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. In this paper I will review the history and development of this technology, and explain why it works so well in practice.

1 Introduction

The subject of this paper is "quasiquotation". 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 [4] and Common Lisp [13], where it is used to write syntactic extensions ("macros") and other program-generating programs.

I have two goals in this paper. First, I wish to simply introduce the reader to quasiquotation and to record some of its history. Second, I wish to draw attention to the under appreciated synergy between quasiquotation and Lisp's "S-expression" data structures.

In this paper when I use the word "Lisp" I will mean primarily the Lisp dialects Common Lisp and Scheme, although what I say will usually be true of most other Lisp dialects. When I write examples of Lisp code, I will write in Scheme, although what I write will often run correctly in many other Lisp dialects.

0

2 Why Quasiquotation?

Before looking at how you would use quasiquotation to write program-generating programs in Lisp, let us consider what would happen if you had to use the C programming language instead. 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

Suppose that you are writing a C program that generates another C program. The most straightforward way for your program to accomplish its task is for it to textually construct the output program via string manipulation. Your 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 and to specialize it as required. Without `fprintf`, you would have had 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);
```

You can tell at a glance that the `fprintf` statement generates a syntactically legal C statement, but when looking at the sequence of calls to `fputs`, this isn't as clear.

Using `fprintf` achieves the central goal of quasiquotation: It assists you by allowing you to write expressions that look as much like the desired output as possible. You can write down what you want the output to look like, and then modify it only slightly in order to parameterize it.

Although `fprintf` makes it *easier* for you to write C programs that generate C programs, two problems with this technology will become clear to you after you have used it for a while:

- The parameters are associated with their values positionally. You have to count arguments and occurrences of "`%s`" to figure out which matches up with which. If there are a large number of parameters, errors will 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 happens if `array_name` was "`*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. You would much rather write 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,[1] you will still be left with the second problem: flat character strings are not really a very good way to represent recursive structures like expressions. You will probably wind up adopting some convention that inserts extra pairs of parenthesis into your output just to be sure that it parses the way you intended.[2]

We have identified three goals for a successful implementation of quasiquotation:

- Quasiquotation should enable the programmer to write down what she wants 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.

As we shall see, the achievement of that last goal is where Lisp will really shine.

2.2 Quasiquotation in Lisp

Now suppose that you are 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 for it to work with Lisp's "S-expression" data structures: lists, symbols, numbers, etc. So suppose your aim is to generate a Lisp expression such as:

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

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 will let you 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 can a Lisp implementation actually make this *work*? What is the underlying technology?

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 . ()))
```

[1] I can think of several ways to do it—none of them very pleasant.

[2] A technique often employed by users of the C preprocessor for the very same reason.

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 you can write:

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

and it will be exactly as if you 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.[3] (One possible expansion algorithm is described in appendix A.) So the backquote notation doesn't change the fact that your program-generating Lisp program works by manipulating Lisp's list structures.

Clearly this backquote notation achieves at least the first two of our three goals for quasiquotation: the code closely resembles the desired output and the parameter expressions appear directly where their values will be inserted.

Our 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.

We can represent expressions using list structure, but 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 data structures that capture even more of the features of the given domain.

But this would be unnecessary complexity.

The problem with strings was that string substitution didn't respect the intended recursive structure of expressions represented as strings. But list structure substitution *does* respect the recursive structure of expressions represented as lists–and we haven't yet identified any additional problems that switching to an even higher-level representation would solve. The introduction of additional data types, and the procedures to manipulate them, would certainly introduce additional complexity into the system. The question is, would that complexity pay for itself by solving some problem, or making something more convenient?

For example, perhaps a well-designed set of abstract data types for various programming language constructs would prevent us from accidentally using quasiquotation to construct programs with illegal syntax. 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.

Such possibilities can't be ruled out, but after twenty years in its present form, no clearly superior representations for quasiquotation to work with have appeared. There just don't seem to be any compelling reasons to complicate matters my moving up to a higher-level representation. All three of our goals for a quasiquotation technology are nicely achieved by S-expression-based quasiquotation.

2.3 Synergy

In fact, there's a wonderful synergy between quasiquotation and S-expressions. They work together to yield a technology that's 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 seeing what this does:

```
(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)
```

[3] Actually, in Scheme [4], the exact expansion *is* specified: it expands into a special `quasiquote` expression. But I have never seen a programmer take advantage of this fact in a way that wasn't somehow problematic, so I am skeptical of its utility, and I will therefore ignore it in this paper.

S-expressions were at the core of McCarthy's original version of Lisp [6]. 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've seen, character string based quasiquotation is an untidy way to work with 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 didn't 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 then.

3 Embellishments

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

3.1 Splicing

We brushed very close to needing splicing in a previous example. 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 knew (for some reason) that that list did not include an `else`-clause, and we wanted 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.[4]

The expanded code might read:

```
(cons 'cond
      (cons (list (list 'eq?
                        var
                        (list 'quote
                              val))
                  expr)
            (append more-clauses
                    '((else #T)))))
```

The reader who finds this expansion unenlightening has my sympathy. 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 some sub-part occurs: 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.

There is no analog of splicing for character string based quasiquotation. (This, incidentally, is another argument for the superiority of S-expression based quasiquotation.)

[4]Given the example, the reader may well wonder why the prefix comma-period (`,.`) wasn't chosen instead. The history section will address this question!

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".

This 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.[5] 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 looking 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 point, imagine that you had written the following macro definition:[6]

```
(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 will return:

```
(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 you may later find yourself writing another macro similar to `catch`:

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

If you suspect you'll be writing many more instances of this kind of macro definition, you 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 wasn't 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 you could have arrived at the definition of `def-caller`: First, 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 you don't have to worry about being confused by nested quasiquotations, and you can write `def-caller` this way:

```
(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 a quasiquotation, 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 she needs it. In practice this is a well-known nested quasiquotation cliché. Every Lisp programmer who uses nested quasiquotation knows the following three clichés:

---

[5] Most other programming languages with a macro facility use a different language to write the macros (e.g., the C preprocessor).

[6] This is not how macros are defined in any actual Lisp dialect that I am aware of—but this isn't a paper about how to write Lisp macros.

`,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.

3.3 Nested Splicing

The interaction of nesting with splicing yields additional interesting fruit. Beyond the three clichés listed at the end of the previous section, things like `,@,X`, `,,@X`, `,@,@X` and `,@',X` will occasionally prove useful. To illustrate the possibilities consider just the following two cases:

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

`,,@X` The value of `X` will appear as a *list* of expressions in the intermediate quasiquotation. The individual values of those expressions will be substituted into the final result.

Intuitively, an atsign has the effect of causing the comma to be "mapped over" the elements of the value of the following expression.

Making nested splicing work properly in all cases is difficult. The expander in appendix A gets it right, but at the expense of expanding into atrocious code.

4 History

The name "Quasi-Quotation" was coined by W. V. Quine [10] around 1940. Quine's version of quasiquotation was character string based. He had no explicit marker for "unquote", instead any Greek letter was implicitly marked for replacement. 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 should clearly have used S-expressions instead!)

McCarthy developed Lisp and S-expressions [6] around 1960, but he did not propose any form of S-expression based quasiquotation.[7]

During 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 [7]. 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.)

The Conniver Manual credits the MDL language [3] for inspiring some of Conniver's features. MDL's notation for data structure construction is related, but it is sufficiently different that I'm unwilling to call it a direct ancestor of today's quasiquote. I'll have more to say about this issue in section 5.1.

After Conniver, quasiquote entered the Lisp programmer's toolkit. By the mid-1970s many Lisp programmers were using their own personal versions of quasiquote—it wasn't 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 being used instead of `,@X` to indicate splicing. This would obviously interfere with nested quasiquotation, but this didn't 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.[8]

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

[7] Given that he was inspired by the λ-calculus, which has its own notion of substitution, one can't help but wonder what Lisp would have been like if he had also tried to work quasiquotation into the mixture!

[8] The expander in use at that time did have bugs handling nested splicing—but I didn't notice that.

`,.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 [14], we finally decided on `,@X` [15].[9]

Meanwhile McDermott altered the Conniver notation slightly by changing `!"X` to `|"X`. In this form it appeared in [1] in 1980.

As far as I know, the problems of nested splicing didn't 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 [12]. I responded with an analysis of nested splicing in which I observed that in order to get nested splicing correct, 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 [13].

Sometime during the 1980s we started to spell "quasi-quote" without the hyphen. My guess is that 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 [13] and Scheme [4] had adopted the modern quasiquote notation.

5 Related Ideas

Here are three ideas related to quasiquotation that I think the reader might be interested in.

5.1 Alternate views of quotation

The backquote notation for quasiquotation (`` `X ``) is clearly inspired by Lisp's "forward quote" notation for ordinary quotation (`'X`). While the backquote notation is an abbreviation for a (potentially) complex series of calls to various list constructors, the forward quote notation is an abbreviation for a simple `quote` expression. (I.e., `'X` is the same as `(quote X)`.) McCarthy invented `quote` expressions as a mechanism for representing the constants that appeared in his M-expression language [6].

Smith [11] and Muller [8] have both argued that that there is something suspect about McCarthy's `quote`. Both are worried that `quote` somehow confuses levels of representation and reference. They would like to replace Lisp's `eval` function with something more in line with the λ-calculus notion of normalization.

Now given that a backquote followed by an expression that contains no commas is indistinguishable from a front quote, and given that these authors have concerns about front quote, they presumably have the same concerns about backquote. So it is interesting that both Smith's 2-LISP and Muller's M-LISP resemble the MDL language [3] in that expressions are notationally distinct from constants, and so constants are (in some sense) implicitly quasiquotations. E.g. an expression like

```
<cdr (X <+ 2 3> Z)>
```

returns

```
(5 Z)
```

This is why I would deny MDL's direct ancestry, via Conniver, of modern Lisp quasiquotation. MDL and Conniver were wandering around in the dark looking for convenient ways to construct list structure. Both found solutions, and Conniver may even have thought that it was following MDL. But in the light of hindsight, we can recognize that Conniver was firmly on the traditional Lisp path, while MDL had stepped off that path and had started in the direction suggested by Smith and Muller.

5.2 Parameterized code

The ‘C language [2] adds a backquote operator to ordinary C as a way of specifying dynamically generated code. Their backquoted expressions, while explicitly inspired by Lisp's use of backquote, do not construct ordinary C data structures (such as structures and arrays), they only build dynamic code objects. This is not surprising since C has no ordinary data structures for representing C programs (other than character strings). Inside a backquote, atsign (`@`) is used to indicate the substitution of another dynamic code object. Splicing is meaningless and backquotes do not nest.

A particularly interesting difference from Lisp's quasiquotation is that in ‘C backquoted code *can* reference variables from the lexically enclosing code. I.e., dynamic code objects are also closures. So a ‘C backquote expression is sort of a cross between a quasiquotation and a λ-expression.

This is similar to Lamping's system of parameterization [5]. His `data`-expressions specify parameterized objects that are sort of cross between a quasiquotation and a closure. Lamping was motivated by a desire to manipulate expressions the way that quasiquotation would allow, but without disconnecting them from the context that they came from.

[9] When quasiquotation was migrated to MacLisp [9] `,.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 [13].

Systems like these demonstrate that there is a lot of unexplored territory in between the notion of data and the notion of expression.

5.3 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, but the following solution, due to Mike McMahon, is my personal favorite:

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

6 Conclusion

It took a while for the Lisp community to discover it, but there's a synergy between quasiquotation and S-expressions.

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 and Jonathan Bachrach. It was Olivier Danvy who thought that the world needed a paper about quasiquotation in Lisp. Julia Lawall provided helpful feedback on early drafts. Pandora Berman also assisted in the preparation of this paper.

References

[1] E. Charniak, C. K. Riesbeck, and D. V. Mcdermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Assoc., Hillsdale, NJ, first edition, 1980.

[2] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. ‘C: A language for fast, efficient, high-level dynamic code generation. In *Proc. Symposium on Principles of Programming Languages.* ACM, Jan. 1996.

[3] S. W. Galley and G. Pfister. The MDL programming language. TR 293, MIT LCS, May 1979.

[4] R. Kelsey, W. Clinger, and J. 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.

[5] J. Lamping. A unified system of parameterization for programming languages. In *Proc. Symposium on Lisp and Functional Programming*, pages 316–326. ACM, July 1988.

[6] J. McCarthy. *LISP 1.5 Programmer's Manual.* MIT Press, 1962.

[7] D. V. McDermott and G. J. Sussman. The Conniver reference manual. Memo 259a, MIT AI Lab, May 1972.

[8] R. Muller. M-LISP: A representation-independent dialect of LISP with reduction semantics. *ACM Transactions on Programming Languages and Systems*, 14(4):589–616, Oct. 1992.

[9] K. M. Pitman. The revised MacLisp manual. TR 295, MIT LCS, May 1983.

[10] W. V. Quine. *Mathematical Logic.* Harvard University Press, revised edition, 1981.

[11] B. C. Smith. Reflection and semantics in Lisp. In *Proc. Symposium on Principles of Programming Languages*, pages 23–35. ACM, Jan. 1984.

[12] G. L. Steele Jr., Jan. 1982. Electronic mail message.

[13] G. L. Steele Jr. *Common LISP: The Language.* Digital Press, second edition, 1990.

[14] G. L. Steele Jr. and G. J. Sussman. The revised report on SCHEME: A dialect of Lisp. Memo 452, MIT AI Lab, Jan. 1978.

[15] D. Weinreb and D. Moon. *Lisp Machine Manual.* Symbolics Inc., July 1981.

A Expansion Algorithm

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

I 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 must supply the following four functions:

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

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

`tag-comma-atsign?` This predicate should 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`. 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. But care must be taken not to perform any optimizations that alter the behavior of nested splicing.

A properly optimizing quasiquotation expander for Common Lisp can be found in [13, Appendix C]. I am not aware of the existence of a correct optimizing quasiquotation expander for Scheme. (None of the Scheme implementations that I tested implement nested splicing correctly.)