[This is a brief tutorial on ARC*. It's intended for readers with little programming experience and no* LISP *experience. It is thus also an introduction to* LISP*.]*

ARC programs consist of expressions. The simplest expressions are things like numbers and strings, which evaluate to themselves.

```
arc> 25
25
arc> "foo"
"foo"
```

Several expressions enclosed within parentheses are also an expression. These are called lists. When a list is evaluated, the elements are evaluated from left to right, and the value of the first (presumably a function) is passed the values of the rest. Whatever it returns is returned as the value of the expression.

```
arc> (+ 1 2)
3
```

Here's what just happened. First `+`, `1`, and `2` were evaluated, returning the plus function, 1, and 2 respectively. 1 and 2 were then passed to the plus function, which returned 3, which was returned as the value of the whole expression.

(Macros introduce a twist, because they transform lists before they're evaluated. We'll get to them later.)

Since expression and evaluation are both defined recursively, programs can be as complex as you want:

```
arc> (+ (+ 1 2) (+ 3 (+ 4 5)))
15
```

Putting the `+` before the numbers looks odd when you're used to writing `1 + 2`, but it has the advantage that `+` can now take any number of arguments, not just two:

```
arc> (+)
0
arc> (+ 1)
1
arc> (+ 1 2)
3
arc> (+ 1 2 3)
6
```

This turns out to be a convenient property, especially when generating code, which is a common thing to do in LISP.

LISP dialects like ARC have a data type most languages don't: *symbols.* We've already seen one: `+` is a symbol. Symbols don't evaluate to themselves the way numbers and strings do. They return whatever value they've been assigned.

If we give `foo` the value 13, it will return 13 when evaluated:

```
arc> (= foo 13)
13
arc> foo
13
```

You can turn off evaluation by putting a single quote character before an expression. So `'foo` returns the symbol `foo`.

```
arc> 'foo
foo
```

Particularly observant readers may be wondering how we got away with using `foo` as the first argument to `=`. If the arguments are evaluated left to right, why didn't this cause an error when foo was evaluated? There are some operators that violate the usual evaluation rule, and `=` is one of them. Its first argument isn't evaluated.

If you quote a list, you get back the list itself.

```
arc> (+ 1 2)
3
arc> '(+ 1 2)
(+ 1 2)
```

The first expression returns the number 3. The second, because it was quoted, returns a list consisting of the symbol `+` and the numbers 1 and 2.

You can build up lists with `cons`, which returns a list with a new element on the front:

```
arc> (cons 'f '(a b))
(f a b)
```

It doesn't change the original list:

```
arc> (= x '(a b))
(a b)
arc> (cons 'f x)
(f a b)
arc> x
(a b)
```

The empty list is represented by the symbol `nil`, which is defined to evaluate to itself. So to make a list of one element you say:

```
arc> (cons 'a nil)
(a)
```

You can take lists apart with `car` and `cdr`, which return the first element and everything but the first element respectively:

```
arc> (car '(a b c))
a
arc> (cdr '(a b c))
(b c)
```

To create a list with many elements use `list`, which does a series of `cons`es:

```
arc> (list 'a 1 "foo" '(b))
(a 1 "foo" (b))
arc> (cons 'a (cons 1 (cons "foo" (cons '(b) nil))))
(a 1 "foo" (b))
```

Notice that lists can contain elements of any type.

There are 4 parentheses at the end of that call to `cons`. How do LISP programmers deal with this? They don't. You could add or subtract a right paren from that expression and most wouldn't notice. LISP programmers don't count parens. They read code by indentation, not parens, and when writing code they let the editor match parens (use `:set sm` in *vi*, `M-x lisp-mode` in *Emacs*).

Like COMMON LISP assignment, ARC's `=` is not just for variables, but can reach inside structures. So you can use it to modify lists:

```
arc> x
(a b)
arc> (= (car x) 'z)
z
arc> x
(z b)
```

Lists are useful in exploratory programming because they're so flexible. You don't have to commit in advance to exactly what a list represents. For example, you can use a list of two numbers to represent a point on a plane. Some would think it more proper to define a point object with two fields, x and y. But if you use lists to represent points, then when you expand your program to deal with n dimensions, all you have to do is make the new code default to zero for missing coordinates, and any remaining planar code will continue to work.

Or if you decide to expand in another direction and allow partially evaluated points, you can start using symbols representing variables as components of points, and once again, all the existing code will continue to work.

In exploratory programming, it's as important to avoid premature specification as premature optimization.

The most exciting thing lists can represent is code. The lists you build with `cons` are the same things programs are made out of. This means you can write programs that write programs. The usual way to do this is with something called a *macro*. We'll get to those later. First, functions.

We've already seen some functions: `+`, `cons`, `car`, and `cdr`. You can define new ones with `def`, which takes a symbol to use as the name, a list of symbols representing the parameters, and then zero or more expressions called the body. When the function is called, those expressions will be evaluated in order with the symbols in the body temporarily set ("bound") to the corresponding argument. Whatever the last expression returns will be returned as the value of the call.

Here's a function that takes two numbers and returns their average:

```
arc> (def average (x y)
       (/ (+ x y) 2))
#<procedure: average>
arc> (average 2 4)
3
```

The body of the function consists of one expression, `(/ (+ x y) 2)`. It's common for functions to consist of one expression; in purely functional code (code with no side-effects) they always do.

Notice that `def`, like `=`, doesn't evaluate all its arguments. It is another of those operators with its own evaluation rule.

What's the strange-looking object returned as the value of the `def` expression? That's what a function looks like. In ARC, as in most LISPs, functions are a data type, just like numbers or strings.

As the literal representation of a string is a series of characters surrounded by double quotes, the literal representation of a function is a list consisting of the symbol `fn`, followed by its parameters, followed by its body. So you could represent a function to return the average of two numbers as:

```
arc> (fn (x y) (/ (+ x y) 2))
#<procedure>
```

There's nothing semantically special about named functions as there is in some other languages. All `def` does is basically this:

```
arc> (= average (fn (x y) (/ (+ x y) 2)))
#<procedure: average>
```

And of course you can use a literal function wherever you could use a symbol whose value is one, e. g.

```
arc> ((fn (x y) (/ (+ x y) 2)) 2 4)
3
```

This expression has three elements, `(fn (x y) (/ (+ x y) 2))`, which yields a function that returns averages, and the numbers 2 and 4. So when you evaluate all three expressions and pass the values of the second and third to the value of the first, you pass 2 and 4 to a function that returns averages, and the result is 3.

There's one thing you can't do with functions that you can do with data types like symbols and strings: you can't print them out in a way that could be read back in. The reason is that the function could be a *closure*; displaying closures is a tricky problem.

In ARC, data structures can be used wherever functions are, and they behave as functions from indices to whatever's stored there. So to get the first element of a string you say:

```
arc> ("foo" 0)
#\f
```

That return value is what a literal character looks like, incidentally.

Expressions with data structures in functional position also work as the first argument to =.

```
arc> (= s "foo")
"foo"
arc> (= (s 0) #\m)
#\m
arc> s
"moo"
```

There are two commonly used operators for establishing temporary variables, `let` and `with`. The first is for just one variable.

```
arc> (let x 1
       (+ x (* x 2)))
3
```

To bind multiple variables, use `with`.

```
arc> (with (x 3 y 4)
       (sqrt (+ (expt x 2) (expt y 2))))
5
```

So far we've only had things printed out implicity as a result of evaluating them. The standard way to print things out in the middle of evaluation is with `pr` or `prn`. They take multiple arguments and print them in order; `prn` also prints a newline at the end. Here's a variant of `average` that tells us what its arguments were:

```
arc> (def average (x y)
       (prn "my arguments were: " (list x y))
       (/ (+ x y) 2))
*** redefining average
#<procedure: average>
arc> (average 100 200)
my arguments were: (100 200)
150
```

The standard conditional operator is `if`. Like `=` and `def`, it doesn't evaluate all its arguments. When given three arguments, it evaluates the first, and if that returns true, it returns the value of the second, otherwise the value of the third:

```
arc> (if (odd 1) 'a 'b)
a
arc> (if (odd 2) 'a 'b)
b
```

Returning true means returning anything except `nil`. `Nil` is conventionally used to represent falsity as well as the empty list. The symbol `t` (which like `nil` evaluates to itself) is often used to represent truth, but any value other than `nil` would serve just as well.

```
arc> (odd 1)
t
arc> (odd 2)
nil
```

It sometimes causes confusion to use the same thing for falsity and the empty list, but many years of Lisp programming have convinced me it's a net win, because the empty list is set-theoretic false, and many Lisp programs think in sets.

If the third argument is missing it defaults to `nil`.

```
arc> (if (odd 2) 'a)
nil
```

An `if` with more than three arguments is equivalent to a nested `if`.

```
(if a b c d e)
```

is equivalent to

```
(if a
    b
    (if c
        d
        e))
```

If you're used to languages with `elseif`, this pattern will be familiar.[1]

Each argument to `if` is a single expression, so if you want to do multiple things depending on the result of a test, combine them into one expression with `do`.

```
arc> (do (prn "hello")
         (+ 2 3))
hello
5
```

[1]Note to Lisp hackers: If you're used to the conventional Lisp `cond` operator, this `if` amounts to the same thing, but with fewer parentheses. E. g. `(cond (a b) (c d) (t e))` becomes `(if a b c d e)`.

JMC's original `cond` didn't have implicit `progn`, so the parens around each pair of clauses were unnecessary. They became necessary soon after, however, when `cond` started to have implicit `progn` in the first Lisp implementations. This probably prevented people from realizing they hadn't originally been needed. But most `cond`s in the wild seem to occur in purely functional code, and thus pay the cost in parens of implicit `progn` without actually needing it. My experience so far suggests it's a net win to offer `progn` à la carte instead of combining it with the default conditional operator. Having to use explicit `do`s may even be an advantage, because it calls attention to nonfunctional code.

If you just want several expressions to be evaluated when some condition is true, you could say

```
(if a
    (do b
        c))
```

but this situation is so common there's a separate operator for it.

```
(when a
  b
  c)
```

The `and` and `or` operators are like conditionals because they don't evaluate more arguments than they have to.

```
arc> (and nil
          (pr "you'll never see this"))
nil
```

The negation operator is called `no`, a name that also works when talking about `nil` as the empty list. Here's a function to return the length of a list:

```
arc> (def mylen (xs)
       (if (no xs)
           0
           (+ 1 (mylen (cdr xs)))))
#<procedure: mylen>
```

If the list is `nil` the function will immediately return 0. Otherwise it returns 1 more than the length of the `cdr` of the list.

```
arc> (mylen nil)
0
arc> (mylen '(a b))
2
```

I called it `mylen` because there's already a function called `len` for this. You're welcome to redefine Arc functions, but redefining `len` this way might break code that depended on it, because `len` works on more than lists.

The standard comparison operator is `is`, which returns true if its arguments are identical or, if strings, have the same characters.

```
arc> (is 'a 'a)
t
arc> (is "foo" "foo")
t
arc> (let x (list 'a)
       (is x x))
t
arc> (is (list 'a) (list 'a))
nil
```

Note that `is` returns false for two lists with the same elements. There's another operator for that, `iso` (from isomorphic).

```
arc> (iso (list 'a) (list 'a))
t
```

If you want to test whether something is one of several alternatives, you could say `(or (is x y) (is x z) ...)`, but this situation is common enough that there's an operator for it.

```
arc> (let x 'a
       (in x 'a 'b 'c))
t
```

The `case` operator takes alternating keys and expressions and returns the value of the expression after the key that matches. You can supply a final expression as the default.

```
arc> (def translate (sym)
       (case sym
         apple 'mela
         onion 'cipolla
               'che?))
#<procedure: translate>
arc> (translate 'apple)
mela
arc> (translate 'syzygy)
che?
```

Arc has a variety of iteration operators. For a range of numbers, use `for`.

```
arc> (for i 1 10
       (pr i " "))
1 2 3 4 5 6 7 8 9 10 nil
```

To iterate through the elements of a list or string, use `each`.

```
arc> (each x '(a b c d e)
       (pr x " "))
a b c d e nil
```

Those `nils` you see at the end each time are not printed out by the code in the loop. They're the return values of the iteration expressions.

To continue iterating while some condition is true, use `while`.

```
arc> (let x 10
       (while (> x 5)
         (= x (- x 1))
         (pr x)))
98765nil
```

There's also a more general `loop` operator that's similar to the C `for` operator and tends to be rarely used in practice, and a simple `repeat` operator for doing something n times:

```
arc> (repeat 5 (pr "la "))
la la la la la nil
```

The `map` function takes a function and a list and returns the result of applying the function to successive elements.

```
arc> (map (fn (x) (+ x 10)) '(1 2 3))
(11 12 13 . nil)
```

Actually it can take any number of sequences, and keeps going till the shortest runs out:

```
arc> (map + '(1 2 3 4) '(100 200 300))
(101 202 303)
```

Since functions of one argument are so often used in LISP programs, ARC has a special notation for them. `[... _ ...]` is an abbreviation for `(fn (_) (... _ ...))`. So our first `map` example could have been written

```
arc> (map [+ _ 10] '(1 2 3))
(11 12 13 . nil)
```

Removing variables is a particularly good way to make programs shorter. An unnecessary variable increases the conceptual load of a program by more than just what it adds to the length.

You can compose functions by putting a colon between the names. I. e. `(foo:bar x y)` is equivalent to `(foo (bar x y))`. Composed functions are convenient as arguments.

```
arc> (map odd:car '((1 2) (4 5) (7 9)))
(t nil t)
```

You can also negate a function by putting a tilde (~) before the name:

```
arc> (map ~odd '(1 2 3 4 5))
(nil t nil t nil)
```

There are a number of functions like `map` that apply functions to successive elements of a sequence. The most commonly used is `keep`, which returns the elements satisfying some test.

```
arc> (keep odd '(1 2 3 4 5 6 7))
(1 3 5 7)
```

Others include `rem`, which does the opposite of `keep`; `all`, which returns true if the function is true of every element; `some`, which returns true if the function is true of any element; `pos`, which returns the position of the first element for which the function returns true; and `trues`, which returns a list of all the non-`nil` return values:

```
arc> (rem odd '(1 2 3 4 5 6))
(2 4 6)
arc> (all odd '(1 3 5 7))
t
arc> (some even '(1 3 5 7))
nil
arc> (pos even '(1 2 3 4 5))
1
arc> (trues [if (odd _) (+ _ 10)]
            '(1 2 3 4 5))
(11 13 15)
```

If functions like this are given a first argument that isn't a function, it's treated like a function that tests for equality to that:

```
arc> (rem 'a '(a b a c u s))
(b c u s)
```

and they all work on strings as well as lists.

```
arc> (rem #\a "abacus")
"bcus"
```

Lists can be used to represent a wide variety of data structures, but if you want to store key/value pairs efficiently, Arc also has hash tables.

```
arc> (= airports (table))
#hash()
arc> (= (airports "Boston") 'bos)
bos
```

If you want to create a hash table filled with values, you can use `listtab`, which takes a list of key/value pairs and returns the corresponding hash table.

```
arc> (let h (listtab '((x 1) (y 2)))
       (h 'y))
2
```

There's also an abbreviated form where you don't need to group the arguments or quote the keys.

```
arc> (let h (obj x 1 y 2)
       (h 'y))
2
```

Like lists and strings, hash tables can be used wherever functions are.

```
arc> (= codes (obj "Boston" 'bos "San Francisco" 'sfo "Paris" 'cdg))
#hash(("Boston" . bos) ("Paris" . cdg) ("San Francisco" . sfo))
arc> (map codes '("Paris" "Boston" "San Francisco"))
(cdg bos sfo)
```

The function `keys` returns the keys in a hash table, and `vals` returns the values.

```
arc> (keys codes)
("Boston" "Paris" "San Francisco")
```

There is a function called `maptable` for hash tables that is like `map` for lists, except that it returns the original table instead of a new one.

```
arc> (maptable (fn (k v) (prn v " " k))
               codes)
sfo San Francisco
cdg Paris
bos Boston
#hash(("Boston" . bos) ("Paris" . cdg) ("San Francisco" . sfo))
```

[Note: Like functions, hash tables can't be printed out in a way that can be read back in. We hope to fix that though.]

There is a tradition in Lisp going back to McCarthy's 1960 paper[2] of using lists to represent key/value pairs:

```
arc> (= codes (("Boston" bos) ("Paris" cdg) ("San Francisco" sfo)))
(("Boston" bos) ("Paris" cdg) ("San Francisco" sfo))
```

This is called an association list, or *alist* for short. I once thought alists were just a hack, but there are many things you can do with them that you can't do with hash tables, including sort them, build them up incrementally in recursive functions, have several that share the same tail, and preserve old values.

The function `alref` returns the first value corresponding to a key in an alist:

```
arc> (alref codes "Boston")
bos
```

There are a couple operators for building strings. The most general is `string`, which takes any number of arguments and mushes them into a string:

[2] *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*, CACM, April 1960. `http://www-formal.stanford.edu/jmc/recursive/recursive.html`

```
arc> (string 99 " bottles of " 'bee #\r)
"99 bottles of beer"
```

Every argument will appear as it would look if printed out by `pr`, except `nil`, which is ignored.

There's also `tostring`, which is like `do` except any output generated during the evaluation of its body is sent to a string, which is returned as the value of the whole expression.

```
arc> (tostring
       (prn "domesday")
       (prn "book"))
"domesday\nbook\n"
```

You can find the types of things using `type`, and convert them to new types using `coerce`.

```
arc> (map type (list 'foo 23 23.5 '(a) nil car "foo" #\a))
(sym int num cons sym fn string char)
arc> (coerce #\A 'int)
65
arc> (coerce "foo" 'cons)
(#\f #\o #\o)
arc> (coerce "99" 'int)
99
arc> (coerce "99" 'int 16)
153
```

The `push` and `pop` operators treat list as stacks, pushing a new element on the front and popping one off respectively.

```
arc> (= x '(c a b))
(c a b)
arc> (pop x)
c
arc> x
(a b)
arc> (push 'f x)
(f a b)
arc> x
(f a b)
```

Like `=`, they work within structures, not just on variables.

```
arc> (push 'l (cdr x))
(l a b)
arc> x
(f l a b)
```

To increment or decrement use `++` or `--`:

```
arc> (let x '(1 2 3)
       (++ (car x))
       x)
(2 2 3)
```

There's also a more general operator called `zap` that changes something to the result any function returns when applied to it. I. e. `(++ x)` is equivalent to `(zap [+ _ 1] x)`.

The `sort` function returns a copy of a sequence sorted according to the function given as the first argument.

```
arc> (sort < '(2 9 3 7 5 1))
(1 2 3 5 7 9)
```

It doesn't change the original, so if you want to sort the value of a particular variable (or place within a structure), use `zap`:

```
arc> (= x '(2 9 3 7 5 1))
(2 9 3 7 5 1)
arc> (zap [sort < _] x)
(1 2 3 5 7 9)
arc> x
(1 2 3 5 7 9)
```

If you want to modify a sorted list by inserting a new element at the right place, use `insort`:

```
arc> (insort < 4 x)
(1 2 3 4 5 7 9)
arc> x
(1 2 3 4 5 7 9)
```

In practice the things one needs to sort are rarely just lists of numbers. More often you'll need to sort things according to some property other than their value, e. g.

```
arc> (sort (fn (x y) (< (len x) (len y)))
           '("orange" "pea" "apricot" "apple"))
("pea" "apple" "orange" "apricot")
```

Arc's sort is stable, meaning the relative positions of elements judged equal by the comparison function won't change:

```
arc> (sort (fn (x y) (< (len x) (len y)))
           '("aa" "bb" "cc"))
("aa" "bb" "cc")
```

Since comparison functions other than `>` or `<` are so often needed, Arc has a `compare` function to build them:

```
arc> (sort (compare < len)
           '("orange" "pea" "apricot" "apple"))
("pea" "apple" "orange" "apricot")
```

We've seen several functions so far that take optional arguments or varying numbers of arguments. To make a parameter optional, just say `(o x)` instead of `x`. Optional parameters default to `nil`.

```
arc> (def greet (name (o punc))
       (string "hello " name punc))
#<procedure: greet>
arc> (greet 'joe)
"hello joe"
arc> (greet 'joe #\!)
"hello joe!"
```

Functions can have as many optional parameters as you want, but they have to come at the end of the parameter list.

If you put an expression after the name of an optional parameter, it will be evaluated if necessary to produce a default value. The expression can refer to preceding parameters.

```
arc> (def greet (name (o punc (case name who #\? #\!)))
       (string "hello " name punc))
*** redefining greet
#<procedure: greet>
arc> (greet 'who)
"hello who?"
```

To make a function that takes any number of arguments, put a period and a space before the last parameter, and it will get bound to a list of the values of all the remaining arguments:

```
arc> (def foo (x y . z)
       (list x y z))
#<procedure: foo>
arc> (foo (+ 1 2) (+ 3 4) (+ 5 6) (+ 7 8))
(3 7 (11 15))
```

This type of parameter is called a "rest parameter" because it gets the rest of the arguments. If you want all the arguments to a function to be collected in one parameter, just use it in place of the whole parameter list.

(These conventions are not as random as they seem. The parameter list mirrors the form of the arguments, and a list terminated by something other than `nil` is represented as e. g. `(a b . c)`.)

To supply a list of arguments to a function, use `apply`:

```
arc> (apply + '(1 2 3))
6
```

Now that we have rest parameters and `apply`, we can write a version of `average` that takes any number of arguments.

```
arc> (def average args
       (/ (apply + args) (len args)))
#<procedure: average>
arc> (average 1 2 3)
2
```

We know enough now to start writing macros. Macros are basically functions that generate code. Of course, generating code is easy; just call `list`.

```
arc> (list '+ 1 2)
(+ 1 2)
```

What macros offer is a way of getting code generated this way into your programs. Here's a (rather stupid) macro definition:

```
arc> (mac foo ()
       (list '+ 1 2))
*** redefining foo
#3(tagged mac #<procedure>)
```

Notice that a macro definition looks exactly like a function definition, but with `def` replaced by `mac`.

What this macro says is that whenever the expression (`foo`) occurs in your code, it shouldn't be evaluated in the normal way like a function call. Instead it should be replaced by the result of evaluating the body of the macro definition, `(list '+ 1 2)`. This is called the "expansion" of the macro call.

In other words, if you've defined `foo` as above, putting `(foo)` anywhere in your code is equivalent to putting `(+ 1 2)` there.

```
arc> (+ 10 (foo))
13
```

This is a rather useless macro, because it doesn't take any arguments. Here's a more useful one:

```
arc> (mac when (test . body)
       (list 'if test (cons 'do body)))
*** redefining when
#3(tagged mac #<procedure>)
```

We've just redefined the built-in `when` operator. That would ordinarily be an alarming idea, but fortunately the definition we supplied is the same as the one it already had.

```
arc> (when 1
       (pr "hello ")
       2)
hello 2
```

What the definition above says is that when you have to evaluate an expression whose first element is `when`, replace it by the result of applying

```
(fn (test . body)
  (list 'if test (cons 'do body)))
```

to the arguments. Let's try it by hand and see what we get.

```
arc> (apply (fn (test . body)
              (list 'if test (cons 'do body)))
            '(1 (pr "hello ") 2))
(if 1 (do (pr "hello ") 2))
```

So when ARC has to evaluate

```
(when 1
  (pr "hello ")
  2)
```

the macro we defined transforms that into

```
(if 1
    (do (pr "hello ")
        2))
```

first, and when that in turn is evaluated, it produces the behavior we saw above.

Building up expressions using calls to `list` and `cons` can get unwieldy, so most LISP dialects have an abbreviation called *backquote* that makes generating lists easier.

If you put a single open-quote character (`) before an expression, it turns off evaluation just like the ordinary quote (') does,

```
arc> `(a b c)
(a b c)
```

except that if you put a comma before an expression within the list, evaluation gets turned back on for that expression.

```
arc> (let x 2
       `(a ,x c))
(a 2 c)
```

A backquoted expression is like a quoted expression with holes in it.

You can also put a comma-at (`,@`) in front of anything within a backquoted expression, and in that case its value (which must be a list) will get spliced into whatever list you're currently in.

```
arc> (let x '(1 2)
       `(a ,@x c))
(a 1 2 c)
```

With backquote we can make the definition of `when` more readable.

```
(mac when (test . body)
  `(if ,test (do ,@body)))
```

In fact, this is the definition of `when` in the Arc source.

One of the keys to understanding macros is to remember that macro calls aren't function calls. Macro calls look like function calls. Macro definitions even look a lot like function definitions. But something fundamentally different is happening. You're transforming code, not evaluating it. Macros live in the land of the names, not the land of the things they refer to.

For example, consider this definition of `repeat`:

```
arc> (mac repeat (n . body)
       `(for x 1 ,n ,@body))
#3(tagged mac #<procedure>)
```

Looks like it works, right?

```
arc> (repeat 3 (pr "blub "))
blub blub blub nil
```

But if you use it in certain contexts, strange things happen.

```
arc> (let x "blub "
       (repeat 3 (pr x)))
123nil
```

We can see what's going wrong if we look at the expansion. The code above is equivalent to

```
(let x "blub "
  (for x 1 3 (pr x)))
```

Now the bug is obvious. The macro uses the variable `x` to hold the count while iterating, and that gets in the way of the `x` we're trying to print.

Some people worry unduly about this kind of bug. It caused the Scheme committee to adopt a plan for "hygienic" macros that was probably a mistake. It seems to me that the solution is not to encourage the noob illusion that macro calls are function calls. People writing macros need to remember that macros live in the land of names. Naturally in the land of names you have to worry about using the wrong names, just as in the land of values you have to remember not to use the wrong values—for example, not to use zero as a divisor.

The way to fix `repeat` is to use a symbol that couldn't occur in source code instead of `x`. In Arc you can get one by calling the function `uniq`. So the correct definition of `repeat` (and in fact the one in the Arc source) is

```
(mac repeat (n . body)
  `(for ,(uniq) 1 ,n ,@body))
```

If you need one or more `uniq`s for use in a macro, you can use `w/uniq`, which takes either a variable or list of variables you want bound to `uniq`s. Here's the definition of a variant of `do` called `do1` that's like `do` but returns the value of its first argument instead of the last (useful if you want to print a message after something happens, but return the something, not the message):

```
(mac do1 args
  (w/uniq g
    `(let ,g ,(car args)
       ,@(cdr args)
       ,g)))
```

Sometimes you actually want to "capture" variables, as it's called, in macro definitions. The following variant of `if`, which binds the variable `it` to the value of the test, turns out to be very useful:

```
(mac aif (expr . body)
  `(let it ,expr (if it ,@body)))
```

In a sense, you now know all about macros—in the same sense that, if you know the axioms in Euclid, you know all the theorems. A lot follows from these simple ideas, and it can take years to explore the territory they define. At least, it took me years. But it's a path worth following. Because macro calls can expand into further macro calls, you can generate massively complex expressions with them—code you would have had to write by hand otherwise. And yet programs built up out of layers of macros turn out to be very manageable. I wouldn't be surprised if some parts of my code go through 10 or 20 levels of macroexpansion before the compiler sees them, but I don't know, because I've never had to look.

One of the things you'll discover as you learn more about macros is how much day-to-day coding in other languages consists of manually generating macroexpansions. Conversely, one of the most important elements of learning to think like a Lisp programmer is to cultivate a dissatisfaction with repetitive code. When there are patterns in source code, the response should not be to enshrine them in a list of "best practices," or to find an IDE that can generate them. Patterns in your code mean you're doing something wrong. You should write the macro that will generate them and call that instead.

Now that you've learned the basics of Arc programming, the best way to learn more about the language is to try writing some programs in it. Here's how to write the hello-world of web apps:

```
arc> (defop hello req (pr "hello world"))
#<procedure:gs1430>
arc> (asv)
ready to serve port 8080
```

If you now go to `http://localhost:8080/hello` your new web app will be waiting for you.

Here are a couple slightly more complex hellos that hint at the convenience of macros that store closures on the server:

```
(defop hello2 req
  (w/link (pr "there")
    (pr "here")))

(defop hello3 req
  (w/link (w/link (pr "end")
            (pr "middle"))
    (pr "start")))

(defop hello4 req
  (aform [w/link (pr "you said: " (arg _ "foo"))
           (pr "click here")]
    (input "foo")
    (submit)))
```

See the sample application in `blog.arc` for ideas about how to make web apps that do more.

We now know enough Arc to read the definitions of some of the predefined functions. Here are a few of the simpler ones.

```
(def cadr (xs)
  (car (cdr xs)))

(def no (x)
  (is x nil))

(def list args
  args)

(def isa (x y)
  (is (type x) y))

(def firstn (n xs)
  (if (and (> n 0) xs)
      (cons (car xs) (firstn (- n 1) (cdr xs)))
      nil))

(def nthcdr (n xs)
  (if (> n 0)
      (nthcdr (- n 1) (cdr xs))
      xs))

(def tuples (xs (o n 2))
  (if (no xs)
      nil
      (cons (firstn n xs)
            (tuples (nthcdr n xs) n))))

(def trues (f seq)
  (rem nil (map f seq)))

(mac unless (test . body)
  `(if (no ,test) (do ,@body)))

(mac awhen (expr . body)
  `(let it ,expr (if it (do ,@body))))

(mac n-of (n expr)
  (w/uniq ga
    `(let ,ga nil
       (repeat ,n (push ,expr ,ga))
       (rev ,ga))))
```

These definitions are taken from `arc.arc`. As its name suggests, reading that file is a good way to learn more about both Arc and Arc programming techniques. Nothing in it is used before it's defined; it is an exercise in building the part of the language written in Arc up from the "axioms" defined in `ac.scm`. I hoped this would yield a simple language. But since this is also the source code of Arc, I've tried to balance simplicity with efficiency. The definitions aren't mathematically minimal if that would be insanely inefficient; I tried that once, and they were.

The definitions in `arc.arc` are also an experiment in another way. They are the language spec. The spec for `isa` isn't prose, like function specs in Common Lisp. This is the spec for `isa`:

```
(def isa (x y)
  (is (type x) y))
```

It may sound rather dubious to say that the only spec for something is its implementation. It sounds like the sort of thing one might say about C++, or the Common Lisp loop macro. But that's also how math works. If the implementation is sufficiently abstract, it starts to be a good idea to make specification and implementation identical.

I agree with Abelson and Sussman that programs should be written primarily for people to read rather than machines to execute. The Lisp defined as a model of computation in McCarthy's original paper was. It seems worth trying to preserve this as you grow Lisp into a language for everyday use.