expression.

Procedures that accept a single-argument auxiliary procedure, and invoke it in some special way, are a fairly common occurrence. Calls to such procedures are often written using a `lambda`-expression to create the auxiliary procedure. So later, the programmer may find himself writing another macro similar to `catch`:

```
(define-macro (collect var expr)
  `(call-with-new-collector
     (lambda (,var) ,expr)))
```

If the programmer suspects he will be writing many more instances of this kind of macro definition, he may decide to automate the process by writing the macro-defining macro:

```
(define-macro (def-caller abbrev proc)
  `(define-macro (,abbrev var expr)
     `(,',proc
        (lambda (,var) ,expr))))
```

The previous two macro definitions can then be written as

```
(def-caller catch
  call-with-current-continuation)
```

and

```
(def-caller collect
  call-with-new-collector)
```

The definition of `def-caller` would be completely straightforward if it were not for the mystical incantation comma-quote-comma (`,',`)—where the heck did *that* come from? It is *not* some new primitive notation, as comma-atsign (`,@`) was. It is the quasiquote notation and the traditional Lisp quote notation (`'`) being used together in a way that can easily be derived from their basic definitions.

Here is how we can derive the definition of `def-caller`: First, we manually expand the quasiquotation notation used in the definition of `catch`:

```
(define-macro (catch var expr)
  (list 'call-with-current-continuation
        (list 'lambda (list var) expr)))
```

Now we do not have to worry about being confused by nested quasiquotations, and we can write `def-caller` as follows:

```
(define-macro (def-caller abbrev proc)
  `(define-macro (,abbrev var expr)
     (list ',proc
           (list 'lambda (list var) expr))))
```

Now turning the calls to `list` back into quasiquotations, taking care to treat `',proc` as an expression, not a constant, yields the original definition.

Of course no Lisp programmer actually rederives comma-quote-comma every time he needs it. In practice this is a well-known nested quasiquotation cliché. Every Lisp programmer who uses nested quasiquotation knows the following three clichés:

`,X` `X` itself will appear as an expression in the intermediate quasiquotation and its value will thus be substituted into the final result.

`,,X` The value of `X` will appear as an expression in the intermediate quasiquotation and the value of that expression will thus be substituted into the final result.

`,',X` The value of `X` will appear as a constant in the intermediate quasiquotation and will thus appear unchanged in the final result.

In trying to remember or reconstruct these clichés, it can be helpful to realize that the innermost comma is associated with the outermost backquote.

3.3. Nested Splicing

The interaction of nesting with splicing yields additional interesting fruit. Since this material is not necessary to understanding the rest of this paper, the reader can skim, or even skip, the rest of this section.

The semantics of nested splicing require some explanation. Consider the case of ``` ``(,,@X) ```. In the absence of any nesting at all, `` `( ``,*anything*`)` is equivalent to `(list` *anything*`)`. So this suggests that ``` ``(,,@X) ``` should be equivalent to:

```
`(list ,@X)
```

And so the elements of the value of `X` will be expressions that will appear as separate arguments in a call to `list`.

Of course in the absence of nesting, `` `( ``,*anything*`)` is *also* equivalent to `(cons` *anything* `'())`, so one might also argue that ``` ``(,,@X) ``` should be equivalent to:

```
`(cons ,@X '())
```

```
    X      ,X      ,@X
  ,',X    ,,X    ,@,X
 ,@',X   ,,@X   ,@,@X
```

Figure 1. Two-stage markup, as used in practice.

This would clearly be less useful, as `cons` will be called with the wrong number of arguments unless the value of `X` is a single element list! The expansion in terms of `list` is the most *useful* of all the various possibilities, and so this is what is used to give nested splicing a semantics.

By similar reasoning, the most useful expansion of `‘(,@`*anything*`)` is `(append` *anything*`)`. So in order to give nested splicing a useful semantics, the code constructed by `read` for a quasiquotation makes anything that follows a comma (`,`) into an argument to `list`, and anything that follows a comma-atsign (`,@`) into an argument to `append`. The expansion algorithm in appendix A follows these rules.

Without splicing, there are two events that can happen at each stage of evaluating a nested quasiquotation: the item might be treated as a constant and passed on to the next stage untouched, or the item might be treated as an expression, whose value will appear in the next stage. Splicing adds a third possibility: the item might might be treated as an expression whose value will be a list, which will be spliced into the next stage.

With two stages, and three possible actions at each stage, there are nine cases. Each row of the table in figure 1 corresponds to a different choice at the first stage, and each column corresponds to a different choice at the second stage. In the first row and column, no action is taken at that stage. In the second row and column the value of the item is substituted at that stage. In the last row and column the value of the item is *spliced in* at that stage. So for example, `,,@X` means that the first stage value of `X` should be a *list* of expressions, and the individual second stage values of those expressions should be substituted into the final result. While `,@,@X` means that the value of `X` should be a list of expressions each of which returns a *list* of values.

Why does the table in figure 1 not simply consist of three different prefixes arranged in all possible compositions of length two? The answer is that it *could* if programmers did not naturally apply certain optimizations. As a first step towards understanding this phenomenon, consider what would happen if we rewrote the table naïvely using comma-quote (`,'`) when we wanted no evaluation at a given stage, plain comma (`,`) when we wanted evaluation, and comma-atsign (`,@`) when we wanted evaluation followed by splicing. Such a table appears in figure 2. This table looks almost like the original—only the top row, and the lower left entry are different.

The difference in the top row is easy to explain: the comma-quote prefix, followed by an S-expression that contains no quasiquotation markup, can always be replaced by the unadorned S-expression. I.e., one can always eliminate the “innermost”

```
,',’X   ,,’X   ,@,’X
 ,',X    ,,X    ,@,X
,',@X   ,,@X   ,@,@X
```

Figure 2. An attempt at regular two-stage markup.

```
          X        ,X        ,@X
       ,',X       ,,X       ,@,X
      ,@',X     ,,@X      ,@,@X

     ,',',X     ,,',X     ,@,',X
      ,',,X      ,,,X      ,@,,X
     ,@',,X     ,,@,X     ,@,@,X

    ,@',',X    ,,@',X    ,@,@',X
  ,@'(,,@X)     ,,,@X     ,@,,@X
,@'(,@,@X)     ,,@,@X    ,@,@,@X
```

Figure 3. Three-stage markup, as used in practice.

occurrence of a comma-quote. Applying this simplification to the top row of figure 2 (twice in the case of the upper left element) yields the first row of figure 1.

The lower left corner is harder to explain. We naïvely generated `,',@X`, but this only works in the particular case where the value of `X` happens to be a list of length one! The problem is easy to see if we write the entry as `,(quote ,@X)`; only if the value of `X` is a singleton list will this generate a legal `quote`-expression. The entry in the original table was `,@',X`, which works by passing the first stage value of `X`, *as a list*, on to the second stage, and performing the splice then.

These differences suggest that there might be some "algebra" of quasiquotation markup that can explain how the table in figure 1 (containing the prefixes that programmers actually use) can be mechanically derived by simplifying the entries in some more regular table. Such a system does in fact exist, and we can discover it by examining the twenty-seven possible cases where quasiquotations are nested *three* levels deep. The three-stage prefixes programmers would use in practice are shown in figure 3. The first three-by-three block contains the prefixes used when no evaluation takes place at the first stage. The second block contains the prefixes used when a simple evaluation takes place at the first stage. The third block contains the prefixes used when an evaluation followed by a splice takes place at the first stage. Within each block the second and third stage actions are arranged as before.

The most striking aspect of figure 3 is the appearance of pairs of parentheses in two of the entries. Both of those entries also contain one more atsign than we would expect, given the number of stages that involve splicing. Let us analyze the case of `,@'(,,@X)` to see what is going on. `,@'(,,@X)` expects the first stage value of `X`

```
,@'(,@'(X))   ,,@'(X)   ,@,@'(X)
    ,@'(,X)       ,,X       ,@,X
   ,@'(,@X)      ,,@X      ,@,@X
```

Figure 4. Correct regular two-stage markup.

to be a list of expressions. Those expressions will each be evaluated in the second stage. The values returned will be untouched in the third stage and will appear in the final result. The key to how this works is the prefix `,@'(···)`, which bundles up the values returned by the second stage in a list, and then splices the elements of that list into the final result.

So we conclude that the prefix we *should* have been using for a stage that performs no evaluation is `,@'(···)` and *not* `,'` as we assumed in out first attempt. This gives us the table in figure 4. Unlike our first naïve attempt, all the entries in this table work in all cases. A programmer could safely rely on composing the "operators" comma (`,`), comma-atsign (`,@`), and comma-atsign-quote-open (`,@'(···)`) to achieve the desired effect inside arbitrarily nested quasiquotations.

In practice, though, programmers prefer to use the simplified prefixes from figure 1. So what are the simplification rules that result in the prefixes programmers actually use? There are three rules, and the reason for each one is straightforward:

1. `,@'(`*expr*`)` simplifies to *expr* if *expr* contains no quasiquotation markup. We already encountered this rule (in a simpler form) above.

2. `,@'(`*expr*`)` simplifies to `,'`*expr* if *expr* contains quasiquotation markup but it does not do any splicing. If *expr* is going to be just a single value, there is no point in wrapping it up in a list and unwrapping it later—we can just return it as is.

3. `'(,@`*expr*`)` simplifies to `',`*expr* if *expr* does not do any splicing. If *expr* is going to return a list of values that we are going to splice into a constant list, we might as well just use the list directly.

So for example, two applications of rule 1 reduce `,@'(,@'(X))` to `X`. Rule 2 reduces `,@'(,X)` to `,',X`. And rule 3 reduces `,@'(,@X)` to `,@',X`.

The table in figure 3 can be generated in the same manner. The two entries where pairs of parentheses appear are cases where none of the simplification rules apply. (The first of those two entries, `,@'(,,@X)`, is of historical interest—it will appear again in section 6.)

4. Example: Turning an interpreter into a compiler

This section presents an extended example that combines quasiquotation with manual partial evaluation to straightforwardly create a simple compiler. This example serves two purposes: First, it shows quasiquotation being used for something more impressive than simply writing macros. Second, it illustrates the "environment problem" that will be the subject of section 5.

4.1. Simple Interpreter

We begin with a completely straightforward Scheme interpreter. Its main entry point is the procedure `evaluate`, which is nothing more than a big dispatch on the kind of the expression:

```
(define (evaluate exp vars vals)
  ((cond ((symbol? exp) eval-variable)
         ((pair? exp)
          (case (car exp)
            ((quote) eval-quote)
            ((if) eval-if)
            ...                   ; other special forms
            ((lambda)
             (case (length (cadr exp))
               ((0) eval-lambda-0)
               ((1) eval-lambda-1)
               ((2) eval-lambda-2)
               (else eval-lambda)))
            (else
             (case (length (cdr exp))
               ((0) eval-application-0)
               ((1) eval-application-1)
               ((2) eval-application-2)
               (else eval-application)))))
         (else eval-constant))
   exp vars vals))
```

The cases of zero, one and two argument procedures and calls are handled separately so that we can avoid the uninteresting complexity of the general cases. The environment, which is traditionally represented as an association-list, has been split into two congruent trees: `vars` is a tree of symbols representing variables, and `vals` is a tree of associated values. E.g., the traditional environment a-list

```
((a . 1) (b . 2) (c . 3))
```

might be represented using the `vars`:

```
((a . b) c)
```

and the `vals`:

```
((1 . 2) 3)
```

The `evaluate` procedure dispatches to a number of different sub-procedures depending on what kind of expression it was called upon to evaluate. Three of them are presented here. The rest of them are easy to reconstruct.

The value of a two argument `lambda`-expression is a procedure that, when called, extends `vars` and `vals` appropriately and then calls `eval-sequence` to evaluate the expressions in its body:

```
(define (eval-lambda-2 exp vars vals)
  (lambda (arg0 arg1)
    (eval-sequence (cddr exp)
                   (list (caadr exp) (cadadr exp) vars)
                   (list arg0 arg1 vals))))
```

A conditional expression is evaluated by first evaluating the test expression. If the test returns a true value, the second expression is evaluated, otherwise the third expression is evaluated:

```
(define (eval-if exp vars vals)
  (if (evaluate (cadr exp) vars vals)
      (evaluate (caddr exp) vars vals)
      (evaluate (cadddr exp) vars vals)))
```

To evaluate a two argument procedure application, the procedure expression and the argument expressions are evaluated, then the resulting procedure is applied to the resulting arguments:

```
(define (eval-application-2 exp vars vals)
  ((evaluate (car exp) vars vals)
   (evaluate (cadr exp) vars vals)
   (evaluate (caddr exp) vars vals)))
```

4.2. Threaded Code Compiler

Now we convert this interpreter into a compiler that emits threaded code built out of closures. This can be accomplished by currying the evaluator:

```
(define (evaluate exp vars vals)
  ((compile exp vars) vals))
```

The `compile` procedure is applied to a source code expression and a tree of variable names. It compiles the expression into a procedure (a closure) that will later be applied to a tree of variable values. This generated procedure will execute the given expression *much* faster than our original interpreter, because much of the overhead of interpretation will be done ahead of time by `compile`. (Feeley and Lapalme have described this technique in some depth [7].)

The dispatch performed by the original `evaluate` is now done by `compile`:

```
(define (compile exp vars)
  ((cond ((symbol? exp) compile-variable)
         ((pair? exp)
          (case (car exp)
            ((quote) compile-quote)
            ((if) compile-if)
            ...                 ; other special forms
            ((lambda)
             (case (length (cadr exp))
               ((0) compile-lambda-0)
               ((1) compile-lambda-1)
               ((2) compile-lambda-2)
               (else compile-lambda)))
            (else
             (case (length (cdr exp))
               ((0) compile-application-0)
               ((1) compile-application-1)
               ((2) compile-application-2)
               (else compile-application)))))
         (else compile-constant))
   exp vars))
```

Notice that `vals` does not appear anywhere in the definition of `compile`; each individual code generator (`compile-if`, etc.) is responsible for generating a procedure that will be applied to the tree of values.

Two argument `lambda`-expressions are compiled as follows:

```
(define (compile-lambda-2 exp vars)
  (let ((run (compile-sequence (cddr exp)
                               (list (caadr exp)
                                     (cadadr exp)
                                     vars))))
    (lambda (vals)
      (lambda (arg0 arg1)
        (run (list arg0 arg1 vals))))))
```

The body of the `lambda`-expression is compiled in advance by `compile-sequence` and the generated procedure is simply a closure that holds the compiled body as the value of the closed-over variable `run`.

Conditional expressions and two argument applications are compiled as follows:

```
(define (compile-if exp vars)
  (let ((run-test (compile (cadr exp) vars))
        (run-true (compile (caddr exp) vars))
        (run-false (compile (cadddr exp) vars)))
    (lambda (vals)
      (if (run-test vals)
```

```
            (run-true vals)
            (run-false vals)))))

(define (compile-application-2 exp vars)
  (let ((run-fun (compile (car exp) vars))
        (run-arg0 (compile (cadr exp) vars))
        (run-arg1 (compile (caddr exp) vars)))
    (lambda (vals)
      ((run-fun vals) (run-arg0 vals) (run-arg1 vals)))))
```

In both cases three sub-expressions are compiled, and then an appropriate closure is created.

Most of the rest of the cases are similar to the three just presented: First, recursively compile the various sub-expressions of the given expression, and then create an appropriate closure that holds those compiled sub-expressions as the values of the closed-over variables.

An important source of efficiency in the generated code is the behavior of the procedure `compile-variable`. When compiling a variable, the most efficient procedure it can return is one of the primitive list structure accessors. E.g., a call like:

```
(compile-variable 'y '(x (y z)))
```

should return the primitive `caadr` procedure.

4.3. Scheme-to-Scheme Compiler

With very little additional work, we can now transform our compiler into one that generates Scheme source code. The currying operation we performed on our original evaluator amounted to a manual act of partial evaluation: We separated the original arguments to the interpreter into static arguments (`exp` and `vars`) and dynamic arguments (`vals`), and then arranged for all the computation that only depends on static information to take place during the call to `compile`. Now all we have to do is to change the code generators so that instead of generating closures, they generate Scheme source code.

Because it does not generate any code itself, the procedure `compile` is unchanged.

We can convert `compile-lambda-2` from a closure generator into a Scheme source code generator with the insertion of just two characters:

```
(define (compile-lambda-2 exp vars)
  (let ((run (compile-sequence (cddr exp)
                               (list (caadr exp)
                                     (cadadr exp)
                                     vars))))
    `(lambda (vals)
       (lambda (arg0 arg1)
         (,run (list arg0 arg1 vals))))))
```

A backquote has been placed in front of the `lambda`-expression that previously created the returned procedure, and a comma has been placed in front of the variable `run`.

The code generators for conditional expressions and two argument applications can also be converted to generate Scheme source code by merely inserting a backquote and some commas:

```
(define (compile-if exp vars)
  (let ((run-test (compile (cadr exp) vars))
        (run-true (compile (caddr exp) vars))
        (run-false (compile (cadddr exp) vars)))
    `(lambda (vals)
       (if (,run-test vals)
           (,run-true vals)
           (,run-false vals)))))

(define (compile-application-2 exp vars)
  (let ((run-fun (compile (car exp) vars))
        (run-arg0 (compile (cadr exp) vars))
        (run-arg1 (compile (caddr exp) vars)))
    `(lambda (vals)
       ((,run-fun vals) (,run-arg0 vals) (,run-arg1 vals)))))
```

In all three cases the expressions constructed by quasiquote have no free variables other than primitive Scheme procedures. The variables that *would* have been free (`run`, `run-test`, `run-fun`, etc.) now appear preceded by commas. Thus, at every stage of the process, `compile` returns a closed expression (modulo primitives), and each code generator preserves that property in the code that it constructs.

Compiling variables is changed so that instead of returning a list structure accessor, it returns the *name* of the appropriate accessor. E.g.,

```
(compile-variable 'y '(x (y z)))
```

now returns the symbol `caadr`.

Using this new version of `compile`, the result of compiling this expression:

```
(lambda (n)
  (if (< n 2)
      1
      (* (fact (- n 1))
         n)))
```

is essentially (after a few trivial β-reductions):

```
(lambda (vals)
  (lambda (arg0)
    ((lambda (vals)
       (if (< (car vals) '2)
```

```
          '1
          (* (fact (- (car vals) '1))
             (car vals))))
    (cons arg0 vals))))
```

Passing the output of `compile` on to a native Scheme compiler will produce a plausible machine code implementation of the original Scheme code.

Of course, we could have fed our original Scheme code *directly* to the native Scheme compiler, and gotten even better machine code. But this exercise is far from pointless, for we can go back to the beginning and start over with an interpreter for some *new* language of our own devising. Then we can follow these same steps to arrive at a version of `compile` that compiles our new language into ordinary Scheme. And then we can compile that Scheme code into native machine code.

5. The Environment Problem

Quasiquote is most often used to construct programming-language expressions. Since quasiquote has no understanding that the lists it constructs are actually expressions with bound variables and scoping rules, the programmer must pay attention to such environment issues himself—a task which can be error prone. In this section, we examine the problems, and survey some of the possible solutions, many of them from beyond the borders of Lisp.

5.1. The problem

The correctness of the compiler in the previous section depended on the property that all the expressions it constructed using quasiquote were closed (modulo Scheme primitives). If some of the expressions had free variables, it would have been necessary to prove that each such variable eventually found itself built into a scope where it was suitably bound. Such proofs can be tricky to construct.

To illustrate the hazards, imagine that we were writing a compiler that generated simple Scheme expressions. We might write a code generating procedure such as:

```
(define (make-list-proc expr const)
  `(lambda (x) (list x ,expr const)))
```

Our intent is to produce the code for a procedure that makes a list of three things, where the first thing will be the argument to the generated procedure, the second thing will result from evaluating an expression that we supply, and the third thing is a given constant.

If we evaluate `(make-list-proc '(+ x 1) 'foo)`, the answer is:

```
(lambda (x) (list x (+ x 1) const))
```

There are two problems here:

- The expression we passed to `make-list-proc` as its first argument had a free variable, `x`, that has unintentionally become bound by the `lambda`-expression that `make-list-proc` constructed. We might call this the "downward capture" problem.
- The variable `const` is free in the constructed expression—it does not refer in any way to the constant `foo` we supplied, despite the fact that the second argument to `make-list-proc` was named `const`. We might call this the "upward escape" problem.

Since quasiquote is just a tool for constructing list structure, it cannot help us with either of these problems. So we must solve such environment problems ourselves, perhaps by writing:

```
(define (make-list-proc expr const)
  (let ((x (gensym)))
    `(lambda (,x) (list ,x ,expr ',const))))
```

This avoids the first problem by generating a fresh variable for use in the `lambda`-expression, and by explicitly building the constant in to the result as a literal.

5.2. Possible solutions

In traditional Lisp dialects (including Common Lisp) programmers are given no assistance in avoiding environment problems except a facility for generating new symbols (usually via a `gensym` procedure). This is the technique employed in the revised `make-list-proc` procedure given above. Using generated symbols, and otherwise carefully chosen names, a programmer *can* write macros that are 100% free of capture problems, but this can be difficult. Most macro writers only bother to prevent the most likely name capture problems, but in practice this approach really is good enough.

Since the mid-1980s, the Scheme community has been experimenting with various technologies that allow (or force) programmers to easily write macros that are free from environment problems. Some of these technologies are high-level macro defining facilities that do not use quasiquotation at all [5, 10], and so they will not concern us here. Others add new low-level tools that are used together with quasiquotation [1, 3].

In one of these low-level systems, a macro definition such as:

```
(define-macro (push expr var)
  `(set! ,var (cons ,expr ,var)))
```

might be written:

```
(define-macro (push expr var)
  (let ((*set! (rename 'set!))
        (*cons (rename 'cons)))
    `(,*set! ,var (,*cons ,expr ,var))))
```

The calls to `rename` somehow arrange to look up the given identifiers in some well-known environment. The details of how something like `rename` works are beyond the scope of this paper, all that matters here is that quasiquotation itself is not altered or extended in any of these systems—the environment problem is solved by some orthogonal mechanism.

Weise and Crew's "syntax macros" for C [22] addresses the environment problem exactly as it is addressed in traditional Lisp: the programmer gets a quasiquote mechanism for building C program syntax trees, plus a `gensym` facility. It is hardly surprising that they emulated traditional Lisp so closely, since their goal was simply to bring the benefits of Lisp's macros to C.

Recently, there has been a growing interest in "staged computation" [4, 6, 8, 20]. Such systems usually employ a quasiquote-like mechanism for constructing code, and thus must somehow address the environment problem.

Davies and Pfenning's Mini-ML$^{\Box}$ language [4] takes the same minimalist approach that the compiler in the previous section took: code constructed by its quasiquote operation (named "box") is always closed, so there can be no unintended variable captures during substitution. Such strict separation of stages is perfectly reasonable in a research language, but might be considered inconvenient in a tool used by front-line programmers.

Any practical system for staged computation will probably have the property Taha and Sheard call "cross-stage persistence" [20]. A system has cross-stage persistence if variables bound in earlier stages can still be referenced in later stages. Their MetaML language has this property, as does the 'C language [6].

One of the reasons our first version of `make-list-proc` failed was because Lisp's quasiquote lacks cross-stage persistence: The symbol `const` appearing in the quasiquote template had no connection to the variable `const` bound in the environment where the quasiquotation was written. With cross-stage persistence, variables in constructed code *can* reference variables from the lexically enclosing environment. In such a system the code objects returned by a quasiquotation must be more than a simple S-expression-like data structure; in order to capture the lexical environment where the quasiquote expression was written, code objects must be closures.[2]

Code objects with cross-stage persistence are similar to Lamping's parameterized objects [11]. Lamping was also motivated by a desire to manipulate expressions in the way that quasiquotation would allow, but without disconnecting those expressions from the context that they came from. His `data`-expressions are another example of something like a quasiquotation that manufactures something like a closure.

6. History

The name "Quasi-Quotation" was coined by W. V. Quine [16] around 1940. Quine's version of quasiquotation was character-string based. He used "corner quotes" ($\ulcorner \cdots \urcorner$) to indicate a quasiquotation, and he had no explicit marker for "unquote," instead any Greek letter was implicitly marked for substitution. So for example, if ϕ denoted the string "`bar`", then

$\ulcorner$`foo(`ϕ`)`$\urcorner$

would denote the string "`foo(bar)`".

Quine used quasiquotation to construct expressions from mathematical logic, and just as we would predict from our experience representing expressions from C, he was forced to adopt various conventions and abbreviations involving parentheses. (He could have avoided these difficulties if he had used S-expressions instead!)

McCarthy developed Lisp and S-expressions [13] around 1960, but did not propose any form of S-expression based quasiquotation. Given that he was inspired by the λ-calculus, which has its own notion of substitution, one cannot help but wonder what Lisp would have been like if he had also tried to work quasiquotation into the mixture.

During the 1960s and 1970s the artificial intelligence programming community expended a lot of effort learning how to program with S-expressions and list structure. Many of the AI programs from those years developed Lisp programming techniques that involved S-expression pattern matching and template instantiation in ways that are connected to today's S-expression quasiquotation technology. In particular, the notion of splicing, described in section 3.1, is clearly descended from those techniques.

But nothing from those years resembles today's Lisp quasiquotation notation as closely as the notation in McDermott and Sussman's Conniver language [14]. In Conniver's notation `` `X``, `,X` and `,@X` were written `!"X`, `@X` and `!@X` respectively, but the idea was basically the same. (Conniver also had a `,X` construct that could be seen as similar to `@X`, so it is possible that this is how the comma character eventually came to fill its current role.)

After Conniver, quasiquote entered the Lisp programmer's toolkit. By the mid-1970s many Lisp programmers were using their own personal versions of quasiquote. Quasiquote was not yet built in to any Lisp dialect.

My personal knowledge of this history starts in 1977 when I started programming for the Lisp Machine project at MIT. At that time quasiquotation was part of the Lisp Machine system. The notation was almost the same as the modern notation, except that `,,X` was used instead of `,@X` to indicate splicing. This would obviously interfere with nested quasiquotation, but this did not bother anyone because it was commonly believed that nested quasiquotation did not "work right."

I set out to figure out why nested quasiquotation should fail to work. Employing the same reasoning process I outlined above in section 3.2, I developed some test cases and tried them out. To my surprise, they actually worked perfectly. The problem was simply that no one had been able to make nested quasiquotation do what they wanted, not that there was a fixable bug.[3]

Now that we knew that nested quasiquotation did in fact work, we wanted to start using it, and so a new notation for splicing had to be found. I suggested `,.X` because `.,X` already does a kind of splicing (see section 3.1). I thought that this would be a good pun. Other members of the group thought it might be confusing. Probably inspired by Scheme, which in those days was using just `@X` to indicate splicing [19], we finally decided on `,@X` [21].[4]

Meanwhile McDermott altered the Conniver notation slightly by changing `!"X` to `|"X`. In this form it appeared in the first edition of *Artificial Intelligence Programming*[2] in 1980.

As far as I know, the problems of nested splicing did not get worked out until 1982. In January of that year Guy Steele circulated an example of quasiquotations nested *three* levels deep. He remarked that `,',',X` was "fairly obvious," but that it took him "a few tries" to get his use of `,,@X` right [17]. I responded with an analysis of nested splicing (very similar to the one given here in section 3.3) in which I observed that in order to get nested splicing to behave in a consistent and useful manner, an expansion algorithm like the one presented in appendix A was required. A correct semantics and expansion algorithm for quasiquotation based on this observation now appears in the second edition of *Common LISP: The Language* [18].

Guy Steele has kindly forwarded to me all of the quasiquote-related messages from his January 1982 electronic mail archives. Among those messages is another example of triple quasiquotation. The example occurs in a message sent to Steele by Bill Long [12] containing some code originally written by David McAllester. The code contains the construct `,@',(list ,@params)`. Since `,(list` *expr*`)` is equivalent to `(,`*expr*`)`, McAllester's code could also have been written as: `,@'(,,@params)`, which the reader may recognize from section 3.3 as one of the two simplest examples of nested splicing that cannot be expressed without using parentheses. So McAllester may well have been the first man to scale that particular quasiquotation mountain.

Sometime during the 1980s we started to spell "quasi-quote" without the hyphen. Perhaps this is the result of the adoption of a special form named "`quasiquote`" into Scheme.

By the end of the 1980s, the standards for Common Lisp [18] and Scheme [9] had adopted the modern quasiquote notation.

7. Self-generation

No paper on quasiquotation in Lisp would be complete without mentioning quasiquote's contribution to the perennial problem of writing a Lisp expression whose value is itself. The quasiquote notation enables many elegant solutions to this problem. The following amusing solution is due to Mike McMahon:

```
(let ((let '`(let ((let ',let))
               ,let)))
  `(let ((let ',let))
     ,let))
```

8. Conclusion

Quasiquotation is a simple idea that combines synergistically with Lisp's S-expressions (also a simple idea!) to create a powerful tool for program construction. Like many simple ideas, a lot of complexity emerges when all of the details are worked out. In this case, the complete interaction of nesting with splicing yields an interesting "algebra of unquotation".

Quasiquotation does have a limitation as a program constructing tool: It has no understanding of the variable scoping rules obeyed by the expressions it is constructing. A variety of different approaches have been proposed to cope with this limitation. This is an active area of current research.

Despite the powerful synergy between quasiquotation and S-expressions, it took almost twenty years from the invention of Lisp for quasiquote to achieve widespread acceptance. The modern quasiquote era has now lasted roughly an additional twenty years. If the *next* twenty years hold any change in store for quasiquote, it will likely be the addition of some solution to the environment problem.

Acknowledgments

Parts of this paper can trace their lineage back to two electronic mail conversations I had in 1982, one with Drew McDermott and one with Guy Steele. While collecting additional information I had helpful discussions with Robert Muller, Brian C. Smith, Gerald Jay Sussman, John Lamping, Glenn Burke, Jonathan Bachrach, Daniel Weise, Mitch Wand, Kent Dybvig, Guy Steele, David McAllester, David Moon, Bill Long and Marc Feeley. Pandora Berman assisted in the research. Olivier Danvy, Julia Lawall and Charles Stewart provided helpful feedback on various drafts of this paper—any remaining problems are entirely my own fault.

Appendix A

This appendix contains a correct S-expression quasiquotation expansion algorithm.

Assume that some more primitive Lisp parser has already read in the quasiquotation to be expanded, and has somehow tagged all the quasiquotation markup. This primitive parser has also supplied the following four functions:

`tag-backquote?` This predicate will be true of the result of reading a backquote (`` ` ``) followed by an S-expression.

`tag-comma?` This predicate will be true of the result of reading a comma (`,`) followed by an S-expression.

`tag-comma-atsign?` This predicate will be true of the result of reading a comma-atsign (`,@`) followed by an S-expression.

`tag-data` This function should be applied to an object that satisfies one of the previous three predicates. It will return the S-expression that followed the quasiquotation markup.

The main entry point is the function `qq-expand`, which should be applied to an expression that immediately followed a backquote character. (I.e., the outermost backquote tag should be stripped off *before* `qq-expand` is called.)

```
(define (qq-expand x)
  (cond ((tag-comma? x)
         (tag-data x))
        ((tag-comma-atsign? x)
         (error "Illegal"))
        ((tag-backquote? x)
         (qq-expand (qq-expand (tag-data x))))
        ((pair? x)
         `(append ,(qq-expand-list (car x))
                  ,(qq-expand (cdr x))))
        (else `',x)))
```

Note that any embedded quasiquotations encountered by `qq-expand` are recursively expanded, and the expansion is then processed as if it had been encountered instead.

`qq-expand-list` is called to expand those parts of the quasiquotation that occur inside a list, where it is legal to use splicing. It is very similar to `qq-expand`, except that where `qq-expand` constructs code that returns a value, `qq-expand-list` constructs code that returns a list containing that value.

```
(define (qq-expand-list x)
  (cond ((tag-comma? x)
         `(list ,(tag-data x)))
        ((tag-comma-atsign? x)
         (tag-data x))
        ((tag-backquote? x)
         (qq-expand-list (qq-expand (tag-data x))))
        ((pair? x)
         `(list (append ,(qq-expand-list (car x))
                        ,(qq-expand (cdr x)))))
        (else `'(,x))))
```

Code created by `qq-expand` and `qq-expand-list` performs all list construction by using either `append` or `list`. It must never use `cons`. As explained in section 3.3, this is important in order to make nested quasiquotations containing splicing work properly.

The code generated here is correct but inefficient. In a real Lisp implementation, some optimization would need to be done. A properly optimizing quasiquotation expander for Common Lisp can be found in [18, Appendix C].

Appendix B

The Scheme standard [9] makes it hard to implement nested splicing as described in this paper. The standard explicitly specifies that `` `X ``, `,X` and `,@X` must be read in as `(quasiquote X)`, `(unquote X)` and `(unquote-splicing X)` respectively. This means that ``` ``(,,@abc) ``` must be equivalent to:

```
(quasiquote (quasiquote ((unquote (unquote-splicing abc)))))
```

and if the value of `abc` is `(a b c)`, then the expression above must evaluate to:

```
(quasiquote ((unquote a b c)))
```

But the language standard does not say anything about what `unquote` means if it is not used with a single sub-expression. Many Scheme implementations will ignore the additional expressions and treat `(unquote a b c)` as if it were `(unquote a)`. Other implementations will signal an error.

At first this looks like a horrible situation: it appears that by overspecifying the read-time behavior of the backquote notation, the Scheme standard has ruined the possibility of having nested splicing behave in the useful way it does in Common Lisp.

Fortunately we can still recover proper nested splicing behavior by making an upward compatible extension to the Scheme standard that assigns a useful meaning to `unquote` and `unquote-splicing` expressions that contain other than a single sub-expression. This extension is the macro-expansion-time analog of the read-time rules given in section 3.3: When a quasiquotation is expanded, each sequence of sub-expressions found inside a `unquote` must be treated as if they were sequential arguments in a call to `list`. Similarly, each sequence of sub-expressions found inside a `unquote-splicing` must be treated as if they were sequential arguments in a call to `append`. So

```
(quasiquote (a (unquote x y) b))
```

should behave the same as

```
(list 'a x y 'b)
```

and

```
(quasiquote (a (unquote-splicing x y) b))
```

should behave the same as

```
(append '(a) x y '(b))
```

The `qq-expand` procedure here differs from the one in appendix A in that it is applied at macro expansion time, rather than at read-time. For this reason it must keep a counter of nested quasiquotations so that it can match occurrences of `unquote` and `unquote-splicing` with the correct enclosing `quasiquote`. This counter is the `depth` argument below. It should initially be supplied as `0`.

As before, the outermost `quasiquote` should be stripped off *before* `qq-expand` is called.

```
(define (qq-expand x depth)
  (if (pair? x)
      (case (car x)
        ((quasiquote)
         `(cons 'quasiquote
```

```
               ,(qq-expand (cdr x) (+ depth 1))))
       ((unquote unquote-splicing)
        (cond ((> depth 0)
               `(cons ',(car x)
                      ,(qq-expand (cdr x) (- depth 1))))
              ((and (eq? 'unquote (car x))
                    (not (null? (cdr x)))
                    (null? (cddr x)))
               (cadr x))
              (else
               (error "Illegal"))))
       (else
        `(append ,(qq-expand-list (car x) depth)
                 ,(qq-expand (cdr x) depth))))
     `',x))
```

As before, qq-expand-list is called to expand those parts of the quasiquotation that occur inside a list, where it is legal to use splicing.

```
(define (qq-expand-list x depth)
  (if (pair? x)
      (case (car x)
        ((quasiquote)
         `(list (cons 'quasiquote
                      ,(qq-expand (cdr x) (+ depth 1)))))
        ((unquote unquote-splicing)
         (cond ((> depth 0)
                `(list (cons ',(car x)
                             ,(qq-expand (cdr x) (- depth 1)))))
               ((eq? 'unquote (car x))
                `(list . ,(cdr x)))
               (else
                `(append . ,(cdr x)))))
        (else
         `(list (append ,(qq-expand-list (car x) depth)
                        ,(qq-expand (cdr x) depth)))))
      `'(,x)))
```

Notes

1. Actually, in Scheme [9], the exact expansion *is* specified: it expands into a special `quasiquote` expression. But this fact is useless in practice, and it will be ignored except in appendix B.
2. The quasiquotes in these staged computation systems differ in many other ways from Lisp's quasiquote. They only manipulate code objects that represent expressions, so there is no danger of building a syntactically incorrect code object. Code objects have types like "code returning `int`," so there is no danger of building a code object in which types fail to check. And because there are no code objects that represent sequences of anything, there is no need for any notion of splicing.

3. The expander in use at that time did have bugs handling nested splicing—but I did not notice them.
4. When quasiquotation was migrated to MacLisp [15] `,.X` was chosen to mean a "destructive" version of splicing. They also thought it was a good pun. This notation was also adopted in Common Lisp [18].

References

1. Alan Bawden and Jonathan Rees. Syntactic closures. In *Proc. Symposium on Lisp and Functional Programming*, pages 86–95. ACM, July 1988.
2. Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott. *Artificial Intelligence Programming*. Lawrence Erlbaum Assoc., Hillsdale, NJ, first edition, 1980.
3. William Clinger. Hygienic macros through explicit renaming. *LISP Pointers*, 4(4):25–28, December 1991.
4. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proc. Symposium on Principles of Programming Languages*, pages 258–283. ACM, January 1996.
5. R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1992.
6. Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. ‘C: A language for fast, efficient, high-level dynamic code generation. In *Proc. Symposium on Principles of Programming Languages*, pages 131–144. ACM, January 1996.
7. Marc Feeley and Guy Lapalme. Using closures for code generation. *Journal of Computer Languages*, 12(1):47–66, 1987.
8. Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Proc. Symposium on Principles of Programming Languages*, pages 86–96. ACM, January 1986.
9. Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.
10. Eugene M. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proc. Symposium on Lisp and Functional Programming*, pages 151–161. ACM, August 1986.
11. John Lamping. A unified system of parameterization for programming languages. In *Proc. Symposium on Lisp and Functional Programming*, pages 316–326. ACM, July 1988.
12. William J. Long, January 1982. Electronic mail message to Guy Steele. Available on-line from `ftp://ftp.bawden.org/archive/wjl-to-gls-11Jan1982.txt`.
13. John McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.
14. Drew V. McDermott and Gerald Jay Sussman. The Conniver reference manual. Memo 259a, MIT AI Lab, May 1972.
15. Kent M. Pitman. The revised MacLisp manual. TR 295, MIT LCS, May 1983.
16. Willard Van Orman Quine. *Mathematical Logic*. Harvard University Press, revised edition, 1981.
17. Guy L. Steele Jr., January 1982. Electronic mail message. Available on-line from `ftp://ftp.bawden.org/archive/gls-15Jan1982.txt`.
18. Guy L. Steele Jr. *Common LISP: The Language*. Digital Press, second edition, 1990.
19. Guy L. Steele Jr. and Gerald Jay Sussman. The revised report on SCHEME: A dialect of Lisp. Memo 452, MIT AI Lab, January 1978.
20. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217. ACM, June 1997.
21. Daniel Weinreb and David Moon. *Lisp Machine Manual*. Symbolics Inc., July 1981.
22. Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. Conference on Programming Language Design and Implementation*, pages 156–165. ACM, June 1993.