3rd ICALP July 1976
257-281

D.P. Friedman
D.S. Wise

# *CONS* SHOULD NOT EVALUATE ITS ARGUMENTS

The *cons*tructor function which allocates and fills records in recursive, side-effect-free procedural languages is redefined to be a *non-strict* (Vuillemin 1974) elementary operation. Instead of evaluating its arguments, it builds suspensions of them which are not coerced until the suspension is accessed by a strict elementary function. The resulting evaluation procedures are strictly more powerful than existing schemes for languages such as LISP. The main results are that Landin's streams are subsumed into McCarthy's LISP merely by the redefinition of elementary functions, that invocations of LISP's evaluator can be minimized by redefining the elementary functions without redefining the interpreter, and as a strong conjecture, that redefining the elementary functions yields the least fixed-point semantics for McCarthy's evaluation scheme. This new insight into the role of constructor functions will do much to ease the interface between recursive programmers and iterative programmers, as well as the interface between programmers and data structure designers.

## INTRODUCTION

It is common to perceive functional evaluation as requiring argument evaluation to be completed before actual functional application begins. In computer programs, however, there has been considerable development of delayed argument evaluation through schemes such as call-by-name in ALGOL 60. Probably because of obsession with arithmetic examples, which are strict (that is, require all arguments in evaluated form), it has been commonly assumed that all elementary functions were strict. During the course of a project on compilation of pure recursive LISP 1.0 (McCarthy et al. 1962) source code into iterative object code, we have uncovered a critical class of elementary functions which probably should never be treated as strict: the functions which allocate or *cons*truct data structures.

We use the term *cons* to refer to this class of functions and later to refer to a particular function which allocates records of two fields. The term is common to several list processing languages (McCarthy et al. 1962; Burstall, Collins, and Popplestone 1971) which require that the arguments to *cons* fix the values of the fields in the new record. This requirement is essential to our analysis because we assume a side-effect-free evaluation scheme in order to guarantee the integrity of environments which are passed subliminally about the system.

It is our thesis that the fields of a newly allocated record can be filled with a structure representing the suspended evaluation of the respective argument, instead of the value of that argument, as is done on systems with strict implementation of *cons*. If all other elementary functions are able to detect these suspensions and to force evaluation only at the time that the value is genuinely critical to the course of the computation (necessary to the value of the main function), then the results are the same as those of a strict evaluation scheme whenever both converge. Convergence is more likely in the new scheme since potentially divergent yet immaterial argument evaluation can be avoided. In programming terms the scheme allows exponential improvement in run times at the cost of linear degradation of the elementary system functions' times and of space overhead in dragging around environments. We are interested in the insights provided for the recursion-compiler problem because the role of constructors is critical in the definition of the source language.

Hoare (1975) has discussed the role of *cons* in building recursive data structures. The power of these structures is welcome because our restriction to purely recursive programs allows us no other kind. The language model we shall use is McCarthy's LISP, known in its basic form as LISP 1.0 or pure LISP. We owe a great deal to his definition and description of the language in terms of its own structures using only five elementary functions. The major results of this paper, in effect, have been implemented on his system with dramatic effects on his semantics resulting from simply changing three of these five functions.

Landin approached the non-strict implementation of *cons* in his discussion of streams (Landin 1965). He describes three elementary functions which accomplish a *cons* strict in only one of its two parameters. This version is satisfactory when the recursion pattern is peculiarly linear and when semantic improvements available from these structures within the interpreter can be ignored.

The remainder of this paper is divided into five sections followed by conclusions. Section I is a brief introduction to LISP notation as interpreted in this paper. Section II presents definitions of the five elementary functions used for the defining language. They provide that *cons* does not evaluate its arguments, but delays them in a form detectable and coercible by two of the other four strict elementary functions. Results in this section are proofs that McCarthy's interpreter built with these elementary functions is properly more powerful than it was as originally specified, and a strong conjecture that the new interpreter, in fact, gives the least fixed-point semantics for LISP. Section III presents a practical implementation for suspensions which prevents repeated coercion of the same suspension. This is accomplished by storing the ultimate value back into the node which ought to have contained it in the original interpretation scheme, replacing the suspension which led to it. Section IV relates Landin's streams to LISP as interpreted with the new *cons*. Streaming is shown to be less powerful by considering cases where evaluation should not follow a sequential pattern. An analogy between streams and sequential files is extended to an analogy between suspensions and random access (overlapping tree structure) files which suggests that file handling may be implicit in programming style. In Section V we consider familiar functions whose arguments are to be selectively evaluated which have hitherto been implemented in LISP as special forms but now are expressible as ordinary functions.

## I. LISP

The five elementary functions presented by McCarthy will be called *:car*, *:cdr*, *:cons*, *:eq*, and *:atom*. These functions are redefined in two ways to allow the interpretation of *cons* to postpone evaluation of its arguments. In both cases the five are simply called *car*, *cdr*, *cons*, *eq*, and *atom*. Our first redefinition is sufficient for the theoretical results in Section II and even Section IV, but are extremely inefficient. The versions of basic functions presented in Section III yield a system of equivalent power, but are more efficient and these definitions are used in both Section IV and Section V.

The notation used throughout the paper for form invocation is the S-expression of McCarthy. The invocation (f a b c) asks that the function, f, be applied to the arguments a, b, and c. Usually this means that the values of the three actual parameters are to be bound to the three formal parameters in the interpretation of the body of f, but there are exceptions. If f were a special form (McCarthy 1962), then the list of the three unevaluated arguments would be bound to the first actual parameter of f. If f were defined with a nontrivial atom as its formal parameter list, as discussed in Section V, then the list of the three values associated with the arguments would be bound to that atom.

A *list* is a sequence of zero or more atomic elements or lists. A list is also written using the parenthesis notation; whether the interpreter accesses it as an expression rather than as a value determines whether evaluation will occur. The empty list is denoted by the atom NIL*; the value of (:car z) is the first element on the list, z; the remainder of the list, z, exclusive of (:car z) is (:cdr z); (:cons q z) gives the list which is the list z with the form q stuck on the front.

A little of the record manipulation of LISP is needed for Section III. Atoms are references to distinguishable structures. The rest of the data structure is represented by references to records of two fields: the *A-field* and the *D-field*. New nodes are available through :*cons* which places its two arguments in the *A-field* and *D-field*, respectively. The functions :*car* and :*cdr* extract the respective fields from a reference to a non-atomic structure. The predicate :*atom* tests if its argument is atomic, and the predicate :*eq* tests if its two atomic arguments are the same. On non-atomic arguments :*eq* is undefined.

---

*Symbol strings composed entirely of upper case letters are constants; that is, they evaluate to themselves. LISP provides the function *quote* for this role; only atoms may be quoted.

We have made a notational change in the syntax of conditional expressions which needs to be explained only to LISPers who have thus far breezed through this section. McCarthy's conditional form, *cond*, requires its tail to be structured as a series of lists of two elements which are often called "cond-pairs." Rather than introduce the redundant extra parentheses which make the pairings explicit, we use the commenting keywords *if*, *then*, *elseif*, and *else* to group and to enhance legibility. In the interpreter we define *cond* to take its predicates and selections unpaired as one long alternating list. The reader who wishes to interpret an invocation of *cond* literally should ignore the commenting keywords.

For example, we postulate the predicate *same* which is defined only in terms of *:eq* and *:atom*, exclusive of the other three elementary functions whose semantics are altered in this paper.

```
(same sexp atm) ≡ (cond
    if (:atom sexp) then (:eq sexp atm)
    else NIL).
```

This function is a convenient way of avoiding applications of *:eq* to non-atoms in the interpreter. In many implementations *:eq* is a reference comparator, which is sufficient for its semantics but also provides unnecessary comparisons on non-atoms. Even in McCarthy's Appendix B interpreter *:eq* is applied to (potential) non-atoms in a manner which we judiciously avoid with *same*.

## II. ALLOCATING WITH INCOMPLETE CONTENTS

*Definition:* A function is *strict in its* $i^{th}$ *parameter* if divergence of its $i^{th}$ argument implies the function diverges with that argument.
*Definition:* A function is *strict* (Vuillemin 1974) if it is strict in all of its parameters.

A strict function may be evaluated by evaluating *all* of its arguments before its definition is interpreted. If it is strict in only a few parameters then the corresponding arguments may be evaluated first. In an environment where all functions are strict, the behavior is like the call-by-value scheme of ALGOL 60. Vuillemin specifies that *all* elementary (machine level) functions, except conditional expressions, are strict, although other functions need not be. The foundation of our scheme is that we weaken this requirement.

In recursive programming languages the role of the constructor function, here called *cons* (McCarthy 1963; Burstall, Collins and Popplestone 1971), is to allocate a new node from the available space pool and to fill its fields with its arguments. Languages with iterative control structures and assignment statements separate these two operations with sequential statements, allowing fields to be undefined while other operations intervene. In both protocols, the value returned by *cons* is a reference to the allocated node.

*Definition:* A *form* is an unevaluated expression.
*Definition:* An *environment* is a function which maps formal parameters to their values.
*Definition:* A *suspension* is a data structure, accessible only to the interpreter of a program, which is composed of a form and an environment for the form's eventual evaluation.

A suspension provides enough information to evaluate a form whenever its value is needed. This obtains because an environment is not subject to side-effects which could invalidate delayed evaluation. Several languages like LISP and SIMULA (Dahl and Nygaard 1966) allow the environment to be accessible as a single data structure. By hiding the environment in a data structure inaccessible to the user, we avoid such a situation. The function, *suspend*, takes a form and an environment as arguments and creates a suspension from them. The auxilliary selector functions, *form* and *env*, are defined over suspensions to return the respective fields. There is also a type predicate, *suspended**.

Our *cons* allocates a fresh node from available space and fills the appropriate fields with a suspension for each argument. This specification makes no assumption about the number of fields within a node, but assumes each field must be large enough to hold a reference to a suspension instead of the eventual value of the suspension. Our examples will presume a node of only two fields,

---

*From these definitions, *suspend*, *form*, and *env* act very much like *:cons*, *:car* and *:cdr*. The difference is that the nodes created by *suspend* and *:cons* are disjoint and clearly distinguished by *suspended* whose domain is the set of references within the system.

which is a model sufficient to represent a node of any size through the "naturally corresponding" list structure described for trees by Knuth (1975). This convention of suspending arguments for *cons* allows it to be non-strict yet never allows the contents of an allocated node to be undefined. The value returned by *cons*, as in the earlier protocols, is a reference to the newly allocated node.

First Redefinition of the Primitives

We present a reinterpretation of the elementary functions for LISP. The elementary predicates, at least, will not be confused because

```
(eq q r) ≡ (:eq q r)
```

and

```
(atom q) ≡ (:atom q) .
```

*Cons* is a *special* form (McCarthy 1962) which takes two arguments that become a single list of two forms bound to its first formal parameter. Whatever environment exists at the time of invocation of *cons* is bound to the second formal parameter. We define *cons* through *scons*:

```
(scons arg env) ≡
    (:cons (suspend (:car args) env)
           (suspend (:car (:cdr args)) env)) .
```

The selectors, *car* and *cdr* always assume their argument is a reference to a node allocated by *:cons*, and never yield a suspension as a result.

```
(car q) ≡ (eval (form (:car q))(env (:car q)));
(cdr q) ≡ (eval (form (:cdr q))(env (:cdr q))).
```

If the evaluation process traverses other suspensions, those other suspensions are only encountered within *car* and *cdr* so evaluation continues. Evaluation within those two functions, called *coercion*, terminates when an atom or an application of *cons* is encountered.

*Observation 1*: The structures built with *scons* have the property that the nodes allocated by *:cons* only contain references to nodes allocated by *suspend*, and that the nodes allocated by *suspend* contain only references to nodes allocated by *:cons* or to atoms.

The evaluation scheme specified appears to be the same as the usual call-by-value protocol similar to that of ALGOL 60. There is a very significant difference: *not in when evaluation occurs, but in how far evaluation proceeds.* When call-by-name forces evaluation on an actual use of a formal parameter, it forces a complete evaluation because the ALGOL 60 model presumes that all elementary functions are strict, at least, in one parameter. In our LISP model with suspensions, *cons* is not strict in any argument, so evaluation stops at the first application of *cons*. As a result, the coercing of a suspension "bottoms out" much sooner than the forced evaluation of a similar parameter called-by-name. For example, if f, g, and h are functions and x, y, and z are arguments to these functions, then evaluation of

```
(car (cons (cons (f x)(g y)) (h z)))
```

does not cause evaluation of either (f x), (g y), or (h z). It returns a reference to

```
(cons (f x) (g y))
```

after performing two storage allocations with *:cons* and constructing four suspensions with *suspend*. In the evaluation of

```
(car (cons (f x) (cons (g y) (h z))))
```

the form

```
(cons (g y) (h z))
```

is converted into a suspension instead of being evaluated, and since that suspension is not accessible to any permanent environment it will never be coerced. It, like the suspension for (h z) in the former example, is lost to the system garbage collector.

We postulate a LISP evaluator for the side-effect-free language known as LISP 1.0 (McCarthy 1962, Chapter one). The appendix presents an interpreter patterned after McCarthy's. The *eval/apply* interpreter is the same interpreter using McCarthy's elementary functions.

We present an example below which does not really fit the language LISP 1.0 because it uses the data structure "number" and arithmetic. We use the example in later proofs about the *eval/apply* LISP 1.0 system which depend on order of evaluation rather than on the properties of arithmetic. We choose to violate the data type of LISP 1.0 in order to present an example of a function which generates a familiar infinite sequence. All arithmetic functions are strict.

*Example:* The infinite sequence $\frac{1}{1}, \frac{1}{4}, \frac{1}{9}, \ldots, \frac{1}{n^2}, \ldots$ can be expressed by (terms 1) where

```
(terms n) ≡ (cons(reciprocal(square n))
                 (terms(add1 n))).
```

This sequence has partial sums which converge to $\pi^2/6$, but that property is not critical to the following discussion. The important fact is that evaluation of (terms n) does not immediately diverge; it results in a node referencing two suspensions. The interpretation of this value may, nevertheless, reflect its divergent behavior. An attempt to print it would diverge because the print routine traverses list structures using strict elementary functions in order to find printable atomic elements. Other uses of (terms 1) do not reflect its potential divergence. For instance, extracting the third term in the sequence can be accomplished by the form

```
(car (cdr (cdr (terms 1)))).
```

The value 1/9 results from the construction of six suspensions during the allocation of three nodes, and *reciprocal* and *square* are invoked only once during the coercion of one of those six suspensions.

## First Results

The first results establish that McCarthy's LISP 1.0 interpreter, here called *:eval/:apply*, is strictly less powerful than the same interpreter, called *eval/apply*, which interprets *car*, *cdr*, and *cons* as described above. The prototype interpreter, presented in the appendix, forms the basis for this argument under two interpretations: *:eval/:apply* is obtained by substituting *:car*, *:cdr*, and *:scons* for all instances of *car*, *cdr*, and *scons* in the *eval/apply* interpreter. We shall refer to a parameter *p* of the former interpreter as *:p* to make the substitution appear more complete.

There are several occurences of *:car*, *:cdr*, and *:cons* in the prototype code; these are not to be changed. They exist because the interpreter builds structures, argument lists and environments, and searches them. The use of *:cons* is required to build these structures, but the non-strict *cons* is only available through the interpreter at this time.

There is no choice but to use McCarthy's original functions for these purposes. In Theorem 3 we shall return to bootstrap the interpreter so that these occurrences of *:cons* are also non-strict.

Because the first three results rest on program-correctness arguments (Manna and Pnueli 1970), we must define three relations which will describe the behavior of the two interpreters for the three kinds of data structures used: values, argument lists, and environments.

*Definition:* The relation "$<_v$" , read "coerces to value," is defined as follows:

i) If (atom a) then a $<_v$ a ;

ii) If (not (atom y)) and x $<_v$ y then both

(car x) $<_v$ (:car y) and (cdr x) $<_v$ (:cdr y) .

*Definition:* The relation "$<_a$" , read "coerces to arglist," is defined as follows:

i) NIL $<_a$ NIL ;

ii) If r $<_v$ s and x $<_a$ y then

(:cons r x) $<_a$ (:cons s y) .

*Definition:* The relation "$<_e$" , read "coerces to environment," is defined as follows:

i) NIL $<_e$ NIL ;

ii) If (atom a) , r $<_v$ s , and x $<_e$ y then

(:cons (:cons a r) x) $<_e$ (:cons (:cons a s) y).

It is fortunate for testing the above relations that the predicates, *atom* and *:atom,* as well as *eq* and *:eq,* coincide.

The first theorem says that whenever the *:eval/ :apply* interpreter converges then the *eval/apply* inte preter converges to a related value from related inpu

*Theorem 1:* If form $<_v$ :form and env $<_e$ :env

then (eval form env) $<_v$ (:eval :form :env) .

*Proof:* The program-correctness induction proceeds on six invariantly true predicates:

1. If form $<_v$ :form and env $<_e$ :env then

(eval form env) $<_v$ (:eval :form :env) ;

2. If fn $<_v$ :fn and args $<_a$ :args and env $<_e$ :env then (apply fn args env) $<_v$ (:apply :fn :arg :env) ;
3. If fpl $<_v$ :fpl and apl $<_a$ :apl and env $<_e$ :env then (pairlis fpl apl env) $<_e$ (:pairlis :fpl :apl :env) ;
4. If (atom at) and env $<_e$ :env then (assoc at env) $<_v$ (:assoc at :env) ;
5. If unargs $<_v$ :unargs and env $<_e$ :env then (evlis unargs env) $<_a$ (:evlis :unargs :env) ;
6. If tail $<_v$ :tail and env $<_e$ :env then (evcon tail env) $<_v$ (:evcon :tail :env) .

*Lemma:* If x $<_v$ :x and y $<_e$ :y then (car x) $<_v$ (:car :x) ; (cdr x) $<_v$ (:cdr :x) ; (scons x y) $<_v$ (:scons :x :y) .

The first two conclusions are trivial: vacuously when x is an atom and by definition of $<_v$ otherwise. In the last case (using *scons* from the appendix)

```
    (scons x y) ≡ (:cons (:cons (car x) y)
                         (:cons (car (cdr x)) y))
```

which is clearly not an atom. Moreover,

```
    (car (scons x y)) = (eval (car x) y)  and
    (cdr (scons x y)) = (eval (car(cdr x)) y) .
```

However,

```
    (:car (:scons :x :y)) = (:eval (:car :x) :y) and
    (:cdr (:scons :x :y)) = (:eval (:car (:cdr :x)):y).
```

Because (car x) $<_v$ (:car :x) and (car(cdr x)) $<_v$ (:car(:cdr x)) by the first part of the lemma, and because of invariant Predicate 1 the result is established. □

Results like this lemma are easily obtained for the other relations, and similar results on *atom* and *eq* are available because these predicates are identical in both interpreters. The proof of

Theorem 1 now degnerates into a line-by-line analysis of the recursive code. We shall only present the arguments on two lines: one from "eval" and one from "apply."

Consider the CONS line in "eval." We want to show that if form $<_v$ :form and env $<_e$ :env and (not (:atom :form)) and (:atom (:car :form)) and (:eq (:car :form) CONS) then the following are all true:

```
(not (atom form)); (atom (car form));
(eq (car form) CONS);
(scons(cdr form)env) <v (:scons(:cdr:form):env).
```

The proof is easy with the lemma. The first three fall from it and the definition of $<_v$ . They and the lemma applied twice give the last required result verifying Predicate 1 for this case.

Finally, consider the CAR line in "apply." Assume that fn $<_v$ :fn , args $<_a$ :args , env $<_e$ :env , (:atom :fn) , and (:eq :fn CAR) . From the definition of $<_a$ we have (:car args) $<_v$ (:car :args) and thence by the lemma (car (:car args)) $<_v$ (:car (:car :args)) establishing this case for Predicate 2.

The remainder of the proof is tediously similar.∎

*Theorem 2*: McCarthy's evaluation scheme with our three elementary functions, *eval/apply*, can evaluate forms on which the unmodified evaluator, *:eval/:apply*, diverges.

*Proof*: The example which appeared above will suffice: (car(cdr(cdr(terms 1)))) which extracts the third term from an infinite sequence. ∎

Next we postulate a system for *eval/apply* bootstrapped upon itself so that the occurrences of *:cons* in the prototype interpreter in the appendix now create suspensions. We call this system the *superinterpreter* for reasons which will become apparent. In the resulting system there is only one breed of *cons*, the kind that suspends its arguments, and only one breed of *car* and *cdr*, the kind which coerce suspensions.

The superinterpreter is not hampered by two kinds of errors which normally cause a function to diverge in *:eval/:apply*. The first case arises from the *cons* in *evlis*. When this *cons* is strict every actual parameter is evaluated; if it is an expression only involving strict operators, such as (quotient 1 0), then evaluation is complete and divergence implies that the form being *eval*uated diverges immediately (call-by-value). If, however, the *cons* in *evlis* is the suspending kind, then argument evaluation is delayed until the result is accessed by the application of a strict elementary function to a formal parameter sometime later during the course of interpretation (call-by-name). All non-elementary functions are assumed to be strict in no parameters until then.

Another error which can be avoided by the suspending *cons* (see *pairlis*) is that of insufficient arguments. (*Pairlis* builds the environment, binding formal and actual parameters.) The only way in which this error will be caught is, again, as a result of a strict elementary function being applied directly or indirectly to the formal parameter which is unbound because of the error.

*Theorem 3:* The superinterpreter is properly more powerful than the interpreters of Theorem 1.

*Proof:* The equivalence of the interpreters when *:eval/:apply* or *eval/apply* converges is established through a proof much like that of Theorem 1, but simpler because with only one *cons* there is only one "coerces to" relation for all structures. The following example converges under the superinterpreter by escaping the pitfalls of argument evaluation and parameter binding by postponing the construction of its internal data structures. Define the function *second* as

```
(second x y z) ≡ y .
```

The form,

```
(second (quotient 1 0) 3)
```

evaluates to 3 in spite of the strictly divergent first argument and the missing third argument. ■

*Example:* As an example of a form whose evaluation diverges in LISP 1.0, even under the superinterpreter, we offer

```
((label gardenpath
   (λ (x)  (cdr (cons x  (gardenpath x) )) )
 )  NIL) .
```

In the evaluation under the superinterpreter the arguments to *cons* are suspended, but the second suspension is continually coerced by application of the strict elementary function *cdr*.

Rosen (1973) has established least fixed-point results for a nondeterministic version of LISP and Wand (1975) has established related results for Reynold's (1972) style interpreters. It is clear that our superinterpreter operates deterministically and that the evaluator never descends the evaluation tree any deeper than required by the strict elementary functions within McCarthy's interpreter. As a result, it appears that the only weaknesses in Rosen's and Wand's proof can be avoided without changing the description of the interpreter in the appendix.*

*Strong conjecture:* The superinterpreter yields the least fixed-point semantics for McCarthy's *:eval/ :apply* LISP 1.0 evaluator.

Another approach to the conjecture may be based on the facts that the interpreter performs pure call-by-name (leftmost substitution rule) and that all elementary functions are 'sequential' (Vuillemín 1974) as they are eventually coerced. In particular, an argument to *cons* is only coerced as if it were part of the form (*car*(*cons*...)) or (*cdr*(*cons*...)) each of which is sequential; the other elementary functions are strict.

Henderson and Morris (1976) have independently discovered a "lazy" evaluation scheme for LISP which is presented with lucid examples and Scott-Strachey semantics. Their scheme is no less powerful than ours because they also provide a non-strict *cons*. By the strong conjecture, then, their scheme is equivalent in power to ours.

## III. SUICIDAL SUSPENSIONS

The scheme for implementing suspensions described in the previous section is terribly impractical for a running interpreter because a suspension is coerced again and again for every access to its value by a strict function. By Observation 1 a traversal of a data structure requires invocation of evaluation at every turn, and if the structure is traversed a second time, then the evaluations will all be repeated, just to get the same result (because suspended environments do not change).

*For another perspective, however, see deBakker

With the predicate, *suspended*, defined over all references within the system, as described in the previous section, we can modify the definition of *car* and *cdr* to prevent any repeated coercions of the same suspension. After the evaluation of the first coercion on a suspension the value is stored in place of the reference to the suspension. Future accesses which would have found and coerced the suspension are instead directed straight to the final value which is referenced in the same way, but is not suspended.

In the last section we saw that changing the *cons* used by the interpreter from strict to non-strict had the effect of changing all user functions from call-by-value to call-by-name. The introduction of the storing versions of *car* and *cdr* into the interpreter has the effect of changing the call-by-name scheme into a call-by-delayed-value (Vuillemin 1974) scheme. Then no argument to any function will be evaluated until it is required by a strict elementary function within the interpreter, and after that it will never be evaluated a second time.

*Observation 2:* There is at most one reference to every suspension in the system.

That reference is in the node allocated by the function *:cons* for which both invocations of *suspend* in the system are arguments. (We emphasize that the functions *form*, *env*, *suspend*, *suspended*, *:car*, *:cdr*, and *:cons* are not available to the user, and that the interpreter only uses them to define the elementary functions *car*, *cdr*, and *cons*.) Moreover, the only time this reference is accessed after its creation is during the evaluation of *car* or *cdr* of that node.

Let *rplacliba* be a function of two arguments defined similarly to *rplaca* of LISP 1.5 (McCarthy 1962). The first argument is a node allocated by *:cons* and the second is a value of some sort. *Rplacliba* performs four steps:

--Notes the reference in the *A-field* of the node, N, which is the first argument;
--Stores the reference to its second argument in the *A-field* of N;
--Liberates (returns to available space) the single node whose reference was noted above;
--Returns the value of the second argument as its value.

*Rplaclibd* is defined similarly for the *D-field*. *Rplacliba* and *rplaclibd* are not available to the user. In our applications the liberated node will always be a suspension and the replaced value will always be a reference to an atom or to a node originally allocated by *:cons*.

Since coercions only occur within *car* and *cdr*, it is those functions which we change in order to avoid repeating them.

```
(car node) ≡ (cond
   if (suspended (:car node)) then
      (rplacliba node (eval (form (:car node))
                            (env (:car node))))
   else (:car node) ) ;
(cdr node) ≡ (cond
   if (suspended (:cdr node)) then
      (rplaclibd node (eval (form (:cdr node))
                            (env (:cdr node))))
   else (:cdr node) ) .
```

If the desired reference is to a suspension, it is coerced and the resulting reference is inserted in place of the original reference. The liberation is possible based on Observation 2 and the conditional test within each function. After replacement there is neither necessity nor ability to access the suspension. If the reference isn't to a suspension, then that replacement has already occurred and the value is directly accessible.

*Theorem 4:* Theorems 1, 2, and 3, and the strong conjecture apply as well to the interpreter using the definition of *car* and *cdr* of this section.

The proof is a trivial program-correctness argument outlined informally above. ■

*Theorem 5:* Using the new functions *car* and *cdr* defined here, the number of calls to *eval* within the superinterpreter during the course of evaluating any form is less than or equal to the number of calls under McCarthy's *:eval/:apply* scheme.

*Proof:* Since the function *:cons* is strict under McCarthy's scheme, evaluation of its arguments always precedes its application. The only evaluations which are suspended in our scheme are precisely those resulting from applications of *cons*. The suspended arguments are eventually evaluated at most once, however. Since we accept his interpreter (essentially) without change, the relation between the numbers of invocations of *eval* follows. ∎

The interpreter which uses the new *cons* with suicidal *car* and *cdr* is remarkably efficient. A *cons* allocates three new nodes instead of just one as in *:eval/:apply*, but avoids the (perhaps infinite) time required to evaluate its arguments. Environments tend to get dragged around the system, preserved from garbage collection by suspended references, but argument evaluation is avoided until absolutely necessary and environment construction, itself, is suspended. On coercion of a suspension from within *car* or *cdr* the node carrying the suspension is automatically released, and when all suspensions to a particular environment have been coerced, then that environment may finally be garbage collected. The only ultimate storage cost results from suspensions which are never coerced. That space is always balanced by the time saved in not evaluating forms to useless arguments as indicated by Theorem 5. We have, therefore, modified the system by increasing linearly the time required for three of the elementary functions at the expense of space required to carry around potentially unneeded environments. However, that storage cost enables us to save time by reducing potentially exponential computation time, and even potentially divergent computation, back to practical limits.

## IV. IMPLICATIONS FOR FILE STRUCTURE

In this section we consider the implications of suspensions on communication with external devices. The requirement that the environment of a conversation be freezable as part of a suspension demands random access files in order to provide easy restoration of the device upon an unanticipated thaw. A useful model for the properties of sequential files may be found in Landin's concept of a stream (Landin 1965; Burstall, Collins, and Popplestone 1971; Hewitt et al. 1974; Burge 1975).

Landin describes a *stream* as a particular type of function which represents a sequence. A stream is applicable to an empty list of arguments and produces a pair whose first element is the next item in the sequence and whose second element is a stream for the remainder of the sequence. This definition provides for a potentially infinite sequence using only strict functions by depending on the user to control the expansion of a stream through explicit application of the successively generated streams. If we assume that the application of a stream is implicit in referring to it, then a stream may be viewed as the result of a *cons* strict in its first argument. In that view Landin's observation (1965) that streams "enable us to postpone the evaluation of the expressions specifying the items of a list until they are actually needed" is true only if lists are always processed from left to right without skipping any entries. This knowledge is available in some circumstances in particular sequential input/output which Landin was prepared to model.

The only operations which need be defined for a stream are these:*

(hs s) ≡ the first element of s;
(ts s) ≡ the stream representing all but (hs s);
(prefixs x s) ≡ the stream whose first element is the value of x and whose remainder is s; and
(nulls s) ≡ TRUE when the stream is empty, FALSE otherwise.

Since streams cannot be arguments to any other elementary function in the system, we can compare our system to the Landin system on the basis of these operations.

*Theorem 5:* McCarthy's LISP 1.0 with our elementary functions can model Landin's streams.

*Proof:* For every occurrence of (hs s) substitute (car s); for (ts s) substitute (cdr s); for (prefixs x s) substitute (cons x s); for (nulls s) substitute (same s NIL) in any program using Landin's streams. The semantics are the same because the strict elementary functions *car* and *cdr* coerce suspensions planted by *cons* in the same way that Landin's *hs* and *ts* apply the function, s, to get the next pair. ∎

*Theorem 6:* McCarthy's LISP 1.0 with our elementary functions can model more than Landin's streams.

*Proof:* The result obtains because (prefixs x s) evaluates its first parameter completely. The two systems would be equivalent if we had defined *cons* to be strict in its first parameter. The example in Section II of the sequence of terms which sums to

---

*In these definitions we have chosen the names from Burge (1975) rather than Landin (1965).

$\pi^2/6$ offers a simple counterexample for Landin's streams. Consider the Boolean form

```
(equal (car (cdr (terms 1)))
       (car (cdr (cdr (terms 0))))) .
```

Our evaluation scheme returns the value TRUE because the two terms selected from the sequence are both 1/4. Had we defined *terms* with Landin's function *prefixs* as

```
(terms n) ≡ (prefixs(reciprocal(square n))
                     (terms (add1 n)) ),
```

then the form would diverge because of a division by zero. ∎

For the remaining discussion on streams the function *prefixs* is treated as *cons* except that it is strict in its first parameter. This makes it particularly useful for describing sequential files. Let the function *read* be defined as on many LISP systems: *read* is a function of zero parameters which removes the next form from the input file and returns it as value. Then the function *input* could be defined to identify the entire file without necessarily reading it:

```
(input) ≡ (prefixs (read) (input)) .
```

If one were then careful to access the input file in order, one could then refer to (car (input)), the first form on input and (cdr (cdr (input))), the remainder of the file after the first two forms. The outer level interpreter "listening loop" for an interactive system might be written as one function, *output* whose value is passed to the printer:

```
(output s) ≡ (prefixs (eval (car s) NIL)
                      (output (cdr s))) .
```

The monitor invocation of (output (input)) runs the interpreter and results in an appropriate output stream.

Consider the function *input* with *cons* substituted for *prefixs* and a predicate *endoffile*:

(input') ≡ (cond
 *if* (endoffile) *then* NIL
 *else* (cons (read) (input')) ) .

If we invoke the form (reverse (input')) our expectation would be that this invocation would reverse the forms taken from the input file. However, because *read* is suspended until the results of *reverse* are accessed, and because *read* is a side-effecting function, the eventual effect if the *reverse* is printed is to copy the input unchanged because the first *read* forced still gets the first form from input. Thus (output(reverse(input'))) and (output (input')) transfers the input file to the output file essentially unchanged, but (output(reverse(input))) actually prints the reversal! The error is that the side-effects of *read* cannot be carried in the environments within the suspensions. If the value of (input') is taken to be a random-access file (as if it were a data structure within the machine) then the result would be the expected one.

We argue that Landin's streams fit the requirement of sequential files. (See the *dynamic list* of POP2 (Burstall, Collins and Popplestone 1971).) Because *prefixs* is strict in its first argument it is impossible to access the remainder of the sequence without noticing the existence of the first element. On the other hand, the non-strict *cons* lends itself to manipulation of random-access (tree structured) files as an extension of the rest of memory: one can move across the tree at a high level without being bothered with details at inferior levels.

In an extremely lucid discussion of streams, Burge (1975) develops the notion of a stream-function as a coroutine structure. With the suspension model of *cons* the same structure may be being traversed by several functions at once: when a suspension is coerced by one function, the value generated by the coersion is left behind in the place of the suspension for others to find if they need it. One interesting effect of this interpretation is that coroutines are written without any conscious effort by the programmer. The parts of the structure which are actually evaluated, as opposed to those which remain suspended, and the order in which evaluation occurs are not easily predicted from outside the system.

Our generalization of *cons* to non-strict is, therefore, a generalization of Landin's *prefixs* in the same way that, as Landin demonstrated, *prefixs* is a generalization of the strict *:cons*. The difference is that the structures built with the non-strict *cons* can have the evaluation of the expressions specifying any part of the overlapping tree structure postponed until they are needed.

## V. FUNCTIONS WHICH SELECTIVELY EVALUATE ARGUMENTS

The use of the non-strict *cons* within the interpreter in a way which suspends argument evaluation until the parameter is used by a strict elementary function enables certain special forms (McCarthy 1962) to be treated as functions. In order to define some of these special forms, we allow a certain class of functions which take an arbitrary number of arguments. The definition of these functions will be flagged by exactly one formal parameter directly following λ which will be bound to the list of (suspended) evaluated arguments. For example, the function *list* can be defined as the function (λ x x) so that if forced, it evaluates to the list of its (arbitrary number of) evaluated arguments. In order to facilitate writing recursions on lists of arguments we use a notation for applying a function to a list of arguments. The notation <f x> calls for an application of the function, f, to the list of evaluated arguments which result from the evaluation of x. Thus, (f a b c) is synonomous with <f (list a b c)>, and in LISP 1.5 (McCarthy 1962, Appendix B), <f x> means (apply (function f) x NIL).

The logical connectives, *and* and *or*, are defined in LISP to take an arbitrary number of arguments and to evaluate them from left to right. The first argument which evalutes FALSE (respectively, TRUE) for the special form *and* ( *or* ) terminates evaluation returning that value; if the argument list is exhausted then the value which results is TRUE ( FALSE ). The explicit order of evaluation requires care in a system implemented with strict elementary functions, because these special forms are not strict in any parameter after the first argument which evaluates to FALSE ( TRUE ). However, in the system which uses the non-strict *cons* internally, evaluation is automatically suspended so that *and* ( *or* ) becomes a function yet its strictness property remains the same.

```
and ≡ (λ x (cond
    if (same x NIL) then TRUE
    elseif (car x) then <and (cdr x)>
    else FALSE)) ;
or ≡ (λ x (cond
    if (same x NIL) then FALSE
    elseif (car x) then (car x)
    else <or (cdr x)> )) .
```

The superinterpreter gives the correct results with these definitions because the *cons* within *evlis* suspends evaluations. The pattern of the recursion with (cdr x) in *and* ( *or* ) would allow this program to work even if *evlis* were implemented with *prefixs* in place of *cons*, because that *cdr* coerces a suspended *evlis* only when the value of the *car* is needed.

The function, *if-then-else*, requires the *cons* rather than the *prefixs* within *evlis* because it does not necessarily access its arguments in order. Again, we treat *if-then-else* as a function, rather than as a special form.

(if-then-else p q r) ≡ (cond<br>
  *if* p *then* q<br>
  *else* r ) .

By generalizing *if-then-else* we can write

conditional ≡ (λ x (cond<br>
  *if* (same x NIL) *then* NIL<br>
  *elseif* (same (cdr x) NIL) *then* (car x)<br>
  *elseif* (car x) *then* (car (cdr x))<br>
  *else* <conditional (cdr (cdr x))> )) .

This *conditional* does not use the *cond-pairs* of McCarthy's interpreter. Moreover, we could not write *conditional* as a function if it did. Instead, forms in odd-numbered argument positions (except the last) are treated as predicates, and the forms in the respectively following (even-numbered) positions are taken as the associated values. With this simplification, the program is free from superfluous bracketings and the evaluator prepares for conditional evaluation (which is suspended) by a normal invocation on *evlis*. Only the odd-numbered arguments are actually evaluated until a non-NIL value is found.

## CONCLUSIONS

The result of any mechanical evaluation scheme is usually passed as a final structure to some print routine which traverses it displaying the elementary parts as part of a picture of the answer. We have proposed an evaluation scheme in which the structure building function (*cons*tructor) is non-strict so that evaluation of its arguments is delayed until they are needed by the strict elementary functions. Therefore, the first evaluation of suspended arguments might be delayed until the traversal procedure within the print routine. If the only ultimate use of a result is to display it, then the only computations necessary are those which directly contribute to the value displayed. We have proposed a very simple scheme for accomplishing this behavior in a nicely structured interpreter (LISP 1.0) simply by partitioning the five elementary functions into the strict and the non-strict.

We have implemented the elementary functions and the interpreter described in Section III, bootstrapping on an existing LISP implementation*. The appendix reflects an interpreter for our version of LISP; it appears very similar to McCarthy's. Significant differences in the behavior of the interpreter arise because the uses of *cons* by the interpreter also cause suspensions. The use in *evlis* suspends argument evaluation; the use in *pairlis* suspends environment construction; the uses in *apply*, *carlis*, and *cdrlis* suspend construction of the multiple-valued structures which result from our operation of functional combination discussed elsewhere (Friedman and Wise 1976a, 1976b). All the resulting suspensions are coerced whenever they occur as arguments to the strict elementary functions. If McCarthy's evaluator is taken intact and interpreted with our elementary functions, the evaluation scheme becomes properly more powerful. We strongly conjecture that, in fact, this interpretation yields the least fixed-point semantics for his evaluator.

In a previous paper (Friedman and Wise 1975) we propose the compilation of recursive programs into iterative machine code. The source code was to be restricted to a "stylized" language in order to assure the mechanical translation. That paper concentrated on the peculiar role of *cons* in a recursive program, which may be reinterpreted in light of the discussion herein. The result of a function which recursively builds a list using *cons*, when run under the interpreter which we propose here, develops its answer in a top-down order as the suspensions are coerced in the traversal within *print*. The normal

---

*It is noteworthy that the popular technique of implementing context-switching with "shallow bindings" and a push-down-list does not allow environments to be saved within suspensions, because suspensions are passed from nested environments out to enclosing environments. See Moses (1970) and Sandewall (1971) for further discussion of the problems with shallow binding schemes involving the role of *function* in LISP.

recursion (McCarthy's) builds the result bottom-up. The goal of iterative code is closer with the natural transformation of bottom-up to top-down code readily available from our understanding of the role of suspensions.

ACKNOWLEDGEMENT

Our deepest gratitude goes to Mitchell Wand who recast our approach to the theoretical results of Section II. The outline for the proof of Theorem 1 and the statement of the strong conjecture are his. We are priveleged to work in an environment brighted by his reflections.

Research reported herein was supported (in part) by the National Science Foundation under grants numbered DCR75-06678 and MCS75-08145.

REFERENCES

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

R.M. Burstall, J.S. Collins, & R.J. Popplestone (1971) *Programming in POP-2.* Edinburgh: Edinburgh University Press.

J. deBakker (1976) Least fixed-point revisited, in *Theoretical Computer Science* (to appear).

O.J. Dahl & K. Nygaard (1966) SIMULA--an ALGOL-based simulation language. *Comm. ACM 9*, 671-678.

D.P. Friedman & D.S. Wise (1975) Unwinding structured recursions into iterations. Tech. Report 19, Com. Sci. Dept., Indiana Univ.

D.P. Friedman & D.S. Wise (1976a) Multiple-valued recursive procedures. Tech. Report 27, Com. Sci. Dept., Indiana Univ.

D.P. Friedman & D.S. Wise (1976b) An environment for multiple-valued recursive procedures. *2nd Symp. on Programming*, Paris.

P. Henderson & J. Morris, Jr. (1976) A lazy evaluator. *Third ACM Symp. on Prin. of Prog. Lang.*, 95-103.

C. Hewitt, P. Bishop, R. Steiger, I. Greif, B. Smith, T. Matson, & R. Hale (1974) Behavioral semantics of nonrecursive control structures, in B. Robinet (ed.), *Prog. Symp.* Springer-Verlag: Berlin, 385-407.

C.A.R. Hoare (1975) Recursive data structures. *Intl. J. of Com. & Info. Sciences 2*, 105-132.

D.E. Knuth (1975) *Fundamental Algorithms (2nd ed.)*, Addison-Wesley: Reading, MA., 333.

P.J. Landin (1965) A correspondence between ALGOL 60 & Church's lambda notation, Part I. *Comm. ACM 8*, 89-101.

J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, & M.E. Levin (1962) *LISP 1.5 Programmer's Manual*, M.I.T. Press: Cambridge, MA.

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

Z. Manna & A. Pnueli (1970) Formalization of properties of functional programs. *J. ACM 17*, 555-569.

J. Moses (1970) The function of FUNCTION in LISP. *ACM SIGSAM Bull. 15*, 13-27.

J.C. Reynolds (1972) Definitional interpreters for higher-order programming languages. *Proc. ACM Natl. Conf.*, 717-740.

B.K. Rosen (1973) Tree-manipulating systems and Church-Rosser theorems. *J. ACM 20*, 160-187.

E. Sandewall (1971) A proposed solution to the FUNARG problem. *ACM SIGSAM Bull. 17*, 29-42.

J. Vuillemin (1974) Correct and optimal implementation of recursion in a simple programming language. *J. Comp. Sys. Sci. 9*, 332-354.

M. Wand (1975) Efficient axioms for algebra semantics. Tech. Report 42, Com. Sci. Dept., Indiana Univ.

# AUTOMATA LANGUAGES AND PROGRAMMING

Third
International
Colloquium
at the
University
of
Edinburgh
edited
by
S.Michaelson
and
R.Milner

---

20.21.22.23
July
1976

*

**Edinburgh**
University Press

P.J. Landin (1965) A correspondence between ALGOL 60 & Church's lambda notation, Part I. *Comm. ACM 8*, 89-101.

McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, M.E. Levin (1962) *LISP 1.5 Programmer's Manual*, Press: Cambridge, MA.

(1963) A basis for a mathematical theory ...n, in *Computer Programming and* ... (eds P. Braffort & D. Hirschberg), ...d: Amsterdam.

...) Formalization of proper- ...s. *J. ACM 17*, 555-569.

J. Mos ... FUNCTION in LISP. *ACM S...*

J.C. Reynolds ... preters for higher-order ... ACM *Natl. Conf.*, 717-

B.K. Rosen (1973) Tree- ... Church-Rosser theorems. .

E. Sandewall (1971) A proposed ... FUNARG problem. *ACM SIGSAM Bull.*

J. Vuillemin (1974) Correct and optima... of recursion in a simple programming la... *Comp. Sys. Sci. 9*, 332-354.

M. Wand (1975) Efficient axioms for algebra seman... Tech. Report 42, Com. Sci. Dept., Indiana Univ.

## APPENDIX

The appendix is in two sections. The first is a summary of the definitions of LISP's elementary functions as set forth in Sections II and III. Functions preceded by a colon (:) refer to McCarthy's five elementary functions. The second section is a prototype interpreter referenced in Sections II and V.

*Elementary functions for Section II proofs*

For the *:eval/:apply* interpreter:

```
(:scons :ab :env) ≡
   (:cons (:eval (:car :ab) :env)
          (:eval (:car(:cdr ab)) :env) );
(:car :x) ≡ (:car :x);
(:cdr :x) ≡ (:cdr :x);
(:eq :x :y) ≡ (:eq :x :y);
(:atom :x) ≡ (:atom :x).
```

For the *eval/apply* interpreter:

```
(scons ab env) ≡
   (:cons (:cons (car ab) env)
          (:cons (car(cdr ab)) env) );
(car x) ≡ (eval (:car(:car x))
                (:cdr(:car x)) );
(cdr x) ≡ (eval (:car(:cdr x))
                (:cdr(:cdr x)) );
(eq x y) ≡ (:eq x y);
(atom x) ≡ (:atom x).
```

*Elementary functions for Section III's practical interpreter*

```
(scons ab env) ≡
   (:cons (suspend (car ab) env)
          (suspend (car(cdr ab)) env) );
(car x) ≡ (cond
   if (suspended (:car x)) then
      (rplacliba x (eval (form(:car x))
                         (env(:car x)) ))
   else (:car x) );
(cdr x) ≡ (cond
   if (suspended (:cdr x)) then
      (rplaclibd x (eval (form(:cdr x))
                         (env(:cdr x)) ))
   else (:cdr x) );
(eq x y) ≡ (:eq x y);
(atom x) ≡ (:atom x).
```

*Prototype interpreter following McCarthy's (1962)*

The following interpreter serves two purposes in the paper. The proofs in Section II refer to the unbracketed lines with appropriate substitutions made for the uncoloned occurrences of the elementary functions. The bracketed lines provide for formal parameter structures suggested in Section V and for functional combination (Friedman and Wise 1976a, 1976b).

The function *same*, defined by

```
(same sexp atm) ≡ (cond
   if (atom sexp) then (eq sexp atm)
   else NIL),
```

is assumed to avoid misinterpretation due to undefined values of *eq* in *apply*.

*The prototype interpreter* [Bracketed lines are ignored in Section II.]

```
(eval form env) ≡ (cond
   if (atom form) then (assoc form env)
   elseif (atom (car form)) then (cond
      if (eq (car form) QUOTE) then (car(cdr form))
      elseif (eq (car form) CONS) then
         (scons (cdr form) env)
      elseif (eq (car form) COND) then
         (evcon (cdr form) env)
      else (apply (car form)(evlis (cdr form)
         env) env) )
   else (apply (car form)(evlis (cdr form) env)
         env) )

(apply fn args env) ≡ (cond
   if (atom fn) then (cond
      if (eq fn CAR) then (car(:car args))
      elseif (eq fn CDR) then (cdr(:car args))
      elseif (eq fn EQ) then
         (eq (:car args)(:car(:cdr args)))
      elseif (eq fn ATOM) then (atom (:car args))
      elseif (eq fn NIL) then NIL
      else (apply (eval fn env) args env) )
   elseif (same (car fn) LAMBDA) then
      (eval (car(cdr(cdr fn)))
            (pairlis (car(cdr fn)) args env))
   elseif (same (car fn) LABEL) then
      (apply (car(cdr(cdr fn))) args
         (:cons (:cons (car(cdr fn))
                       (car(cdr(cdr fn)))) env))

   [elseif (anynull args) then NIL ]
   [else (cons (apply (car fn) (carlis args) env)
               (apply (cdr fn) (cdrlis args) env))])
```

```
(pairlis fpl apl env) ≡ (cond
   if (atom fpl) then env
   else (:cons (:cons (car fpl)(:car apl))
               (pairlis (cdr fpl)(:cdr apl) env)) )

[(pairlis fpl apl env) ≡ (cond
   if (atom fpl) then (cond
      if (eq fpl NIL) then env
      else (:cons (:cons fpl apl) env))
   else (pairlis (car fpl)(:car apl)
                 (pairlis (cdr fpl)(:cdr apl) env)))]

(assoc at env) ≡ (cond
   if (eq (:car(:car env)) at) then (:cdr(:car env))
   else (assoc at (:cdr env)) )

(evlis unargs env) ≡ (cond
   if (atom unargs) then NIL
   else (:cons (eval (car unargs) env)
               (evlis (cdr unargs) env)) )

(evcon tail env) ≡ (cond
   if (atom tail) then NIL
   elseif (atom (cdr tail)) then
      (eval (car tail) env)
   elseif (eval (car tail) env) then
      (eval (car(cdr tail))env)
   else (evcon (cdr(cdr tail))env) )

[(anynull lis) ≡ (cond
   if (atom lis) then FALSE
   elseif (same (:car lis) NIL) then TRUE
   else (anynull (:cdr lis)) ) ]

[(carlis mtx) ≡ (cond
   if (atom mtx) then NIL
   else (cons (car(:car mtx))
              (carlis (:cdr mtx))) ) ]

[(cdrlis mtx) ≡ (cond
   if (atom mtx) then NIL
   else (cons (cdr(:car mtx))
              (cdrlis (:cdr mtx))) ) ]
```