INTERPRETERS FOR FUNCTIONAL PROGRAMMING

by

David S. Wise

Computer Science Department

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 111

INTERPRETERS FOR FUNCTIONAL PROGRAMMING

DAVID S. WISE

JULY, 1981

*To appear in Functional Programming and Its Applications (ed. J. Darlington), Cambridge University Press (1982).

This material is based upon work supported by the National Science Foundation under Grant MCS77-22325.

INTERPRETERS FOR FUNCTIONAL PROGRAMMING

DAVID S. WISE

Indiana University

The title of this tutorial is a bit of a pun. Not only do I present here a complete "program" _for_ evaluating a "program" in a functional language (a pure variant of LISP of McCarthy [1963]) but also I develop a complex program in a style typical _for_ such languages.

To absorb this material it is not sufficient merely to understand how the interpreter works; one must appreciate why the interpreter is written in just this way. Ease of writing _and_ comprehending is an important (and, to FORTRAN buffs, a strangely difficult) advantage of this style. Also important is the ease with which programs can be changed, as we shall see later when this interpreter is modified slightly with drastic changes to the semantics of the language. Worth mention (but not treated) is the ease of proving and of implementing such programs in various systems. All these points are important _for_ programming, but the last two are beyond this introduction.

0. INTRODUCTION

This four-part introduction to LISP is built around the LISP EVAL/APPLY interpreter. (An interpreter is a program that gives meaning to programs directly - without transforming source code in any way.) McCarthy [1960] presents such an operational implementation of the language in the first paper on LISP, it appears as the "Chapter I intepreter" by McCarthy _et al_, [1962], and has since become the easiest way to understand the language deeply. It has appeared in modified form in numerous appendices and manuals to clarify implementation or modification to the original language, so it is very much a part of LISP lore.

Aside from _completely_ describing a functional language, the interpreter developed in the first section here is significant because it demonstrates that both data and program are in the same memory as lists! A fundamental property of von Neumann

machines, that program and data are represented in the same store, reappears in LISP, having been forgotten in numerous "more sophisticated" programming schemes before and since. Indeed the plethora of LISP mutations facilitated by accessibility of "program" structures suggests that this convention has given LISP a vehicle for change that has kept it current and popular.

The interpreter evaluates an operator to a "closure", a function which includes static binding information. The second section shows how closures can be used in a data object to derive "streams" without altering the language. A stream may be pictured as a data structure which unfolds as it is traversed. The facility for streams allows applicative programming to handle not only ordinary input/output problems, but also internal structures specified to be infinite. One example is the list of _all_ prime numbers.

The third section deals with interpreter modifications, some motivated by stream operations. The interpreter of the first section is gently altered to facilitate some operations with pleasantly surprising (i.e. drastic) changes to LISP semantics. The ease with which these changes are made is testimony to the power of the applicative style and of interpretation.

Finally, the fourth section considers storage management. This is principally an argument that list structure is a viable elementary data representation, because dereferenced structures can be recycled automatically. Too many systems (e.g. PASCAL) depend on the user to return unused structure; this practice is neither dependable nor necessary. Reliable, efficient, automatic storage management is possible, particularly for the simple structures used by LISP.

Two kinds of garbage collection and reference-counting are reviewed, as well as hybrid schemes that use reference counting to postpone garbage collection. The demands of parallel processing architectures, an attractive application area for functional programming, are considered.

1. THE EVAL/APPLY INTERPRETER

This section builds up to a complete interpreter for LISP _in_ LISP. Reynolds [1972] describes this approach as "meta-circular," and discusses its utility and limitations. From the perspective of formal semantics this is dangerous since the least fixed-point for such an interpreter maps everything to $\perp$. What is wanted is a greater fixed-point, an operational semantics: if you understand the language then you'll

understand it. This interpreter wants an implemention in a lower-level language.

The explanation of the evaluator is built up through several subsections introducing syntax, environments, functions and closures, circularities, and then finally introducing eval and apply. Eval is the half of the interpreter that deals in special ways with the environment; its decisions deal with special forms or evaluation strategies that are not ordinary function invocations. Ordinarily, operands are evaluated first (called-by-value) and then passed with the function to apply. Most of apply's decisions deal with identifying primitive functions; applications of a user-defined function are handed back to eval after updating the environment with its parameter bindings.

1.1 Syntax

Before any semantics we need one simple piece of syntax: the LISP S-expression. An S-expression is either an atom (i.e. an ALGOL identifier or a number) or it is a (perhaps null) list of S-expressions. Atoms are written by their "name":

```
8 FAR 2 MUCH;
```

Lists are written enclosed by parentheses:

```
(AS ROUND BRACKETS (R) (OFT 10) CALLED).
 x = (A B C D).
```

One should not think of S-expressions as "little boxes", although such a picture can be useful later, especially when considering storage management. So Figure 1 shows x in "little boxes";

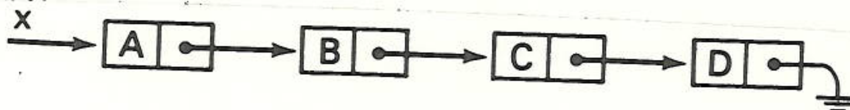

Figure 1: x bound to (A B C D)

Now forget that you saw it! All that matters is that x has a first element (or car), A, and a suffix (or cdr) that must be a list: (B C D). What x means depends entirely on how x is used. (Compare this with the meaning of a memory address depending upon which register it is in.) As data, x is a list of four elements; as an expression it is an application of an operator A to three operands B, C, and D. Thus (car x) evaluates to an operator and (cdr x) evaluates to the list of operands. Note that "(car x)" is not an operator; it's just another list; if we evaluate it in the right environment it should derive an operator. Cons is a dyadic operator which

prefixes (the value of) its first operand to the list that is (the value of) its second, yielding a list one item longer. McCarthy [1963] describes cons as a disjoint union operator, decomposed by car and cdr. (In Section 4 we shall acknowledge that it does allocate little boxes.)

Thus, LISP S-expressions provide atoms and type-free lists. If necessary, anything can be a truth value: NIL and () are false and everything else is true.

1.2 Environments

Evaluation of expressions (that aren't constants) requires some sort of environment. The environment is another kind of list structure which associates identifiers with values to which they have been bound during some function invocation that helped define that environment. In other words, it codes a map from names to values.

The outermost environment, extant at the top level of the interpreter, contains all global bindings. The interpreter given here assumes nothing of that environment -- it could be empty -- but it does embed much meaning inside the interpreter itself. Meaning implemented in the interpreter of course cannot be changed, but implementation on a specific site could include a local library by providing a rich outermost environment.

Since environments are never printed, it doesn't matter that they look ugly, but we must know their shape. An environment is a list of zero or more associations; an association is a list whose first element is a list of identifiers (atoms) and whose remainder is the list of values to which they have been bound. For example
E = (((Q R S) 1 2 3) ((P Q) () 37)) is an environment with Q bound to 1, R to 2, S to 3, and P to (). Through the second association of E, Q might have been bound to 37 except that leftmost associations take precedence. (Figure 2 illustrates the boxes.)

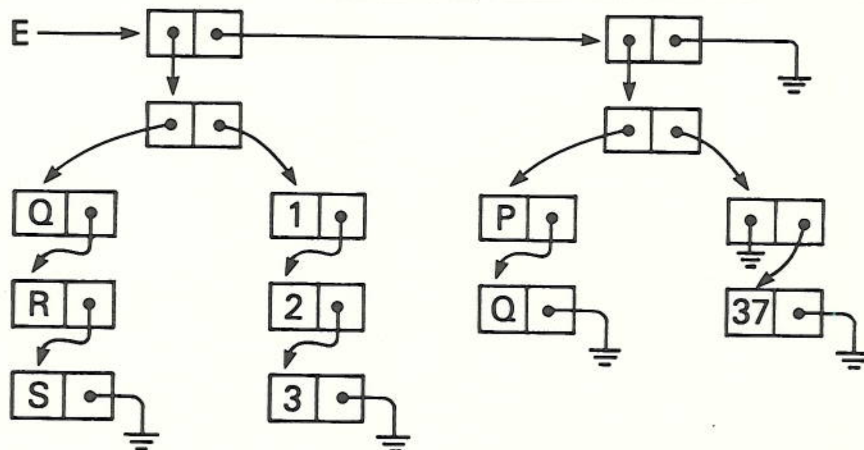

Figure 2. The Environment E

We build up such "ribcage" environments [Johnson, 1977] from older environments with formal parameter lists and actual (evaluated) parameter lists:

```
bind = (lambda  (parameters arguments  oldenvt)
                (cons (assct parameters arguments) oldenvt));
assct = cons.
```

(This lambda notation, inspired by λ-calculus, is explained in the next section.) In building environments we need only prefix the new set of bindings at the front of the old environment, where they take precedence. Given an identifier it will be necessary to look up its binding.

```
lookup = (lambda (ident envt) (if
         if     (null? envt) then (error-unbound)
         elseif (member? ident (top-idents! envt))
                 then (search ident (top-idents! envt)
                                    (top-vals! envt))
         else   (lookup ident (cdr envt)) ));

top-idents! = (lambda (envt) (car (car envt)));
top-vals!   = (lambda (envt) (cdr (car envt)));

search  = (lambda (ident params argums) (if
          if     (null? params) then (error-member)
          elseif (null? argums) then (error-too-few)
          elseif (eq? (car params) ident) then (car argums)
          else (search ident (cdr params)(cdr argums)) )).
```

Several stylistic conventions are followed in the above code. All conditionals have commenting keywords _if/then/else_ interspersed which are inserted purely for legibility; they are not present in actual code, so the stuttered "(if _if_" is really just "(if" at the top of each conditional. All predicates end with a question mark; _null_? and _eq_? are primitive but _member_? is a standard exercise. I use the _car_ and _cdr_ access primitives explicitly only as part of a recursion pattern or within probe functions which access one field of a "record". These probe functions have meaningful (field) names ending with an exclamation point and thereby help comment the program. The arguments to the conditional are paired on one line (forming an _if then_ pattern) with the last (_else_) standing alone. In the common case, as above, where the _else_ line directly reinvokes the function being defined, that definition exhibits _tail recursion_, a simple recursion pattern which is well known to be equivalent to a simple loop in iterative style programming. Another common _else_ line invokes some "error" notification scheme.

This code is not intended to be perfectly efficient (_Member_

and *search* could be combined as one traversal); it is intended to be definitive and clear. The interpreter written in this style is polished (but somewhat non-standard).

1.3 Functions and closures

The user writes an operator as a triple (list) whose first element is the keyword, *lambda*. We also may use an identifier bound to a function that results from evaluation of such a triple. Borrowed from λ-calculus, this triple has as its second item an identifier list, the formal parameters of the function, and its third is an expression that is the function body.

Other than local formal parameters can occur in the body. Typically non-local variables and function names can occur "free" there, as well. How to interpret free variables that are not immediately local parameters is a strange issue in LISP, which is here handled conventionally (except by LISP standards) using "static scoping". (This issue relates syntactically to *funarg* in ordinary LISPs.) This is effected by evaluating a *lambda* triple into a *closure* quadruple. The last two items in a quadruple are a formal parameter list and a body, just as in a *lambda* triple. The first thing is the word *closure*; the second is an environment. That environment will determine the meaning of all free variables in the body. Determining that environment is simple: evaluation of a *lambda* triple yields a *closure* quadruple which seizes the then local environment. Since such triples are evaluated [Steele and Sussman, 1976] when functional bindings are established, free variables can be resolved long before such a function is invoked. Closures may be returned as, or as part of, values resulting from function invocation to the caller. (Caveat: this is extraordinary for LISP!).

As functions, integers are projection functions. For example 3, as a function, returns its third argument; (3 39 9 33 3) evaluates to the integer 33. For subscripting of vectors one can use *apply*, defined below, to extract the i-th element in a list; e.g. if x is bound to (*A* *B* *C*) then (*apply* *3* *x*) returns *C*, and (*car* *fred*) is synonomous with (*apply* *1* *fred*).

1.4 Circular environments

While other, awkward schemes are directly available through λ-calculus, we need a simple and direct scheme for building circular environments. Such an environment might, for example, contain the bindings of operator names to their respective function closures (all closed in the one environment) so that one function could be referenced freely in the body of any other simply by using its name. This is necessary because we

want to use non-trivial (i.e. indirect) recursion patterns freely with the efficiency of explicit circularity.

We borrow the word letrec from Landin [1966] and introduce a quadruple associated with it; of course letrec is the first item in that list. The second and third are two lists of the same length: one of identifiers and one of expressions. The fourth item is a single expression whose evaluation yields the value of the evaluated letrec form. That (fourth) evaluation, however, proceeds in a very rich environment. That environment includes the bindings of all those identifiers (second) to the evaluation of all those expressions (third) -- the evaluations to proceed in this rich environment.

Terribly circular? Not quite. For now we will allow to the user only lambda triples in that third list, and they quickly evaluate to closures establishing only circular references -- not circular traversals. Later, however, we shall see how to relax this restriction.

For example the EVAL/APPLY interpreter which follows should ultimately appear as

```
(letrec (eval apply...)
        (defn-of-eval  defn-of-apply...)
        eval)
```

which allows eval to call apply -- using just that name -- and apply to call eval. Actually we can also include all "help functions", like lookup and bind, defined in this same environment and thereby hidden from the top level of the interpreter which receives the closure for eval from evaluation of this letrec, without seeing the binding of any names. Figure 3 is useful here.

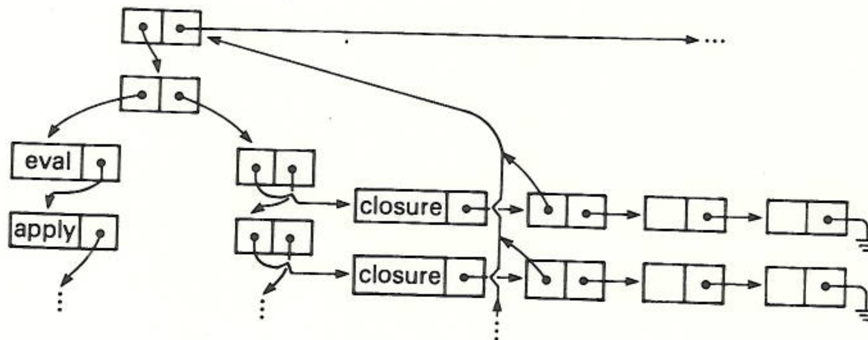

Figure 3. Circular Environment

1.5 Evaluation--EVAL

We now proceed to a fundamental call-by-value interpreter. An S-expression to be evaluated is either an atom or a list whose car is an operator, ordinarily evaluating to a function, and whose cdr is a list of operands that all must be evaluated to arguments before that function is computed -- as in conventional call-by-value protocol.

The ordinary case above is that when eval directly calls apply. There are exceptions however. Constants, such as integers, evaluate to themselves without further work. Identifiers are turned over to lookup for evaluation according to their bindings in the current environment. Lambda forms are closed quite quickly. Conditional expressions conventionally have their operands evaluated selectively in sequence: first a predicate and then a value if that returns "true"; otherwise another predicate or an else alternative. Evaluation of a letrec form also results in a second call to eval after building a circular environment.

One last exception allows evaluation to stop at values other than constants; we allow quoted values to stop evaluation by just "opening" the quotation. Thus (quote s-exp) always evaluates to s-exp for any LISP S-expression (i.e. value) you want. For legibility we use upper-case to denote quoted variables Thus (quote s-exp) and S-EXP are synonomous. Quotes are removed in eval.

```
eval = (lambda (exp env)(if
       if     (constant? exp) then exp
       elseif (atom? exp)     then (lookup exp env)
       elseif (quoted? exp)   then (dequote exp)
       elseif (eq? (oprtr! exp) IF)
          then (conditional (opnds! exp) env)
       elseif (eq? (oprtr! exp) LAMBDA)
          then (close (opnds! exp) env)
       elseif (eq? (oprtr! exp) LETREC)
          then (selfref (opnds! exp) env)
       else (fnapplication (eval# exp env)) ));

eval# = (lambda (explist env)(if
        if (null? explist) then ()
        else (cons (eval  (car explist) env)
                   (eval# (cdr explist) env)) )).
```

Eval# applies eval to every expression in a list, returning the list of values.

```
fnapplication = (lambda (evaluated) (apply(car evaluated)
                                          (cdr evaluated) ));
```

```
oprtr!   = car;
opnds!   = cdr;
selfref  = (lambda (suffix env) (eval (exp! suffix)
               (circular (idents! suffix)
                         (values! suffix) env) ));
exp!     = (lambda (triple)(apply 3 triple));
idents!  = (lambda (triple)(apply 1 triple));
values!  = (lambda (triple)(apply 2 triple));
circular = (lambda (formals opnds env)
               (letrec (newenv)
                       ((bind formals (eval# opnds newenv) env))
                newenv))
```

Circular is aptly named since it defines letrec with letrec (inside eval defined using eval.) It builds the circularities of Figure 3, where we assume that everything in explist is trivial (for now a constant, or quoted, or a lambda form) so eval# does no serious work. (In fact, in Section 1.2 we must change assct to strcons of Section 2.1 to make this code work.)

```
close = (lambda (fnpair env)(cons CLOSURE (cons env fnpair)));
conditional = (lambda (condpairs env)(if
              if     (null? condpairs) then nil
              elseif (singletonelse? condpairs)
                            then (eval (car condpairs) env)
              elseif (eval (predicate! condpairs)  env)
                            then (eval (value! condpairs) env)
              else (conditional (elsepart! condpairs) env)  ));
singletonelse? = (lambda (c) (null? (cdr c)));
predicate!     = car
value!         = (lambda (c) (car(cdr c)));
elsepart!      = (lambda (c) (cdr(cdr c)));
quoted?        = (lambda (e)(eq? (car e) QUOTE));
dequote        = (lambda (quoted) (apply 2 quoted)).
```

Recall that QUOTE really means (quote quote) in reading these definitions. We introduce one more primitive and an abbreviation for it. The form (list a b c) will evaluate to a list of the three values resulting from evaluation of a, b, and c; this generalizes to lists of any length. Its shorthand will be angle brackets: <a b c> is synonomous with (list a b c). Using both conventions <FRED 8 FISH> evaluates to (fred 8 fish) in any environment.

```
constant? = (lambda (e)(if
            if      (null? e) then TRUE
            elseif  (number? e) then TRUE
            elseif  (member? e <CAR CDR CONS EQ ATOM NULL LIST>,
                                  then TRUE
            else   nil)).
```

1.6 Evaluation -- APPLY

The function apply also receives just two arguments: a function and a list of arguments. Mostly it expends its effort dispatching primitives, but it also handles user-defined functions (received as closures). All constants are acceptable primitive functions.

```
apply = (lambda (fn args)(if
        if (null? fn) then ()
        elseif (atom? fn) then (if
             if      (number? fn)    then (index fn args)
             elseif (eq? fn CAR)    then (car   (frst! args))
             elseif (eq? fn CDR)    then (cdr   (frst! args))
             elseif (eq? fn CONS)   then (cons  (frst! args)
                                                (scnd! args))
             elseif (eq? fn EQ?)    then (eq?   (frst! args)
                                                (scnd! args))
             elseif (eq? fn ATOM?)  then (atom? (frst! args))
             elseif (eq? fn NULL?)  then (null? (frst! args))
             elseif (eq? fn LIST)   then args
             else   (errorprimitive))
        elseif (closure? fn)
             then(eval (body!  fn)
                       (bind (params! fn) args (env! fn))) ));
closure? = (lambda (fn) (eq? (car fn) CLOSURE));
body!    = (lambda (closure) (apply 4 closure));
params!  = (lambda (closure) (apply 3 closure));
env!     = (lambda (closure) (apply 2 closure));
frst!    = (lambda (lis) (apply 1 lis));
scnd!    = (lambda (lis) (apply 2 lis));
index    = (lambda (i vector) (if
              if i = 1 then (car vector)
              else (index (predecessor i) (cdr vector)) )).
```

Arithmetic is ignored here; it may either be implemented implicitly using internal representation of numbers or explicitly using Peano arithmetic with cons, cdr, and null?

2. STREAMS

Letrec and function closures are very powerful constructs. Taken together, as we shall see in this section, they allow the scope of functional programming to be expanded to include most conventional programs in the style we have been using above.

This section first develops cons into Landin's [1965] prefix operator by using closures to delay evaluation. Then it develops functional combination as a syntactic tool for dealing with the stream structure that arises from prefix*, here called

strcons. Several examples develop these ideas.

2.1 Cons for streams

As McCarthy [1963] defined LISP, all arguments must be evaluated before apply is invoked; specifically the code for CONS in apply indicates that its arguments are called-by-value. Static binding offers us an opportunity to change that convention. The following development is inspired by Henderson [1980].

Suppose we could define two operators, delay and force, that could be used together to postpone evaluation:

```
delay = (lambda (exp) (lambda () exp);
force = (lambda (closure) (closure)).
```

(This definition does not quite work for delay, as will be discussed below.) Delay is intended to receive an operand and turn it into a function of no arguments, without even inspecting the operand and using only the least time. That closure is a "recipe" for the value which is an object that can be passed around the system still unevaluated, or it can be forced as needed.

This gives the user a nice, but somewhat dangerous facility since he might delay work that is already delayed. Therefore, we shall restrict its use to the direct operands of cons, really the only mechanism at all in this system for establishing any binding (whether in user structure or in binded environments). Once delayed at the time binding is established, a value need never be delayed further.

Landin's approach is a version of cons which delays its second parameter. This he uses to build streams.

```
strcons = (lambda (a d) (cons a (delay d)));
head    = (lambda (stream) (car stream));
tail    = (lambda (stream) (force(cdr stream))).
```

Rather than giving the user force and delay, we give him strcons which generates a new "type"; therefore he must use head and tail to probe streams. We have axioms similar to McCarthy's [1963] for disjoint union:

```
(car (cons x y)) = x = (head (strcons x y));
(cdr (cons x y)) = y = (tail (strcons x y)).
```

(strcons x y) can exist even when evaluation of y doesn't converge. In other words, y is called-by-name by the composition tail·strcons, but not called by strcons alone.

Indeed if tail is never applied to a particular stream then

eval is never applied to that second argument of *strcons* and we save time. It could be invoked more than once as well (but see the next section.)

Unfortunately, both *delay* and *strcons* cannot be defined as user functions, because (under the interpreter in Section 1) they require that an operand be treated carefully. That is the job of *eval*, and user-defined functions are only treated in *apply* after *eval#* has evaluated all operands. Therefore, code involving *delay* and *strcons* cannot run properly because their operands would be evaluated too early through conventions of user functions. The solution is to envision both *delay* and, specifically, *strcons* as macros to be expanded before the program is run. Whenever we write

```
(strcons element lis)
```

such code will expand to

```
(cons element (lambda () lis))
```

before execution, for any operands *element* and *lis*. In Section 3.3 this need for macros is elided.

This gives us a mechanism for handling structures that are not "all there", such as an external file or a communication stream. Let us consider a "file" to be a stream of characters. Then a function which counts the number of times the letter "q" appears in a file is

```
countq =* (lambda (file) (count file 0));
count  =  (lambda (file cnt) (if
          if     (null? file) then cnt
          elseif (eq? (head file) Q)
                 then (count (tail file) (successor cnt))
          else        (count (tail file) cnt) )).
```

The use of *head* and *tail* in this file traversal, in place of *car* and *cdr* is important because it buffers the call-by-value semantics of LISP for the *file* stream. It is possible -- even appropriate -- that the suffix of this stream remain in secondary storage until *tail* forces it into main memory. As we shall see in the fourth section, once a file prefix is dereferenced, it is erased. Therefore, this program runs in

*More properly this definition appears as

```
    countq = (letrec (count)
                     ((lambda (file cnt)(if...) ))
               (lambda (file)(count file 0)))
```

but, following the style of *eval/apply*, I shall continue with the repeated = symbol and understand that *letrec* is really used to "wrap" recursive and mutually recursive function definitions.

time and space resource analogous to FORTRAN code: a small buffer and a loop (tail recursion) until end-of-file () is reached.

This behavior is typical of many file filtering utilities (e.g. text editiors.) The tail-recursive stream style is the feature in functional programming that provides the facility for solving such ordinary problems without the operational analogues of large buffers and deep recursion stacks.

2.2 Multiple streams and functional combination

Let us consider another example: a "decoder" of a file, which sends all "q"'s to one output file, and all other characters in sequence to another. This function has two results structured as a list -- i.e. an ordered pair of streams. Since its output is interpreted in two pieces, we hyphenate the name of the function to suggest the two results:

```
q-nonq = (lambda (file)(if
         if     (null? file) then <<><>>
         elseif (eq? (head file) Q)
                then <(strcons (head file)
                               (apply 1 (q-nonq(tail file))) )
                      (apply 2 (q-nonq (tail file))) >
         else        <(apply 1 (q-nonq (tail file)))
                      (strcons (head file)
                            (apply 2 (q-nonq(tail file))) )>
           )).
```

This is nearly unreadable, so before remarking on its behavior, we shall introduce "functional combination" to make it *look* pretty.

Let us extend the definition of function from primitive or lambda-form to include a list of functions, all of the same arity. Such a list is a combination of the functions [Friedman and Wise, 1978]. The number of arguments to such a *functional combination* must be equal to the arity of the functions and each element must evaluate to a list. The actual parameter list is perceived as a matrix, evaluated and passed from *eval* to *apply* in a row-major representation. The result of such an application is the list of the applications of the individual functions to the columns of that matrix; application is in column-major order. For example

```
                (< sum  product quotient  difference>
                 < -3     3        16         2      >
                 <  4     3         2         1      >)
evaluates  to    (  1     9         8         1      ),  if
```

arithmetic primitives are implemented. We shall see how this is implemented in Section 3.

Now we can use functional combination and very careful vertical alignment to express the same program for q-nonq

```
q-nonq = (lambda(file) (if
         if      (null? file) then     <    <>         <>  >
         elseif (eq?(head file)Q)then(<   strcons     2    >
                                      <(head file)    ZERO>
                                      (q-nonq (tail file)) )
         else (<  2      strcons    >
               <ZERO    (head file)>
               (q-nonq (tail file))) )).
```

Zero is merely a place-holder --like 0 in decimal arithmetic. Alternatively, if one can cope with conditionals in the function position:

```
q-nonq = (lambda (file) (if
         if   (null? file) then <<><>>
         else((if(eq?(head file)Q)<strcons 2><2 strcons>)
               <    (head file)        (head file)       >
               (     q-nonq            (tail file)       )))).
```

Either of these latter definitions is crisper than the first where the recursive call occurred repeatedly due to difficulties with call-by-value and delay protocol. All three read a stream until both a "q" and "non-q" character have been encountered, returning two streams starting with these characters.

A useful notation for use with functional combination is the asterisk to imply infinite homogeneous streams. Semantically <e*> is analogous to

```
(letrec (stream)
        ((strcons e stream))
 stream)
```

although we can implement it as in Figure 4, as well. Thus <0*> evaluates to the zero vector of infinite length.

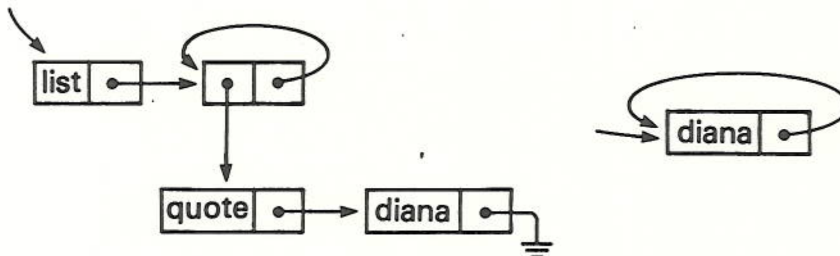

Figure 4. ⟨DIANA*⟩ and the Result of Evaluating it.

Adopting the convention that the result of a functional combination is bounded in width by the shortest of the functional form or any of its arguments, we find the asterisk notation can abbreviate functional combination further*. We could as well have used <(head file)*> as the second last line of the immediately preceeding example. Even more dramatically

```
eval# = (lambda(opnds env) (<eval*>
                             opnds
                            <env* > )).
```

2.3 Examples

Streams alone are sufficient to express certain infinite operations. For instance, infinite vectors are possible of which only a finite prefix can ever be used:

```
vectorsum = (lambda (vec1 vec2) (if
            if     (null vec1) then <>
            elseif (null vec2) then <>
            else (strcons (sum (head vec1) (head vec2))
                   (vectorsum (tail vec1) (tail vec2))) )).
```

With recursion we can even define the tail of a vector in terms of the vector, itself. This use of letrec works because streams are not manifested completely on evaluation; the effort to complete evaluation to a stream is constrained to evaluation of its head. Thus, we can define such a stream in a letrec as long as its head is defined independently of the stream; the delayed tail may be so defined, however. Because the tail is represented as a closure it can be defined circularly. This gives rise to a generative style of programming whereby a structure uses itself to define itself; data bindings as well as function bindings can be recursive.

```
naturals = (strcons 0 (vectorsum <1*> naturals))
```

evaluates to (0 1 2 3 4...) as far as one traverses using head and tail. Such recursions must be "seeded" explicitly with base values, directly analogous to bases of recursion or of inductive proofs of their correctness. Larger problems have complicated bases:

---

* There is much familiar about functional combination using asterisk notation. Backus [1978] uses α "apply to all", LISP uses MAPCAR, and APL uses nothing, all to expand functions over structures.

```
fibonacci = (strcons 1 (strcons 1 (vectorsum
                                   fibonacci
                                   (tail fibonacci))))
```

evaluates to (1 1 2 3 5 8...).

```
pascaltriangle = (letrec
                    (generate)
                    ((lambda(row)(strcons row
                        (generate(strcons 1 (vectorsum
                                               row
                                               (tail row)) )) )))
                  (generate (strcons 1 <0*>)) )
```

```
evaluates to    ( (1 0 0...)
                  (1 1 0 0...)
                  (1 2 1 0 0...)
                  (1 3 3 1 0 0...)
                   . . .          ).
```

More use of functional combination can make these definitions even shorter. For instance, vectorsum = <sum*>.

The Sieve of Eratosthenes is a classic and efficient generator of prime numbers. It uses only addition to cast out non-primes from an (ordinarily) bounded list of integers, leaving a residue of primes. The facility of programming using infinite streams, however, allows a sieve to be written which sifts all primes from the stream of all integers (greater than one.) No bound is necessary. The idea of an open ended sieve is due to Quarendon [Henderson, 1980].

The next program, a sieve, illustrates a generator which actually "grows". The stream of integers above one is passed through a chain of filters. (Initially there are no filters in the chain.) Any number that sifts through is known to be prime, and so generates a new filter in the chain that removes all of its multiples. Such a filter may be defined:

```
filter = (lambda (nextmultiple prime stream) (if
         if (eq? (head stream) nextmultiple)
               then (filter (sum nextmultiple prime) prime
                            (tail stream))
         else (strcons (head stream) (filter
                 (if if (lessthan? (head stream) nextmultiple)
                     then nextmultiple
                     else (sum prime nextmultiple))
                                  prime (tail stream))) )).
```

Such a filter, seeded with a prime for its first two arguments, will remove all multiples from the third, a stream of integers. Then

```
candidates = (strcons 2 (vectorsum <1*> candidates));
sieve      = (lambda (stream) (seft(head stream)(tail stream)));
sift       = (lambda (prime stream) (strcons prime
                  (sieve (filter prime prime stream)) ));
primes     = (sieve candidates);
```

Primes evaluates to (2 3 5 7 11 13 17 19 ...).

Another example illustrating generators is a solution for a problem attributed to Hamming: to generate a sequence in strictly ascending order beginning with 1, such that 2y, 3y, and 5y each occur in the sequence if and only if y does. The sequence begins (1 2 3 4 5 6 8 9 10 12 15 16 18 20 24...).

```
composites235 = (letrec (merge x2 x23 x235)
                        ( (lambda (s1 s2)(if
                           if (lessthan  (head s1)(head s2))
                              then (strcons (head s1)
                                            (merge (tail s1)s2))
                           else (strcons (head s2)
                                         (merge s1 (tail s2))) ))
                           (strcons 1 (<product*>
                                       <2*>
                                        x2        ))
                           (strcons 1(merge(tail x2) (<product*>
                                                      <3*>
                                                      x23   )))
                           (strcons 1(merge(tail x23)(<product*>
                                                      <5*>
                                                      x235  ))) )
                        )
                  x235)
```

Some remarks will help in reading this code. Merge is a simple binary merge of infinite, sorted streams of numbers. The stream x2 (respectively x23 and x235) is a monotonic sequence of all integers, all of whose prime factors are in the set {2} (respectively {2,3} and {2,3,5}). These three streams are defined mutually and serially; the seed of 1 in each multiplicand stream is sufficient to assure that each is infinite. Monotonicity may be proved by induction for x2, for x23, and finally for x235.

The last example is dificult. The problem is to construct a doubly-linked circular ring structure from a list. The object is to allow efficient cyclic traversal in either direction after the ring is built. The program is to be written in purely applicative style; one infers that the object of double-linking is not to facilitate insertion and deletion.

A "node" in the ring will be a triple composed of its content (taken from the input list), its left neighbor, and its right neighbor. The value of the ring function will be the node corresponding to the head of the input list. In building these triples I intend structures and environments to be built using strcons instead of cons. (In Section 3 we see that all occurrences of cons are well fixed to delay argument evaluation.) It ought to be clear that the second two members of a triple have mutually dependent definitions so circularities will arise through letrec schemes. Therefore, input must be a finite list to avoid infinite circularities.

```
ring = (lambda (lis) (if
       if     (null? lis) then <>
       elseif (null? (cdr lis)) then(letrec (node) (
                                             <(car lis)node node>)
                                        node)
       else   (letrec(topnode l-rsuffix)(
                        <(car lis)(right! l-rsuffix)(left! l-rsuffix)>
                        (double-link (cdr lis) topnode topnode))
              topnode);
left!  = (lambda (pair) (apply 1 pair));
right! = (lambda (pair) (apply 2 pair)).
```

The "help" function double-link returns two results, the left and right nodes of a doubly linked chain, whose end elements have their own left and right pointers each referencing its second two arguments. The first argument is the list to be chained:

```
double-link = (lambda (leftref rightref)(if
              if (null?(cdr lis)) then((lambda (node)<node node>)
                                         <(car lis) leftref rightref>)
              else (letrec (node l-rsuffix) (
                            <(car lis) leftref (left! l-rsuffix)>
                            (double-link(cdr lis)node rightref))
                     <node (right! l-rsuffix)> ) )).
```

This definition works because the triples are not manifested as units. It is possible to establish a reference to a triple without knowing its second two fields; one can imagine the pointers being filled in only after all triples have been "allocated".

3. REVISING THE INTERPRETER

In this section I shall modify the interpreter to reflect the lessons on streams and functional combination above. These modifications illustrate the ease of manipulating applicatively specified programs, as they confirm the usefulness of the facilities being added. As in Section 1, the language is still used operationally to describe the language.

3.1 Functional combination

Functional combination fits nicely into *apply* because McCarthy [1962, Chapter I] did not specify *apply* completely through to its *else* condition. So these two lines can be added at its end:

```
elseif (member? () args) then ()
else   (cons  (apply (car fn)(<car*> args))
              (apply (cdr fn)(<cdr*> args)) );
```

These lines are all that is necessary to implement functional combination and its minimal width rule.

Three features already present in Section 1 are used, however. First, the operand in an expression is already being evaluated, so that lists like <car*> are evaluated to lists like (car car ....) as functions. Second, since operands are all evaluated to arguments as *eval* calls *apply* (by *eval#*) evaluation of the matrix of actual parameters already occurs in row-major order. Thirdly, like McCarthy's [1962, Appendix B] this interpreter returns an emply list if the function happens to be null.

All that must be added is a check to see if any argument row is exhausted, completing the test for minimum width, and the explicit (recursive) dispatch of comlumn-major evaluation. Here the language uses itself, as we use functional combination to select off each column for column-major application. A fact we have not used may be read from this implementation: functional combinations may be nested as long as the arguments are nested properly as well.

3.2 Suspending cons

Since streams proved so useful, they ought to be implemented within the interpreter. As long as we are delaying one argument, however, we shall delay them both as Friedman and Wise [1976] recommend. The correct place to intercept evaluation is in *eval* before operands are touched. The following line is inserted into *eval* (where the IF test is):

```
elseif (eq? (oprtr! exp) CONS)
       then (cons (suspend (frst! (opnds! exp)) env)
                  (suspend (scnd! (opnds! exp)) env))
```

where

```
suspend = (lambda (form env)<SUSPEND form env>).
```

What happens then is that a user call to *cons* is intercepted immediately in *eval*, a little box is allocated by

cons as before, but it is filled with two triples called *suspensions*. A suspension is essentially an operand closure, delayed on creation to be forced on access. Since this change is in the interpreter rather than in a local library we now know that every user invocation of *cons* is so treated. Call this arrangement "suspending *cons*" or *suscons*.

Now comes a side-effect -- a very benign and natural one albeit. (This is the first "non-pure" program in the paper.) A reference to a suspension may be displaced by a reference to its suspended value at any time. Since suspensions are determinate and exist only to yield this value, such a convention turns out to be harmless.

In *apply* the lines for *car* and *cdr* must be made sensitive to suspensions:

```
    elseif (eq? fn CAR) then (first (frst! args))
    elseif (eq? fn CDR) then (rest  (frst! args)).
```

First and *rest* watch for suspensions.

```
first = (lambda (box) (if
        if   (suspended? (car box)) then (car (coercecar box))
        else (car box)  ));
rest  = (lambda (box) (if
        if   (suspended? (cdr box)) then (cdr (coercecdr box))
        else (cdr box)  ));
suspended? = (lambda (e) (if
             if     (atom? e) then nil
             elseif (null? e) then nil
             elseif (eq? (car e) SUSPEND) then TRUE
             else   nil ));
coercecar =  (lambda (box) (rplaca box (coerce (car box)) ));
coercecdr =  (lambda (box) (rplacd box (coerce (cdr box)) ));
coerce    =  (lambda (triple) (apply EVAL (cdr triple)).
```

Note here that all *car*'s and *cdr*'s are, as they were *before* this enhancement of the interpreter, blind field-accessing primitives. *Rplacd* and *rplaca* are LISP assignments into an extant box. (Very dangerous in the hands of novices; apocryphal stories abound.) This use is controlled, to displace the suspension by its value, returning the box itself as its value. *Coerce* is the analog of *force* in Section 2.1; it resumes the suspended computation.

Now the user's instances of *car* and *cdr* are all interpreted as described, but the system's *car* and *cdr* are as yet unaltered. Notice that the user cannot find out whether something is suspended. In order to access a field he must use his *car* or *cdr* which alters the system by removing any

immediate suspensions, much like inserting a probe into a perfect physical system. Suspensions are invisible to him.

Their effects, however, are present. Streams are implemented by default rather than by request; infinite and circular structures are programmed using the same vocabulary as for simple structures. Suspensions are never evaluated twice since a coerced value is "memo-ized" immediately where it can be found and reused later. Except for the time to test whether (each) value is suspended, the interpreter runs faster since some suspensions may never be evaluated.

3.3 A call-by-need interpreter

If suscons is extended further the language change is even more remarkable. Having implemented suscons, imagine an interpreter which uses suscons uniformly in place of cons. Then all environments are suspended, and argument evaluation is postponed until lookup actually has been invoked. That is, the list of arguments given to apply by eval will always be suspended, so that all of the eval# work has been postponed. That work does get done at the time the argument list is traversed, ordinarily by lookup.

This yields a call-by-need protocol [Wadsworth, 1971]; or call-by-delayed-value [Vuillemin, 1974]. Rigorous call-by-need for cons is sufficient to impress call-by-need, via Section 2, on all other functions in the system (because of the way bind and lookup are written).

When we have call-by-need protocol if becomes an ordinary primitive function; if becomes a constant, to be picked up in apply by the new line

```
    elseif (eq? fn IF) then (condscan args)
```

where

```
condscan = (lambda (pairs) (if
              if     (null? pairs) then nil
              elseif (null? (cdr pairs)) then (car pairs)
              elseif (apply 1 pairs) then (apply 2 pairs)
              else   (condscan (cdr (cdr pairs))) )).
```

Notice that condscan does not invoke eval since eval# has already suspended that work; the probing coerces it. The IF line is thereby eliminated from eval, as if CONS and IF traded spots in the interpreter.

Moreover, the cons in the new last line of apply is also suspending, so functional combination does not expand except as needed. The difficulty of using angle brackets in the ring example is implicitly solved regardless of structure.

With everything suspended there is a question whether any computation occurs at all. Any evaluation beyond the top "box" of a result must be coerced by a probe. What forces work to be done? The requirement for output!

An answer stream (or list) is handed to an output device driver still suspended. In order to print (or transmit or access) that structure car and cdr will be applied, perhaps as part of a (preorder) traversal of the structure. These probes force the result structure to appear and, indirectly, to coerce other internal values, as well. Nothing, however, is coerced unless it is necessary to a final answer and, moreover, much structure manifested for the device will be completely dereferenced as soon as the device has traversed past it. Therefore, allocation and deallocation are implicitly overlapped with computation, reducing the total internal space required to represent user disk structure. In effect, UNIX pipes have been conjured from functional style.

4. STORAGE MANAGEMENT

The interpreter depends very much on the ability to build list structures. Not only is the user free to invoke cons to build his lists, but also the interpreter uses cons to establish bindings (through eval# and bind), closures, circular environments (all in Section 1.5), and functional combination (Section 3.1). Indeed, it is the interpreter's use of cons which allows lazy evaluation to be impressed on the language so easily (Section 3.2).

The ability to build structure is, therefore, essential to the LISP interpreter. Cons allocates from a heap or from "available space". But where is structure torn down; how is unused space recovered? These questions ought to be raised by those students of data structure who are accustomed to explicit imperatives to return structure to an available space list. Such imperatives are, of course, alien to the functional programming philosophy.

The answer is that structure can be (and ought to be) recycled automatically. Abandonment of a structure is sufficient to permit an automatic scheme to recover that space for eventual reallocation by cons. This section is a brief review of such techniques, invisible in and irrelevant to the language definition embodied by the interpreter, but implicit in any implementation. Any of these algorithms would be imbedded within a serious implementation of cons.

A major contribution of LISP has been the progress it motivated in automating storage management. Knuth [1975] gives a good overview. Results outlined there and updated here

should convince one that automatic techniques are preferable to trusting the user to identify correctly his own dereferenced structure; the interpreter, therefore, is freed from explicitly xcondemning obsolete structure, as well.

4.1 Reference counts

Quite an elementary scheme for storage management is Collin's [1960] reference counts. Every node in the _heap_ (the region of memory allocated for linked storage) includes an extra field capable of holding an integer count of the references _to_ this node. The range of this count, from one up to some limit -- the "infinity" of counts -- need not be wide enough for all data structures (The worst case is all pointers referencing a single node), but a wider range is better. Every time a node is borrowed by establishing another binding to it, the reference count is increased unless it is already "infinity"; every time such a binding is abandoned the count is decreased unless it is "infinity" or one. Set to one on initial allocation by _cons_, the count of a uniquely referenced node ought to be one even if a "non-infinite" number of now obsolete bindings once had shared it. Thus the final dereferencing may be detected whereupon, the node is returned to available space. Nodes whose counts reach "infinity" must either be recopied in a new node, or they become permanently allocated.

There are three problems with reference counts. First, not all circular structures can be managed in this way. A simple circular list without external reference might still have all counts set to one, and thus not be returned to available space. Bobrow [1980], and Friedman and Wise [1979] suggest approaches to solve this problem, although neither solution is complete. The fact that circularities can only be established via _letrec_ is in our favor here; _letrec_ is only rarely used outside of recursive function definitions. Thus, we might be able to characterizè popular patterns of _letrec_ usage where reference counts are sufficient using these or other approaches, but they are not adequate for all applications of _letrec_. (e.g. the result of _ring_ in Section 2.3).

A second problem is the overhead for incrementing and decrementing counts with creation and abandonment of bindings. While that effort is proportional to structure use rather than to structure size, it is a good candidate for implementation in special purpose (parallel) hardware or microcode.

Finally, a reference count field must be quite large to cover worst-case counts, but only rarely do such counts get very big. Worst-case engineering requires much unused space --

at a premium in this problem. One solution is to tolerate an "infinity" -- a maximum reference count -- that is quite small. Deutsch and Bobrow [1976] suggest such a choice as part of a hybrid scheme; reference counts are used to postpone garbage collection, which can be used to recover such circular structure when reference counting fails.

In its favor, reference counting is decentralized and does not increase the address space. There needs to be no central control of reference count fields in a multiprocessor, as long as time is allowed for any pending increments to take effect before an ultimately "dereferenced" node is destroyed. This suggests a hardware simplicity for use in general parallel processing, perhaps using a "shadow" memory with the same addresses as the heap to maintain counts.

4.2 Mark/sweep garbage collection

Garbage collection is an effective, although (at first) apparently brutal solution to storage management. It presumes that every node in the heap is available until proven used. This is effected by a mark bit, initially cleared, in every node. Every active pointer in a register of the interpreter is taken as the root of used structure, and every such structure is traversed and marked. After the mark phase, the heap is swept sequentially; unset mark bits indicate available nodes (garbage) to be returned to available space.

The traversal of each structure requires time proportional to its size. Conventional traversal algorithms treat each structure as a tree to be traversed in preorder, where atoms, null pointers, and already marked nodes are taken as external nodes (leaves). A node is marked on its first visit. Knuth [1975] explains several algorithms, of which the last, due to Deutsch, Schorr, and Waite, is the most elegant because it uses no extra stack in its traversal. Space being at a premium, the stack is maintained in reversed tree pointers that are restored as the stack is popped.

A simple sweep phase clears all mark bits and reassembles a new available space list. L. Morris [Wise, 1979] has described an elegant two-pass sweep phase (with an extra bit needed in each pointer) that compacts the used nodes at one extreme of the heap. This is of interest because a compacted heap simplifies future allocations, allows for variously sized nodes, and aids in sharing heap space with adjacent sequential structures. The sweep phase requires time proportional to the size of the heap.

Mark/sweep garbage collectors, coded inside the primitive <u>cons</u>, are invoked when readily available space is exhausted.

Thus, a random invocation of cons may take very much time, postponing useful computation while garbage collection proceeds. That cost, however, should be considered to be distributed across neighboring invocations of cons which proceed trivially to allocate available space.

Such unpredictable delays have given garbage collection a bad name, discouraging its application in real-time applications. (Section 4.3 treats real-time collectors.) Mark/sweep remains economical on small machines, however, and forms a fine hybrid with reference counts. The traversal of all pointers can be used to reestablish accurate counts beneath "infinity" even after a node had a count so high. Several authors have described schemes that allow garbage collection to proceed in parallel with the user's program (the interpreter); the best description is by Gries [1977]. Higher scale parallel processing would require garbage collection to proceed on more than one processor at once, raising problems of expensive intercollector communication.

4.3 Recopying garbage collectors

A different class of garbage collectors tracing its history to Minsky, is best described by Baker [1978]. It uses no special mark bits within nodes and it runs in one "phase", but it does require that major portions of the heap are not used. (This requirement is not offensive in a time-shared environment where a program can expand its heap temporarily for the purpose of garbage collection.) Easiest to explain is the case in which only half of the available heap is ever used.

With a sequential array representing each half of the heap, we picture the lower half as full -- the upper half as available. The accessible information in the lower half will be copied into the upper end of the upper half. As each node is copied, its "forwarding address" will be left in its place in the lower half. Forwarding addresses are easily recognized by their magnitude, referencing the other half of the heap.

Recopying begins with all active pointers in the interpreter's registers. Those nodes are recopied into the highest nodes in the heap, leaving forwarding addresses, and the registers are forwarded directly. These few nodes form the "seed" of the newly copied active heap. Thereafter the already copied nodes are traversed sequentially (downwards in the upper half of the heap) and each pointer is treated. Such pointers cannot yet refer to nodes already copied; they refer to active nodes in the originally full (lower) half of the heap. Such nodes either contain a forwarding address, in which case the pointer is merely updated, or they are active but yet uncopied. In the latter case such nodes are treated as the interpreter

registers were: the node is copied intact, leaving a forwarding address, and the pointer is updated.

Thus, the size of the recopied heap expands ahead of the sequential traversal, until all active nodes have been forwarded. When the traversal is done all pointers have been forwarded, all accessible nodes are compacted in one end of the formerly available (upper) half of memory, and the formerly full (lower) half contains only obsolete forwarding addresses. Therefore, it can be condemned as a unit, and the halves reverse roles for the next garbage collection.

In spite of the extra space required, this scheme offers much. Linear data structures can be recopied in sequential order using a traversal sensitive to such types. In addition to the capability to run in parallel with the user's program, Baker [1978] has demonstrated that this garbage collector can run in "real time". That is, the effort of recopying can be distributed among the various implementations of _car_, _cdr_, and _cons_ so that each executes a finite share of continuous garbage collection on every invocation. This is sufficient to assure that _cons_ never runs out of space until a half-heap is indeed fully allocated.

Therefore, the space and time distribution of recopying, real-time, garbage collection seems to compare with that for reference counting. The space for an extra half-heap compares with the space for reference counts; the time is distributed uniformly among invocations which manipulate the heap in both cases. While circular structures and compaction can be handled by recopying, it does require a larger address space and proceeds without the locality of control of reference counting.

Acknowledgements: Kudos are due to McCarthy [1963] and Reynolds [1972] for laying out the rules of this game. I thank Steve Johnson and Mitchell Wand for their incisive criticisms and Nancy Garrett, diligent typest. The National Science Foundation has supported some research reflected here under grant no. MCS77-22325.

REFERENCES

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm. ACM, 21, no.8, pp.613-641.

Baker, H.G., Jr. (1978). List processing in real time on a serial computer. Comm. ACM, 21, no.4, pp.280-294.

Burge, W.H. (1975, uncited). Recursive Programming Techniques. Addison- Wesley, Reading, MA.

Bobrow, D.G. (1980). Managing reentrant structures using reference counts. ACM Trans. on Programming Languages and Systems, 2, no.3, pp.269-273.

Collins, G.E. (1960). A method for overlapping and erasure of lists. Comm. ACM, 3, no.12, pp.655-657.

Deutsch, L.P. and D.G. Bobrow. (1976). An efficient incremental, automatic garbage collector. Comm. ACM, 19, no.9, pp. 522-526.

Friedman, D.P. (1974). The Little LISPer. Science Research Associates, Palo Alto.

Friedman, D.P. and D. S. Wise. (1976). CONS should not evaluate its arguments. In Automata, Languages and Programming, ed. Michaelson, S. and R. Milner, pp.257-284, Edinburgh University Press, Edinburgh.

Friedman, D.P. and D.S. Wise. (1978). Functional combination. Computer Languages, 3, pp.31-35.

Friedman, D.P. and Wise, D.S. (1978, uncited). Unbounded computational structures. Software-Practice and Experience 8, pp.407-415.

Friedman, D.P. and D.S. Wise. (1979). Reference counting can manage the circular environments of mutual recursion. Information Processing Letters, 8, no.2, pp.41-44.

Gries, D. (1977). An exercise in proving parallel programs correct. Comm. ACM, 20, no.12, pp.921-930.

Henderson, P. (1980). Functional Programming, Application and Implementation. Prentice-Hall International, Englewood Cliffs, N.J.

Johnson, S.D. (1977). An Interpretive Model for a Language Based on Suspended Construction. M.S. Thesis, Indiana University, Bloomington, In.

Kahn, G. and MacQueen, D.B. (1977, uncited). Coroutines and networks of parallel processes. In Information Processing 77, ed. Gilchrist, B., pp.993-998, North-Holland, Amsterdam

Knuth, D.E. (1975). The Art of Computer programming I, Fundamental Algorithms (2nd ed.), Addison-Wesley, Reading, MA, Section 2.3.5.

Landin, P. J. (1965). A correspondence between ALGOL 60 and Church's lambda notation. Comm. ACM, 8, no. 2, pp.89-101.

Landin, P.J. (1966). The next 700 programming languages. Comm. ACM, 9, no. 3, pp.157-162.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. Comm. ACM, 3, no.4, pp.184-195.

McCarthy, J. (1963). A basis for a mathematical theory of computation. In Computer Programming and Formal Systems, eds. Braffort, P. and D. Hirschberg, pp.33-70, North Holland, Amsterdam.

McCarthy, J., P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin. (1962). LISP 1.5 Programmer's Manual. M.I.T. Press, Cambridge, Ma.

Reynolds, J.C. (1972). Definitional interpreters for higher-order programming languages. In Proc. 25th ACM National Conference, pp.717-740, ACM, New York.

Steele, G.L., Jr. and Sussman, G.J. (1976). LAMBDA: the ultimate imperative. AI Memo No. 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Vuillemin, J. (1974). Correct and optimal implementation of recursion in a simple programming language. J. Comp. Sys., 9, no.3, pp.332-354.

Wadsworth, C. (1971). Semantics and pragmantics of lambda-calculus. Ph.D. dissertation, Oxford.

Wise, D.S. (1979). Morris's garbage compaction algorithm restores reference counts. ACM Trans. on Programming Languages and Systems, 1, no.1, pp.115-122.