Self-Reproducing Programs in Common Lisp

Peter Norvig
Computer Science Division, University of California
Berkeley, CA 94720
norvig@teak.berkeley.edu

May 7, 1990

This paper reviews the classic self-reproducing expressions in Lisp, and presents some new ones that are unique to Common Lisp.

The Classic Self-Evaluating Expressions

The search for self-reproducing programs—programs that print their own sources—is a common exercise going back at least to [Bratley and Millo, 1972]. In Lisp, there is no such notion as a program *per se*, so the exercise is instead to find self-reproducing or self-evaluating expressions. Of course, there are many trivial self-evaluating atoms, such as these two:

```
t
2
```

Thus, it is traditional to limit the quest to non-atomic expressions. It is well-known that the following expression fits the bill, and as it uses only the most basic primitives, it will work in any dialect of Lisp:

```
((lambda (x) (list x (list (quote quote) x)))
 (quote (lambda (x) (list x (list (quote quote) x)))))
```

There are a few interesting variations on this theme. In a modern Lisp with backquote notation, a more succinct version is possible:

```
((lambda (x) (list x `',x)) '(lambda (x) (list x `',x)))
```

In Common Lisp (but not Scheme) it is possible to write an equivalent but more obtuse version:

```
((lambda (list) (list list `',list))
 '(lambda (list) (list list `',list)))
```

Conversely, in a Scheme where the printed representation of a function is the source code of the function, we can simply say:

```
((lambda (x) (list x x))
 (lambda (x) (list x x)))
```

This version is also self-reproducing in Common Lisp if you install the following handy macro definition for `lambda`,[1] and if your Common Lisp prints macro expansions as the original source code.

```
(defmacro lambda (args &body body)
  "Allow (lambda (x) ...) instead of #'(lambda (x) ...)"
  `#'(lambda ,args .,body))
```

Jon L White has suggested that the self-reference implicit in these self-evaluating expressions is reminiscent of the self-reference done by the Y combinator. The Y combinator is what one needs to add to the lambda calculus to allow recursion (see [Field and Harrison, 1988, p. 133]). In a normal-order reduction calculus with Scheme syntax, we can write Y as:

```
(define (Y f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

[1] If you hate those unsightly `#'` marks as much as I do, you'll use this macro even when you aren't playing with self-evaluating expressions.

Indeed, the body of Y looks just like the canonical self-evaluation expression with `list` replaced by `f`, and with an extra function call thrown in. We can show that this is a proper definition of Y with the following proof of the identity (Y f) = (f (Y f)):

```
(Y f) = ((lambda (x) (f (x x)))
         (lambda (x) (f (x x))))
      = (f ((lambda (x) (f (x x)))
            (lambda (x) (f (x x)))))
      = (f (Y f))
```

The key to this derivation is that (Y f) reproduces itself, along with an additional call to f. Our self-evaluating expression reproduces itself in the same way, but doesn't add an additional call. However, we can show that

```
(Y identity) = (identity (Y identity)) = (Y identity)
```

So `(Y identity)` is self-evaluating in the normal-order reduction calculus, although in an applicative-order language like Lisp it results in infinite recursion (see [Gabriel, 1988] for a discussion of applicative Y).

Some New Self-Evaluating Expressions

Let's return to the main point of this article: novel self-evaluating expressions. Once the door is opened to the full lexical conventions of Common Lisp, some very succinct new solutions are possible. Consider:

```
#1='#1#
```

This is the list whose first element is the symbol `quote` and whose second element is the list itself. While the expression is certainly self-*evaluating,* it does have the drawback that it is only self-*reproducing* in an environment where `*print-circle*` is non-nil. That restriction is lifted with the following version:

```
#1=(setq *print-circle* '#1#)
```

Those who complain that the goal should be to find a self-evaluating function call can embed these solutions in lambda expressions:

```
#1=((lambda () '#1#))
#1=((lambda () (setq *print-circle* '#1#)))
```

An alternate approach makes use of the oft-forgotten variable -, which in Common Lisp is bound to the current input to the read-eval-print loop (just as * is bound to the previous result). Thus, the following two expressions are self-reproducing when typed to a read-eval-print loop:

```
-
(identity -)
```

The first of these is the only non-constant atomic expression that is guaranteed to be self-evaluating, while the second is of course non-atomic.

As an aside, the following is one of the shortest infinite looping expression:

```
(eval -)
```

Richard Fateman provided a few more short infinite looping expression:

```
(loop)
#1=(progn #1#)
```

In fact, an infinite loop results when `progn` is replaced by any function, or by `multiple-value-call`, `multiple-value-prog1`, `tagbody`, `assert`, `unwind-protect`, `case`, `prog1`, `prog2`, `do`, `and`, `or`, `when` or `unless`.

In summary, it has been assumed that the best way to write a self-reproducing program is to bind a variable to part of the program, and then output that variable twice, along with enough "glue" to comprise the rest of the program. This short paper shows that it is also possible to get the same results by circular reference using the Y combinator or Common Lisp's unique `#1=` and `-` conventions.

References

[1] Bratley, P. and Millo, J. Computer Recreations. Self-reproducing automata. *Software Practice and Experience*, 2, (1972) 397-400.

[2] Field, A.J. and Harrison, P.G. *Functional Programming*, Addison-Wesley, (1988).

[3] Gabriel, R. The Why of Y. *Lisp Pointers*, 2 **2**, (1988) 15-25.

Once upon a time, in a kingdom not far from here, a king summoned two of his advisors for a test. He showed them both a shiny metal box with two slots in the top, a control knob, and a lever. "What do you think this is?"

One advisor, an engineer, answered first. "It is a toaster," he said. The king asked, "How would you design an embedded computer for it?" The engineer replied, "Using a four-bit microcontroller, I would write a simple program that reads the darkness knob and quantizes its position to one of 16 shades of darkness, from snow white to coal black. The program would use that darkness level as the index to a 16-element table of initial timer values. Then it would turn on the heating elements and start the timer with the initial value selected from the table. At the end of the time delay, it would turn off the heat and pop up the toast. Come back next week, and I'll show you a working prototype."

The second advisor, a computer scientist, immediately recognized the danger of such short-sighted thinking. He said, "Toasters don't just turn bread into toast, they are also used to warm frozen waffles. What you see before you is really a breakfast food cooker. As the subjects of your kingdom become more sophisticated, they will demand more capabilities. They will need a breakfast food cooker that can also cook sausage, fry bacon, and make scrambled eggs. A toaster that only makes toast will soon be obsolete. If we don't look to the future, we will have to completely redesign the toaster in just a few years."

"With this in mind, we can formulate a more intelligent solution to the problem. First, create a class of breakfast foods. Specialize this class into subclasses: grains, pork, and poultry. The specialization process should be repeated with grains divided into toast, muffins, pancakes, and waffles; pork divided into sausage, links, and bacon; and poultry divided into scrambled eggs, hard-boiled eggs, poached eggs, fried eggs, and various omelet classes."

"The ham and cheese omelet class is worth special attention because it must inherit characteristics from the pork, dairy, and poultry classes. Thus, we see that the problem cannot be properly solved without multiple inheritance. At run time, the program must create

(CONTINUED)