UNWINDING STYLIZED RECURSIONS INTO ITERATIONS

Daniel P. Friedman

David S. Wise

Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 19

UNWINDING STYLIZED RECURSIONS INTO ITERATIONS

DANIEL P. FRIEDMAN
DAVID S. WISE

REVISED: DECEMBER, 1975

UNWINDING STYLIZED RECURSIONS INTO ITERATIONS

Daniel P. Friedman

David S. Wise

Computer Science Department
Indiana University
Bloomington, Indiana 47401

Abstract

Although there are those who argue that recursion is an unnatural programming construct, it has been demonstrated to be rather an intuitive and entirely convenient way to program if learned first. Moreover, this control for repetitive processes better approximates the standards of readability, provability, and changeability required in schemes of structured programming. We bridge the gap between recursive software, like "pure LISP" programs, and iterative hardware by proposing a compilation technique. The compiler has a significant subset of recursive function definitions built with cons as its domain. It is described and illustrated by the mappings it generates for four prototype programs, representative of a powerful class of functions.

Key words and phrases: compilation, LISP, structured programming, constructor, control structure.

CR categories: 4.12, 4.22, 5.24, 5.27.

Introduction

Although there are those who argue that recursion is an unnatural programming construct, it has been demonstrated by Papert [Pap72] to be rather an intuitive and entirely convenient way to program if learned first. Moreover, this control for repetitive processes better approximates the standards of readibility, provability, and changeability required in schemes of structured programming. We bridge the gap between recursive software and iterative hardware by proposing a compilation technique. The compiler has a significant subset of recursive function definitions built with cons as its domain. It is described and illustrated by the mappings it generates for four prototypes representative of a powerful class of functions.

A significant challenge of Knuth [Knu74] is the effective replacement of recursion within loops. Meyer and Ritchie [M&R67] have shown that all primitive recursive functions can be expressed in a loop language, but primitive recursion is a weak model because it does not directly allow for the natural use of recursion in programming, providing only for a single basis and a single recursion. Chandra [Cha72] has shown that linear recursions can be compiled into efficient programs without recursion, and Walker and Strong [W&S73] investigated problems of indirect recursion in numeric computations. Risch [Ris73] has implemented a LISP-based system for compiling a small class of recursions (i.e. some which fit the pattern of primitive recursion). Darlington and Burstall [D&B73] also treated this problem in a system based on a theorem prover and a pattern matcher. The latter two efforts are similar

to ours; our approach differs in that we specify a source language, stylized recursion, for which the compilation scheme is effective. This class admits functional definitions of many alternatives whose compilation is straightforward and automatic.

The remainder of this paper is composed of six parts. The first is a section on definitions, including the definition of stylized recursion. The second section introduces four examples of stylized recursive code which will be used to illustrate the effect of the compiler. The third section presents a rough description of the run-time environment of the compiled code and the unoptimized transformation of stylized source program into iterative code for that environment. The fourth section presents optimized and non-optimized compiled code for the four functions introduced in the second section. The final section contains our conclusions on the power of the transformations described here.

Definitions

A set of function definitions is defined to be well nested if, whenever function P calls function Q, no function called by Q calls P [Weg75] . The well nested property imposes a partial ordering on the set of function definitions within a particular program.

A recursive definition is presented in the form of a sequence of cases (as in recursive function theory or like the cond structure of LISP). Case analysis is necessary to separate a basis condition (termination condition) from instances of recursion. Consider the following example from McCarthy [McC62] :

```
(subst x y z) ≡ (cond
     if[1] (equal y z) then x
     elseif (atom z) then z
     else (cons (subst x y (car z)) (subst x y (cdr z))) ) .
```

Subst substitutes x for every occurence of y in z. We find, and this example illustrates, that direct recursive calls occur most often as parameters to another function. We call these other functions constructors and restrict recursive definitions in this paper so that cons is the only constructor. Evaluation of a constructor (cons) yields the value of the function. Other candidates for constructors are the class of creation functions for records of arbitrary size [Hoa75] and binary operations for commutative monoids [M&B67]. The function definition presented as a sequence of cond lines (or cases)

```
(cond
if p1 then e1
elseif p2 then e2
   .
   .
   .
elseif pn-1 then en-1
else en )
```

is considered stylized if the following criteria are met:

(i) Every p_i , e_i is defined in terms of its own formal parameters, itself, and other well nested functions.

(ii) There is some basis expression e_b such that neither e_b nor any of $p_1 \ldots p_b$ are expressed in terms of the function being defined. Nonrecursive functions are stylized simply by letting n = 1 = b.

[1] Underlined symbols within function definitions are comments.

(iii) There may be other ground expressions, e_g, such that e_g is expressed without recursive calls.

(iv) Any recursive invocation of the function within its definition must have as one of its arguments a reduction of the associated formal parameter. The reduction must bring that argument "closer" to a value which will satisfy some p_b or p_g for a basis or a ground condition. Furthermore, other arguments in that recursive call associated with other basis conditions must either be reductions or identical to their corresponding formal parameter.

(v) An e_i or p_i may be an explicit recursive call.

(vi) An e_i may be composed of a constructor which has some recursive invocations of the function being defined as arguments.

(vii) A p_i may be composed of an independently defined predicate which has some recursive invocations of the function being defined as arguments.

This definition provides for defining a recursive function with arbitrarily many cases. The well nested property excludes simultaneous recursion (where functions P and Q are each directly recursive and also call each other) and the seventh criteria denies the use of nested recursions (where a recursion occurs as an argument to a recursive call to the same function) like the definition of Ackermann's function [Men64] .

The normal emulation of recursion (stacking of environments) passes through an environment-level twice: once invoking the recursive call and once upon its return when the constructor is applied.

A good iterative emulation would treat any environment only once, obviating the need for a stack. Under the above restrictions the only operation performed later is the application of the constructor. Given the constructor and its properties we seek a pseudo-function which yields the same final effect as the constructor but whose applications occur in the order of the invocations of the original recursion. This pseudo-function is a delayed-builder. It is convenient to separately maintain the locality where a delayed-builder takes effect; we call this locality a site. Rather than preserving an entire environment until the next level of recursion returns and then applying the constructor, only the delayed-builder (with its site) is preserved and is applied at the immediate lower level before deeper recursions are emulated. When the recursions "bottom out" this emulation need only apply the immediately previous delayed-builder to the value of the terminal case. Commutative and associative constructors such as integer multiplication have themselves as delayed-builders; compare the recursive and iterative programs for factorial.

The constructor cons, which is the exclusive constructor for the examples in this paper, may now be described completely in terms of the nomenclature introduced above. Cons performs two functions. It allocates a new node from available space and it fills the two fields (A-field and D-field) of that node with the value of its actual parameters. The field selector functions are car and cdr respectively. The value returned is the address of the newly allocated node. It is significant that this value may be determined without fulfilling the field filling phase [F&W75] . Fields of an

allocated node may remain empty for a while before the first application of the field extracting function (car or cdr) so long as the field assignment operator (rplaca or rplacd) is applied first. Therefore, the constructor cons may be effected by an immediate node allocation and by a future application of the delayed-builders, rplaca or rplacd at the site of the allocated node.

Prototype function definitions

As an illustration of various uses of cons with arguments which are direct recursions we present four closely related functions. The four follow the classification of recursive patterns in The Little LISPer [Fri74] . These functions all have the basic effect of removing incidences of the structure a from the list structure z. (Other natural effects are substitution of x for a, insertion of x before a, or insertion of x after a.) They differ only in the pattern of their application. Rember removes the first a at the top level of z; allrember removes all occurences of a at the top level of z; allrember* removes every occurence of a from z; and rember* removes the first (leftmost) occurence of a at any level of z. Recursive definitions of the four follow:

```
(rember a z) ≡ (cond
     if (null z) then ()
     elseif (equal a (car z)) then (cdr z)
     else (cons (car z) (rember a (cdr z))) );
(allrember a z) ≡ (cond
     if (null z) then ()
     elseif (equal a (car z)) then (allrember a (cdr z))
     else (cons (car z) (allrember a (cdr z))) );
```

```
(allrember* a z) ≡ (cond
     if (null z) then ()
     elseif (equal a (car z)) then (allrember* a (cdr z))
     elseif (atom (car z)) then (cons (car z)(allrember* a (cdr z)))
     else (cons (allrember* a (car z)) (allrember* a (cdr z))) );

(rember* a z) ≡ (cond
     if (null z) then ()
     elseif (equal a (car z)) then (cdr z)
     elseif (atom (car z)) then (cons (car z) (rember* a (cdr z)))
     elseif (equal (car z)(rember* a (car z)) then
               (cons (car z) (rember* a (cdr z)))
     else (cons (rember* a (car z)) (cdr z)) ).
```

These examples illustrate the expressiveness of stylized recursion. We appreciate that the class of functions definable in a stylized manner is the class of primitive recursive functions, but the facility of definition under the stylized rules is more practical than the systactic restrictions of primitive recursion itself [Men64] . We claim that these four classifications represent useful recursive schemes. Functions like quicksort or the binary tree traversals can be described using similar schemes.

The compiler

The compiler for stylized recursive functions operates on any function definition which meets the restrictions of the definition and keys on the format of a legal definition to carry out the translation. Each cond-line, (p_i, e_i) in the notation above, is treated separately and in order, and is mapped into a corresponding alternative for a conditional expression imbedded within the until part of a simple repeat loop in the target code. If (p_i, e_i) is a basis or ground condition, then the value of the image cond-line will be true, indicated by the keyword stop; if not then the value will be false, or cycle.

The resulting repeat loop has an iteration pattern which is the image of the recursion pattern of the recursive code when applied to fixed arguments. The actual effect of the iterative code is manifested through side-effects (assignment statements) on local variables however. The prototype code generated by the compiler provides for all alternatives of stylized-recursion, and therefore declares many more local variables (temporaries) than may actually be required for a specific translation. We assume that a meliorizer phase follows the compilation phase which purges unnecessary locals.

Specifically, many stacks are provided in a simple translation. This provision appears unreasonable since we strive for translation into stackless iterative code, but it may be observed that these stacks are only used in certain cases. Linear recursive functions [Cha72] may be compiled without them. Two features of stylized code require some stacking.

The first occurs when the right-hand-side of a cond-line is a constructor (cons), both of whose arguments are recursive invocations of the function being defined. In this case, the evaluation of the second argument is delayed by stacking the relevant site (available from an immediate allocation), delayed-builder, and arguments for the recursive call in several stacks, one for each of these variables. Then the local variables are reset to effect the recursive call for the first argument to cons. When the simulation of the first recursion is completed, the environment for the second recursion is recovered from all these stacks and that recursion is effected.

The second use of a stack occurs when a recursive invocation occurs within a predicate on the left-hand-side of a cond-line. It is clear that the result of such a recursion will not become part of the answer currently abuilding; it is only needed as direction of what is to be done next. Therefore, local variables, including their corresponding stacks are pushed onto a system stack, sysstk. Included in this massive suspension is the code for the remainder of the function, including the wrapping predicate which will be evaluated on completion of the required recursive call, whose value depends on the recursion just being simulated.

Every basis condition tests to see if both of these stacks has been exhausted. If nothing is stacked then the basis condition stops; if either stacking protocol has pending operations, then the former environment is recovered and the result of the basis (or ground) line is a cycle into the suspended environment. In many cases, particularly the instances of linear recursion, the stacks will be empty.

The reason that the stacking scheme is so complicated lies in the fact that the generated code is prototype and presumes that a meliorizing pass will follow the compiler to eliminate much stacking code. For instance, the stacks which are provided for right-hand-side recursion are not consolidated into a single stack before meliorization so that redundant stacking can be avoided. If a stack is never pushed (even though attempted pops are part of the compiler image of every basis) then it and all uses of it can be purged from the compiled code. If a stack is homogeneous, a situation which occurs when every push onto a particular stack involves the same constant value, then the stack may be replaced by a counter indicating its length and the constant indicating its contents.

After analysis of the stack associated with a particular program variable, that variable may be omitted because it is a constant or a trivial function of another variable. Then the meliorizer may consolidate the stacks from all functions into a single one which carries the minimal information necessary to sustain the computation. If the original recursive definition provides a linear recursion, then the result of this pass will be stackless code. If it is not linear then the stack will be used minimally to preserve environment only while another recursive call of this same function is simulated -- not merely for the duration of the evaluation of another independently defined function.

The target code of the compiler may be easily described with this understanding of the prototype stack mechanism for the iterative code. The iterative code is a LISP prog with local variables for the site, delaybuilder, answer, and a stack

associated with each of these and the formal parameters. In addition the systemstack and a temporary variable are provided. All stacks are initialized to () and ans is established as a dummy node whose D-field will receive the eventual result. (The reason that the D-field is chosen is that the corresponding delayed-builder is the same as that required when a recursion occurs as the second argument to cons. Such a recursion is frequent in LISP, a fact which increases the liklihood that delayb will be a constant valued variable and therefore purgable later.) The initial site and delayb are determined so that the first part of the result will land in the D-field of ans. After initialization the main repeat loop is executed and then the D-field of ans is returned.

The predicate portion of the loop is a long conditional statement with as many alternatives as the source code conditional up to the first occurence of a recursion on the left-hand-side of a cond-line. Each line in the compiled conditional expression corresponds to a line of the source code and, in many cases, the correspondence is quite straightforward.

When p_i is a simple predicate, defined without a recursive call, the identical predicate appears in the compiled code as a left-hand-side. When it involves a recursion then the compiled conditional is terminated with necessary pushes onto sysstk including a reference to the compiled remainder of the source code conditional which is to be resumed after the recursion is simulated.

When e_i is defined without recursion, then the current delayed-builder is applied to the current site with the value returned from the evaluation of e_i. The computation then may stop if all stacks are empty; if not then the preserved computations are resumed. If

e_i is a direct recursive call (without a constructor), then neither site nor the delayed-builder change, but the parameters are assigned their new (reduced) values and the loop is repeated. When e_i is cons applied to recursive calls then the new node is allocated immediately and is inserted in the current site by the current delayed-builder. Then it becomes the new site. If only one of the arguments to cons is a recursive call then the other field may be filled in immediately and nothing need be stacked. The delayed-builder is reassigned according to which argument requires the recursion, the parameters are assigned their reduced values and the loop is repeated. If, however, both arguments are recursive invocations then the arguments to the second call, its delayed-builder, and the site are stacked (on their corresponding stacks) while assignments are made to initiate the simulation of the first recursion directly as above. The loop is cycled; eventually a basis or ground condition is encountered which recovers the stacked values to simulate the recursion for the second argument.

Compiled examples

In order to demonstrate the behavior of our compiler/translator we present prototypes for the code actually compiled for each of the four examples presented earlier. In the unwound forms of the recursions below (cons *** ***) plays the roll of an allocation of an unfilled node which is kept as a site to be later filled in by the delayed-builders rplaca2 and rplacd2 which are identical to rplaca and rplacd (they side-effect A-fields and D-fields) but each returns its second parameter as its value.

The following system primitives are defined in Appendix A: repeat, rplaca2, rplacd2, push, pop, and notempty.

Let us consider the compiled version of rember.

```
(rember a z) ≡ (prog (temp ans site)
     (setq ans (cons *** ***))
     (setq site ans)
     (repeat until (cond
          if (null z) then <(rplacd2 site ())
                            stop>
          elseif (equal a (car z)) then <(rplacd2 site (cdr z))
                                         stop>
          else <(setq temp (rplacd2 site (cons (car z) ***)))
                (setq site temp)
                (setq z (cdr z))
                cycle> ))
     (return (cdr ans))  )[1].
```

The left-hand-side of each cond-line (predicate) is identical to that of the original code. The first two cond-lines correspond to bases and the only action is to effect the outstanding delayed-builder. (cdr ans) is always the evolving answer. For each recursion of the original code, the compiled version will perform one iteration, advancing site. After each iteration, site is the newest node on the structure with an unfilled cdr. When a basis is encountered the function halts immediately without any hidden backout.

The unwound version of allrember (below) is only a little more intricate. All the comments about rember apply to the code of allrember which introduces two new features.

---

[1] $[s_1\ s_2\ \ldots\ s_\nu]$ and $<s_1\ s_2\ \ldots\ s_\nu>$ evaluates s_1 through s_ν. The former returning a list of the evaluated expressions, the latter returning just the evaluation of s_ν. Cf. LISP's functions list and progn.

```
(allrember a z) ≡ (prog (temp ans delayb site)
     (setq ans (cons *** ***))
     (setq site ans)
     (setq delayb RPLACD2)
     (repeat until (cond
          if (null z) then <(apply delayb [site ()])
                            stop>
          elseif (equal a (car z)) then <(setq a a)
                                         (setq z (cdr z))
                                         cycle>
          else <(setq temp (apply delayb [site (cons (car z) ***)]))
                (setq site temp)
                (setq delayb RPLACD2)
                (setq a a)
                (setq z (cdr z))
                cycle> ))
     (return (cdr ans)) )[2].
```

As the site moves down the growing structure, delayb is rebound at each step. In this instance, delayb is always bound to rplacd2 so it is redundant, but this code illustrates the use of apply which effects the delayed-builder at the current site on every iteration. The second redundancy is the rebinding of a on every iteration; this operation corresponds to the evaluation of the first argument on each recursion. Compare this to the rebinding of z on each iteration which arose even in rember from the rebinding of the second parameter on each recursion. These redundancies are prime candidates for the meliorizer which would yield code very similar to the version of rember which appears above.

Next we consider the unwound version of allrember*.

---

[2]Symbol strings composed entirely of upper case symbols are constants; that is, they evaluate to themselves (see Appendix A).

```
(allremember* a z) ≡ (prog (temp ans zstk delayb delystk site sitestk)[3]
     (setq ans (cons *** ***))
     (setq site ans)
     (setq delayb RPLACD2)
     (repeat until (cond
          if (null z) then <(apply delayb [site ()])
                           (cond
                    if (notempty sitestk) then <(pop sitestk site)
                                                (pop delystk delayb)
                                                (pop zstk z)
                                                cycle>
                    else stop )>
          elseif (equal a (car z)) then <(setq z (cdr z))
                                         cycle>
          elseif (atom (car z)) then <(setq temp (apply delayb
                                                  [site (cons (car z) ***)]))
                                      (setq site temp)
                                      (setq delayb RPLACD2)
                                      (setq z (cdr z))
                                      cycle>
          else <(setq temp (apply delayb [site (cons *** ***)]))
                (push RPLACD2 delystk)
                (push temp sitestk)
                (setq site temp)
                (setq delayb RPLACA2)
                (push (cdr z) zstk)
                (setq z (car z))
                cycle> ))
     (return (cdr ans))  ).
```

---

[3] All locals (i.e. prog variables) are initialized to () by LISP. For stacks, those with suffix "stk", this initialization is sufficient; other locals must be initialized explicitly.

The last line of the source code presents a new problem for our compiler; both arguments for cons are recursive calls to allrember*. The proper handling of the two recursions involves multiprocessing, but we mimic the design of LISP which provides that the arguments of a function are to be evaluated left-to-right (i.e. evlis). The target code of this compiler therefore shares the regretable property that the well-defined order of evaluation may be abused by authors of side-effecting code. We use a stacking scheme to implement such multiple recursions. The stacks that we must introduce in the compiled code of allrember* are required by the evlis [McC62] convention; without this convention the stack could be replaced by a set; without the requirement for emulating the multiprocessor, insertions into such a set could be replaced by task generation. The departure from allrember is the introduction of stacks for site, delayb, and the formal parameter z. The four lines of cond in the unwound version correspond to the four lines of cond in the source code and the predicates are identical. The first cond-line identifies a basis condition. In the response to this predicate we check for an empty stack. If the stacks are not empty, then they are popped, the program cycles, and the iteration is continued; otherwise the iteration stops. If there were other basis lines, they would be compiled similarly. The last cond-line has more than one recursive call as arguments. The first is processed iteratively just as in the previous examples, after the parameter for the remaining calls have been stacked in reverse order. (In this case there is only one set of arguments to be stacked.)

The interesting characteristic of rember* (below) is in the fourth cond-line of the source code. In that line there is a recursive call in the left-hand-side which does not immediately contribute to the structure being grown as the final answer. In order to emulate the evaluation of that recursive call, it is necessary to suspend the growth of the current resultant value. All local parameters are pushed onto the sysstk until that recursion has a value.

There is still a strong association between the cond-lines of the source code and those of the compiled code for rember*, but not all the corresponding lines occur within the cond following until. The remainder, found in the push onto sysstk are paired with the cond-lines following the occurrence of the left-side recursion in the source code. The compilation of a left-side recursion closes the cond following the until with a sequence of stacking operations to preserve the current state of growth.

The compiler also handles function definitions with recursions from within more than one left-hand-side, with multiple recursions as arguments on both left and right side of a single cond-line. These facilities are so powerful, however, that a general example would perform an obscure operation.

```
(rember* a z) ≡ (prog (sysstk temp ans delayb site)
   (setq ans (cons *** ***))
   (setq site ans)
   (setq delayb RPLACD2)
   (repeat until (cond
        if (null z) then <(apply delayb [site ()])
                          (cond
                   if (notempty sysstk) then <(pop sysstk site)
                                              (pop sysstk delayb)
                                              (pop sysstk z)
                                              (pop sysstk temp)
                                              (setq temp (eval temp))
                                              (pop sysstk ans)
                                              temp>
                   else stop )>
        elseif (equal a (car z)) then <(apply delayb [site (cdr z)])
                          (cond
                   if (notempty sysstk) then <(pop sysstk site)
                                              (pop sysstk delayb)
                                              (pop sysstk z)
                                              (pop sysstk temp)
                                              (setq temp (eval temp))
                                              (pop sysstk ans)
                                              temp>
                   else stop )>
        elseif (atom (car z)) then <(setq temp (apply delayb
                                        [site (cons (car z) ***)]))
                                    (setq site temp)
                                    (setq delayb RPLACD2)
                                    (setq z (cdr z))
                                    cycle>
        else <(push ans sysstk)
              (push (quote (cond
                   if (apply (quote equal) ans) then
                          <(setq temp (apply delayb
                              [site (cons (car z) ***)]))
                           (setq site temp)
                           (setq delayb RPLACD2)
                           (setq z (cdr z))
                           cycle>
                   else <(setq temp (apply delayb
                             [site (cons *** (cdr z))]))
                         (setq site temp)
                         (setq delayb RPLACA2)
                         (setq z (cdr z))
                         cycle> ))  sysstk)
             (push z sysstk)
             (push delayb sysstk)
             (push site sysstk)
             (setq ans [(car z) ***])
             (setq site (cdr ans))
             (setq delayb RPLACA2)
             (setq z (car z))
             cycle> ))
   (return (cdr ans))  ).
```

Conclusions

The compiling scheme described above has been implemented in LISP 1.6 compiling from stylized recursive code into LISP prog format. Appendix B contains unmeliorized (prototype) code for the four examples presented. The possible meliorization may be understood by comparing the code in that appendix to the iterative code presented earlier. The ultimate compilation into machine language as target code is a trivial modification of this procedure.

We appreciate that the present definition of stylized recursion admits just primitive recursive functions. However, the class of function definitions included within the syntax of stylized recursion is much larger than the class of function definitions which must meet the formal restrictions of primitive recursion [Men64]. Moreover, related results on the behavior of cons as a constructor allow practical use of recursive functions defined without a basis condition [F&W75,Bur75]. Instances of such functions can only be used as arguments to operations which extract specific elements from structures of arbitrary size. This restriction is shown to be quite tolerable in several practical applications [FWW76]. Under this kind of restriction we might allow functions to be defined without basis conditions, lifting the requirement that arguments to recursive calls must contain at least one reduction to a formal parameter and providing for the definition of any partial recursive function.

The attraction of this compilation scheme is the promise of effective translation of well-written recursive code into efficient iterative code. That most efficient machine architecture is

iterative would no longer be a justification for writing iterative programs. Training a programmer first to program with structured recursive code would then be as good in terms of efficiency at run time, just as it is now better in terms of programmer efficiency.

Appendix A Utility Functions

The definitions of the system primitives used in the body of the compiled code is presented below.

Functions whose arguments are evaluated -- exprs

```
(rplaca2 x y) ≡ (car (rplaca x y));
(rplacd2 x y) ≡ (cdr (rplacd x y));
(notempty s) ≡ (not (null s)).
```

Functions whose arguments are unevaluated -- fexprs

Fexprs always have two arguments. The first, here called args, has a binding of a list of unevaluated arguments; the second, called env, has a binding of the current environment for later evaluation of the unevaluated arguments.

```
(repeat args env) ≡ (prog ()
                          test (cond
                               if (eval (car args) env) then (return NIL)
                               else (go test) ));
(push args env) ≡ (set (cadr args)
                          (cons (eval (car args) env)(eval (cadr args) env)));
(pop args env) ≡ <(set (cadr args)(car (eval(car args) env)))
                       (set (car args)(cdr (eval (car args) env)))> .
```

A note about constants

RPLACA2, RPLACD2, and *** are assumed to be quoted. The variable cycle always evaluates to false and stop always evaluates to true.

Appendix B Actual Compiler Output (unmeliorized)

```
*(PRETTY (GET @REMBER @EXPR))

(LAMBDA (A Z )(COND
   ((NULL Z )NIL )
   ((EQUAL A (CAR Z ))(CDR Z ))
   (T (CONS (CAR Z )(REMBER A (CDR Z ))))))
NIL
*(TRANSLATE  @REMBER)

(COMPILE/ TIME: 1256 MILLISECONDS)
*(PRETTY (GET @REMBER @EXPR))

(LAMBDA (A Z )(PROG
   (SYSSTK TEMP ANSW ASTK ZSTK DELAYB DELYSTK SITE SITESTK )
   (SETQ SYSSTK NIL )
   (PROGN
      (SETQ ASTK NIL )
      (SETQ ZSTK NIL ))
   (PROGN
      (SETQ ANSW (LIST *** ))
      (SETQ SITESTK (LIST ))
      (SETQ SITE ANSW )
      (SETQ DELYSTK (LIST ))
      (SETQ DELAYB RPLACD2 )
      (QUOTE (WE ASSUME CONS CONSTRUCTOR CLASS )))
   (REPEAT UNTIL (COND
      ((NULL Z )(PROGN
         (APPLY: DELAYB SITE NIL )
         (COND
            ((NOTEMPTY SITESTK )(PROGN
               (POP SITESTK SITE )
               (POP DELYSTK DELAYB )
               (PROGN
                  (POP ZSTK Z )
                  (POP ASTK A ))
               CYCLE ))
            ((NOTEMPTY SYSSTK )(PROGN
               (PROGN
                  (POP SYSSTK SITESTK )
                  (POP SYSSTK SITE )
                  (POP SYSSTK DELYSTK )
                  (POP SYSSTK DELAYB )
                  (POP SYSSTK ZSTK )
                  (POP SYSSTK ASTK )
                  (POP SYSSTK Z )
                  (POP SYSSTK A ))
               (POP SYSSTK TEMP )
               (SETQ TEMP (EVAL TEMP ))
               (POP SYSSTK ANSW )
               TEMP ))
            (T STOP ))))))
```

```
        ((EQUAL A (CAR Z ))(PROGN
           (APPLY: DELAYB SITE (CDR Z ))
           (COND
              ((NOTEMPTY SITESTK )(PROGN
                 (POP SITESTK SITE )
                 (POP DELYSTK DELAYB )
                 (PROGN
                    (POP ZSTK Z )
                    (POP ASTK A ))
                 CYCLE ))
              ((NOTEMPTY SYSSTK )(PROGN
                 (PROGN
                    (POP SYSSTK SITESTK )
                    (POP SYSSTK SITE )
                    (POP SYSSTK DELYSTK )
                    (POP SYSSTK DELAYB )
                    (POP SYSSTK ZSTK )
                    (POP SYSSTK ASTK )
                    (POP SYSSTK Z )
                    (POP SYSSTK A ))
                 (POP SYSSTK TEMP )
                 (SETQ TEMP (EVAL TEMP ))
                 (POP SYSSTK ANSW )
                 TEMP ))
              (T STOP ))))
        (T (PROGN
           (SETQ TEMP (APPLY: DELAYB SITE (CONS (CAR Z )*** )))
           (PROGN
              (QUOTE (WE STILL ASSUME CONS CLASS = ELSE ACCESS MAY BE E
MBELLISHED ))
              (PROGN)
              (PROGN)
              (SETQ SITE TEMP )
              (SETQ DELAYB RPLACD2 ))
           (PROGN
              (SETQ A A )
              (SETQ Z (CDR Z )))
           CYCLE ))))
     (RETURN (CDR ANSW ))))
NIL
*
```

```
 (PRETTY (GET @ALLREMBER @EXPR))

(LAMBDA (A Z )(COND
   ((NULL Z )NIL )
   ((EQUAL A (CAR Z ))(ALLREMBER A (CDR Z )))
   (T (CONS (CAR Z )(ALLREMBER A (CDR Z ))))))
NIL
*(TRANSLATE @ALLREMBER)

(COMPILE/ TIME: 1051 MILLISECONDS)
*(PRETTY (GET @ALLREMBER @EXPR))

(LAMBDA (A Z )(PROG
   (SYSSTK TEMP ANSW ASTK ZSTK DELAYB DELYSTK SITE SITESTK )
   (SETQ SYSSTK NIL )
   (PROGN
      (SETQ ASTK NIL )
      (SETQ ZSTK NIL ))
   (PROGN
      (SETQ ANSW (LIST *** ))
      (SETQ SITESTK (LIST ))
      (SETQ SITE ANSW )
      (SETQ DELYSTK (LIST ))
      (SETQ DELAYB RPLACD2 )
      (QUOTE (WE ASSUME CONS CONSTRUCTOR CLASS )))
   (REPEAT UNTIL (COND
      ((NULL Z )(PROGN
         (APPLY: DELAYB SITE NIL )
         (COND
            ((NOTEMPTY SITESTK )(PROGN
               (POP SITESTK SITE )
               (POP DELYSTK DELAYB )
               (PROGN
                  (POP ZSTK Z )
                  (POP ASTK A ))
               CYCLE ))
            ((NOTEMPTY SYSSTK )(PROGN
               (PROGN
                  (POP SYSSTK SITESTK )
                  (POP SYSSTK SITE )
                  (POP SYSSTK DELYSTK )
                  (POP SYSSTK DELAYB )
                  (POP SYSSTK ZSTK )
                  (POP SYSSTK ASTK )
                  (POP SYSSTK Z )
                  (POP SYSSTK A ))
               (POP SYSSTK TEMP )
               (SETQ TEMP (EVAL TEMP ))
               (POP SYSSTK ANSW )
               TEMP ))
            (T STOP ))))
```

```
        ((EQUAL A (CAR Z ))(PROGN
           (PROGN
              (SETQ A A )
              (SETQ Z (CDR Z )))
           CYCLE ))
        (T (PROGN
           (SETQ TEMP (APPLY: DELAYB SITE (CONS (CAR Z )*** )))
           (PROGN
              (QUOTE (WE STILL ASSUME CONS CLASS = ELSE ACCESS MAY BE E
MBELLISHED ))
              (PROGN)
              (PROGN)
              (SETQ SITE TEMP )
              (SETQ DELAYB RPLACD2 ))
           (PROGN
              (SETQ A A )
              (SETQ Z (CDR Z )))
           CYCLE ))))
     (RETURN (CDR ANSW ))))
NIL
*
```

```
*(PRETTY (GET @ALLREMBER* @EXPR))

(LAMBDA (A Z )(COND
   ((NULL Z )NIL )
   ((EQUAL A (CAR Z ))(ALLREMBER* A (CDR Z )))
   ((ATOM (CAR Z ))(CONS (CAR Z )(ALLREMBER* A (CDR Z ))))
   (T (CONS (ALLREMBER* A (CAR Z ))(ALLREMBER* A (CDR Z ))))))
NIL
*(TRANSLATE @ALLREMBER*)

(COMPILE/ TIME: 1415 MILLISECONDS)
*(PRETTY (GET @ALLREMBER*  @EXPR))

(LAMBDA (A Z )(PROG
   (SYSSTK TEMP ANSW ASTK ZSTK DELAYB DELYSTK SITE SITESTK )
   (SETQ SYSSTK NIL )
   (PROGN
      (SETQ ASTK NIL )
      (SETQ ZSTK NIL ))
   (PROGN
      (SETQ ANSW (LIST *** ))
      (SETQ SITESTK (LIST ))
      (SETQ SITE ANSW )
      (SETQ DELYSTK (LIST ))
      (SETQ DELAYB RPLACD2 )
      (QUOTE (WE ASSUME CONS CONSTRUCTOR CLASS )))
   (REPEAT UNTIL (COND
      ((NULL Z )(PROGN
         (APPLY: DELAYB SITE NIL )
         (COND
            ((NOTEMPTY SITESTK )(PROGN
               (POP SITESTK SITE )
               (POP DELYSTK DELAYB )
               (PROGN
                  (POP ZSTK Z )
                  (POP ASTK A ))
               CYCLE ))
            ((NOTEMPTY SYSSTK )(PROGN
               (PROGN
                  (POP SYSSTK SITESTK )
                  (POP SYSSTK SITE )
                  (POP SYSSTK DELYSTK )
                  (POP SYSSTK DELAYB )
                  (POP SYSSTK ZSTK )
                  (POP SYSSTK ASTK )
                  (POP SYSSTK Z )
                  (POP SYSSTK A ))
               (POP SYSSTK TEMP )
               (SETQ TEMP (EVAL TEMP ))
               (POP SYSSTK ANSW )
               TEMP ))
            (T STOP ))))
```

```
        ((EQUAL A (CAR Z ))(PROGN
           (PROGN
              (SETQ A A )
              (SETQ Z (CDR Z )))
           CYCLE ))
        ((ATOM (CAR Z ))(PROGN
           (SETQ TEMP (APPLY: DELAYB SITE (CONS (CAR Z )*** )))
           (PROGN
              (QUOTE (WE STILL ASSUME CONS CLASS = ELSE ACCESS MAY BE E
MBELLISHED ))
              (PROGN)
              (PROGN)
              (SETQ SITE TEMP )
              (SETQ DELAYB RPLACD2 ))
           (PROGN
              (SETQ A A )
              (SETQ Z (CDR Z )))
           CYCLE ))
        (T (PROGN
           (SETQ TEMP (APPLY: DELAYB SITE (CONS *** *** )))
           (PROGN
              (QUOTE (WE STILL ASSUME CONS CLASS = ELSE ACCESS MAY BE E
MBELLISHED ))
              (PROGN
                 (PUSH RPLACD2 DELYSTK ))
              (PROGN
                 (PUSH TEMP SITESTK ))
              (SETQ SITE TEMP )
              (SETQ DELAYB RPLACA2 ))
           (PROGN
              (PROGN
                 (PUSH A ASTK )
                 (PUSH (CDR Z )ZSTK ))
              (SETQ A A )
              (SETQ Z (CAR Z )))
           CYCLE ))))
     (RETURN (CDR ANSW ))))
NIL
*
```

```
*(PRETTY (GET @REMBER*  @EXPR))

(LAMBDA (A Z )(COND
   ((NULL Z )NIL )
   ((EQUAL A (CAR Z ))(CDR Z ))
   ((ATOM (CAR Z ))(CONS (CAR Z )(REMBER* A (CDR Z ))))
   ((EQUAL (CAR Z )(REMBER* A (CAR Z )))(CONS (CAR Z )(REMBER* A (CDR
 Z ))))
   (T (CONS (REMBER* A (CAR Z ))(CDR Z )))))
NIL
*(TRANSLATE @REMBER*)

(COMPILE/ TIME: 3045 MILLISECONDS)
*(PRETTY (GET @REMBER*  @EXPR))

(LAMBDA (A Z )(PROG
   (SYSSTK TEMP ANSW ASTK ZSTK DELAYB DELYSTK SITE SITESTK )
   (SETQ SYSSTK NIL )
   (PROGN
      (SETQ ASTK NIL )
      (SETQ ZSTK NIL ))
   (PROGN
      (SETQ ANSW (LIST *** ))
      (SETQ SITESTK (LIST ))
      (SETQ SITE ANSW )
      (SETQ DELYSTK (LIST ))
      (SETQ DELAYB RPLACD2 )
      (QUOTE (WE ASSUME CONS CONSTRUCTOR CLASS )))
   (REPEAT UNTIL (COND
      ((NULL Z )(PROGN
         (APPLY: DELAYB SITE NIL )
         (COND
            ((NOTEMPTY SITESTK )(PROGN
               (POP SITESTK SITE )
               (POP DELYSTK DELAYB )
               (PROGN
                  (POP ZSTK Z )
                  (POP ASTK A ))
               CYCLE ))
            ((NOTEMPTY SYSSTK )(PROGN
               (PROGN
                  (POP SYSSTK SITESTK )
                  (POP SYSSTK SITE )
                  (POP SYSSTK DELYSTK )
                  (POP SYSSTK DELAYB )
                  (POP SYSSTK ZSTK )
                  (POP SYSSTK ASTK )
                  (POP SYSSTK Z )
                  (POP SYSSTK A ))
               (POP SYSSTK TEMP )
               (SETQ TEMP (EVAL TEMP ))
               (POP SYSSTK ANSW )
               TEMP ))
            (T STOP ))))
```

```
        ((EQUAL A (CAR Z ))(PROGN
           (APPLY: DELAYB SITE (CDR Z ))
           (COND
              ((NOTEMPTY SITESTK )(PROGN
                 (POP SITESTK SITE )
                 (POP DELYSTK DELAYB )
                 (PROGN
                    (POP ZSTK Z )
                    (POP ASTK A ))
                 CYCLE ))
              ((NOTEMPTY SYSSTK )(PROGN
                 (PROGN
                    (POP SYSSTK SITESTK )
                    (POP SYSSTK SITE )
                    (POP SYSSTK DELYSTK )
                    (POP SYSSTK DELAYB )
                    (POP SYSSTK ZSTK )
                    (POP SYSSTK ASTK )
                    (POP SYSSTK Z )
                    (POP SYSSTK A ))
                 (POP SYSSTK TEMP )
                 (SETQ TEMP (EVAL TEMP ))
                 (POP SYSSTK ANSW )
                 TEMP ))
              (T STOP ))))
        ((ATOM (CAR Z ))(PROGN
           (SETQ TEMP (APPLY: DELAYB SITE (CONS (CAR Z )*** )))
           (PROGN
              (QUOTE (WE STILL ASSUME CONS CLASS = ELSE ACCESS MAY BE E
MBELLISHED ))
              (PROGN)
              (PROGN)
              (SETQ SITE TEMP )
              (SETQ DELAYB RPLACD2 ))
           (PROGN
              (SETQ A A )
              (SETQ Z (CDR Z )))
           CYCLE ))
```

```
        (T (PROGN
           (PUSH ANSW SYSSTK )
           (PUSH (QUOTE (COND
              (((LAMBDA (K )(APPLY (QUOTE EQUAL )K ))ANSW )(PROGN
                 (SETQ TEMP (APPLY: DELAYB SITE (CONS (CAR Z )*** )))
                 (PROGN
                    (QUOTE (WE STILL ASSUME CONS CLASS = ELSE ACCESS MA
Y BE EMBELLISHED ))
                    (PROGN)
                    (PROGN)
                    (SETQ SITE TEMP )
                    (SETQ DELAYB RPLACD2 ))
                 (PROGN
                    (SETQ A A )
                    (SETQ Z (CDR Z )))
                 CYCLE ))
              (T (PROGN
                 (SETQ TEMP (APPLY: DELAYB SITE (CONS *** (CDR Z ))))
                 (PROGN
                    (QUOTE (WE STILL ASSUME CONS CLASS = ELSE ACCESS MA
Y BE EMBELLISHED ))
                    (PROGN)
                    (PROGN)
                    (SETQ SITE TEMP )
                    (SETQ DELAYB RPLACA2 ))
                 (PROGN
                    (SETQ A A )
                    (SETQ Z (CAR Z )))
                 CYCLE ))))SYSSTK )
           (PROGN
              (PUSH A SYSSTK )
              (PUSH Z SYSSTK )
              (PUSH ASTK SYSSTK )
              (PUSH ZSTK SYSSTK )
              (PUSH DELAYB SYSSTK )
              (PUSH DELYSTK SYSSTK )
              (PUSH SITE SYSSTK )
              (PUSH SITESTK SYSSTK ))
           (PROGN
              (SETQ ASTK NIL )
              (SETQ ZSTK NIL ))
           (PROGN
              (SETQ ANSW (LIST (CAR Z )*** ))
              (SETQ SITESTK (LIST ))
              (SETQ SITE (CDR ANSW ))
              (SETQ DELYSTK (LIST ))
              (SETQ DELAYB RPLACA2 )
              (QUOTE (WE ASSUME CONS CONSTRUCTOR CLASS )))
           (PROGN
              (SETQ A A )
              (SETQ Z (CAR Z )))
           CYCLE ))))
    (RETURN (CDR ANSW ))))
NIL
*
```

References

[Bur75] W. H. Burge. Recursive Programming Techniques, Addison-Wesley, Reading, MA (1975).

[Cha72] A. Chandra. Efficient compilation of linear recursive programs. Stanford Artificial Intelligence Project, STAN-CS-72-282 (1972).

[D&B73] J. Darlington and R. M. Burstall. A system which automatically improves programs. Proc. 3rd International Conference on Artificial Intelligence, Stanford University (1973), 479-489.

[Fri74] D. P. Friedman. The Little LISPer, Science Research Associates, Palo Alto (1974).

[FWW76] D. P. Friedman, D. S. Wise, and M. Wand. Recursive programming through table look-up. Technical Report 45, Computer Science Dept., Indiana University (1976).

[F&W75] D. P. Friedman and D. S. Wise. Cons should not evaluate its arguments. Technical Report 44, Computer Science Dept., Indiana University (1975).

[Hoa75] C. A. R. Hoare. Recursive data structures. Internat. J. Comput. Information Sci. 2 (June, 1975), 105-132.

[Knu74] D. E. Knuth. Structured programming with go to statements. Comput. Surveys 6, 4 (December, 1974), 261-302.

[M&B67] S. MacLane and G. Birkhoff. Algebra, Macmillan, New York (1967).

[Men64] E. Mendelson. Introduction to Mathematical Logic, Van Nostrand, Princeton, NJ (1964).

[M&R67] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. Proc. ACM National Conference, ACM Publication P-67, Thompson Book Co., Washington (1967), 465-469.

[Pap72] S. A. Papert. Teaching children to be mathematicians versus teaching about mathematics. Int. J. Math. Educ. Sci. Tech. 3 (1972), 249-262.

[Ris73] T. Risch. REMREC--a program for automatic recursion removal in LISP. Report DLU 73/24, Datalogilaboratoriet, Uppsala University, Sweden (1973).

[W&S73] S. A. Walker and H. R. Strong. Characterizations of flowchartable recursions. J. Comp. Systems Sci. 7 (1973), 404-447.

[Weg75] B. Wegbreit. Mechanical program analysis. Comm. ACM 18, 9 (September, 1975), 528-539.