The COMFY 6502 Compiler

Author: *Henry G. Baker, http://home.pipeline.com/˜hbaker1/home.html*; *hbaker1@pipeline.com*

Last June [Baker97], we described the COMFY language which is intended to be a replacement for assembly languages when programming on 'bare' machines. This month, we provide a description of the November, 1976 COMFY-65 compiler for the MOS 6502 8-bit processor [MOSTech76], which processor—as the brains of the Apple II and the Atari personal computers—was one of the most popular microprocessors of all time.[1] Although our work on COMFY-65 was primarily a pedagogical exercise, our analogous COMFY-Z80 compiler was used to develop the code for an intelligent ASCII terminal with a $512 \times 512 \times 1$ bitmapped display in 1979. Much of the paper below was written in November, 1976, and has been edited to reflect the changes resulting from our recent port of the code from PDP-10 Maclisp to GNU Emacs Lisp [GNUELisp90] for this paper. No attempt has been made to describe the details of the 6502 architecture, except that it has an 8-bit accumulator, two 8-bit index registers ('i' and 'j', renamed from 'X' and 'Y'), a stack, and various 'page zero' locations can be used to hold full 16-bit indirect addresses.

In this paper, we provide the code for the *entire* COMFY-65 compiler, in order to demonstrate as forcefully as we can the notion that *compilers don't have to be big and complex in order to be extremely good and extremely useful*. We will wager that the text of this compiler is perhaps $1/10$ to $1/100$ the size of a standard macro assembler and far more capable. With the power of the entire Lisp language available for use within COMFY-65 macros, the amount of intelligence one can embed in these macros is limitless. Furthermore, the efficient one-pass nature of the COMFY-65 compiler means that COMFY can conceivably be used as the 'binary' load format, thus doing away with binary 'loader' formats completely.

[1]We understand that the 6502 architecture still lives on in the form of 'silicon macros' for various chip design systems.

COMFY-65

COMFY-65 is a 'medium level' language for programming on the MOS Technology 6502 microcomputer [MOSTech76]. COMFY-65 is 'higher level' than assembly language because 1) the language is *structured*—**while-do**, **if-then-else**, and other constructs are used instead of **goto**'s; and 2) complete subroutine calling conventions are provided, including formal parameters. On the other hand, COMFY-65 is 'lower level' than usual compiler languages because there is no attempt to shield the user from the primitive structure of the 6502 and its shortcomings. Since COMFY-65 is meant to be a replacement for assembly language, it attempts to provide for the maximum flexibility; in particular, almost every sequence of instructions which can be generated by an assembler can also be generated by COMFY. This flexibility is due to the fact that COMFY provides all the non-branching operations of the 6502 as primitives.

Why choose COMFY over assembly language? COMFY provides most of the features of assembly language with few of the drawbacks. In addition, the programs are far more readable than their assembly language equivalents. For example, one of the biggest pains in assembly language is generating labels for instructions that will be branched to. Not only do these labels greatly increase the size of the symbol table required, but obscure the structure of the program. COMFY eliminates all labels which are used for branching; names are used only for variables and subroutines. This elimination of labels is achieved by introducing **if-then-else**, **while-do**, and other control structures into the language.

COMFY is faster and easier to use than assembly language because it is only *one pass* instead of two (or more). For this reason and the fact that its symbol table is far smaller, there is no need to keep binary versions of most programs because COMFY can live inside the computer and compile programs directly into storage.

COMFY's argument and parameter handling conventions allow parameters to be accessed symbolically and temporary locations to be allocated and accessed as easily as in Algol. The programmer is given the choice of leaving the argument on the stack or "shallow binding" it to any location in memory [Baker78b]. (Shallow binding an argument to a location first saves the current value of that location on the stack, then sets the location to the value of the argument.) Thus, any subroutine can use locations on page zero without conflict by means of shallow binding. Examples of this power will be shown after the control primitives of COMFY have been presented.

Executable instructions in COMFY come in three flavors: *tests*, *actions*, and *jumps*. *Tests* have two possible outcomes: *succeed* and *fail* and therefore have two possible *continuations*—i.e., streams of instructions to execute next. If the test succeeds, the *win* continuation is executed; if the test fails, the *lose* continuation is executed. On the 6502, the tests are *carry*, *zero*, *negative*, and *overflow*, which succeed if the corresponding flags are on and fail if they are off.

Actions are simply executed and always *succeed*; therefore the *win* continuation always follows and the *lose* continuation is always ignored. On the 6502, the actions are all the instructions which do not divert the program counter.

Jumps are executed and ignore both their continuations. On the 6502 the only two jump instructions are *Return* (from subroutine) and *Resume* (after interrupt).

COMFY's compositional operators

(not e**)**

`not` is a unary operator which has a COMFY expression as an argument. `not` has the effect of interchanging the win and lose continuations for its argument expression. In other words, the win continuation of `(not` e`)` becomes the lose continuation of e and the lose continuation of `(not` e`)` becomes the win continuation for e.

(seq e_1 e_2 ... e_n**)**

`seq` takes a sequence of COMFY expressions and tries to execute them in sequence. If they all succeed, then the whole expression succeeds. If any one fails, the sequence is immediately terminated and the lose continuation for the whole expression is executed. In usual usage, all the e_i are actions, which corresponds to a simple instruction stream.

(if e_1 e_2 e_3**)**

`if` takes as arguments three expressions—e_1, e_2, and e_3. COMFY first executes e_1 and if it succeeds, e_2 is executed. The success or failure of e_2 then determines the success or failure of the whole `if` expression. If, on the other hand, e_1 fails, then e_3 is executed and its success or failure determines that for the whole `if` expression. In other words, `if` uses the success or failure of e_1 to choose which of e_2 or e_3 to execute next; whichever one is not chosen is not executed at all. Notice that the failure of e_1 cannot cause the failure of the whole expression.

(while e_1 e_2**)**

`while` takes two COMFY expressions as arguments—e_1 and e_2. Intuitively, e_1 is used to control a loop which successively executes e_2. Every time through the loop, COMFY executes e_1 and if it succeeds, the loop is executed once again. If e_1 fails, the loop is terminated and the success continuation for the whole `while` expression is executed. The body of the loop consists of the expression e_2. If e_2 succeeds, the loop continues; but if e_2 fails, the loop is exited and the failure continuation for the whole `while` expression is executed. This feature can be used to solve the problem of multiple exits from loops. For example, suppose that a loop is comparing two character strings for equality. The loop must stop in one of two ways—either the strings differ at some point or the comparison runs off the end of one of the strings. A sample subroutine to do this task is shown below.

```
(if (seq (lj # 0)                 ; for j=0 to n-1.
         (while (seq (cj n) llt)  ; llt fails if "carry" is set.
                (seq (l @j x)     ; compare x:y.
                     (c @j y)
                     equal        ; fail if unequal.
                     j+1)))       ; j=j+1.
    (seq ...                      ; return equality.
         return)
    (seq ...                      ; return inequality.
         return))
```

(alt e_1 e_2 ... e_n**)**

`alt` is the 'dual' of `seq`. `alt` takes a sequence of COMFY expressions and tries to execute them in sequence. If they all fail, then the entire `alt` expression fails. If any one succeeds, the sequence is immediately terminated (i.e., the rest of the sequence is not executed) and the entire `alt` expression succeeds. In usual usage, the e_i are tests; thus, `(alt` e_1 e_2`)` succeeds if and only if *either* e_1 or e_2 succeeds (we don't even find out if both would have succeeded because only the first is executed in this case).

(loop e**)**

The `loop` expression expects a single argument which is simply executed over and over again. It is equivalent to the infinite expression `(seq` e e e ...`)`. Thus, the `loop` expression can never succeed, for that would require an infinite number of executions, but it can fail.

(n e) = `(seq` e e ... e`)` (n times).

This COMFY expression is a shorthand for the expression `(seq` e e ... e`)` having exactly n e's in it. This makes many tasks like shifting on the 6502 much easier. For example, to shift the accumulator left 3 positions, one need only write `(3 sl)`.

How to Use COMFY-65

COMFY-65 lives in the file
`http://home.pipeline.com/~hbaker1/lisp/cfycmp.lsp`.
In order to run COMFY, this file must be loaded into GNU Emacs Lisp.

Set some variable—say `mprog`—to have the COMFY-65 expression as its value. Now one need only say `(compile mprog` ⟨win⟩ ⟨lose⟩`)`, where ⟨win⟩ and ⟨lose⟩ aare two *numbers* which indicate machine addresses to go to depending upon whether the program succeeds or fails. In most cases, the program will return with a `return` or a `resume`, so that both these numbers will be ignored. Therefore, putting two zeros here will usually work fine.

COMFY-65 compiles its code into the array `mem` by inserting the compiled code one byte at a time, working its way down from the top. The Lisp variable `f` indicates the lowest byte in this array which has been used so far. `compile` also returns as its Lisp value the address of the first byte of the program (the address to which one should `jms` to in order to execute the program).[2]

COMFY's symbol table is the atom structure of Lisp. In other words, to define the label `x` as referring to the address decimal 60, one should execute the instruction `(setq x 60)` to Lisp *before* one tries to compile a program which refers to `x`. This must be done because COMFY is a *one-pass* compiler, which needs to know the values of its labels *before* they are used.[3]

The core intelligence of the COMFY compiler is in the functions `compile`, `emit`, and `genbrc`. `compile` recursively examines the program while expanding macros and calling `emit` and `genbrc` to produce the actual code. `emit` understands the various addressing modes and assembles action instructions appropriately. `genbrc` is responsible for generating optimal conditional branches—it tries like crazy to produce short branches, and succeeds most of the time. Don't let the short and sweet nature of the COMFY-65 compiler fool you—it produces better branching code than all but the very best optimizing compilers.[4]

COMFY Macros

COMFY has a very powerful macro facility for handling non-primitive instructions. This macro facility works by *pattern-matching* the input expression to the *macro template* and using *pattern-directed assembly* to compute its output.

For example, suppose that the "rotate right" instruction were not primitive on the 6502 (some older models have this problem). Then we could define it (at least for the accumulator) by giving COMFY the instruction:

```
(define cmacro rr '(8 rl))
```

The first two words `define cmacro` indicate that we are defining a COMFY MACRO whose *pattern* is `rr` and whose *body* is `'(8 rl)`. Since both the pattern and the body are constants, COMFY replaces every occurrence of `rr` as a machine instruction by `(8 rl)` (which in turn compiles into `(seq rl rl rl rl rl rl rl rl)`).

To define "rotate right" for other than the accumulator, we need to get hold of the address which is passed to the `rr` macro and use it in the body. This is done by executing:[5]

```
(define cmacro (rr . ,p)
  '(seq push (l ,@ p) rr (st ,@ p) pop))
```

Let us illustrate this macro with an example of its use. Suppose the instruction `(rr i foo)` appeared in a COMFY program. COMFY would notice that `rr` was a `cmacro` and match the pattern `(rr . ,p)` against `(rr i foo)`. The first part, `rr`, matches and the second part `,p` indicates that `p` is a variable which will match anything and take that anything on as a value. Thus, the value of `p` becomes `(i foo)`. The body of `rr` starts with a `'` indicating pattern directed assembly. Inside the body, `,@ p` tells COMFY to "insert the value of `p` here in the expression". Thus the value of the assembly becomes `(seq push (l i foo) rr (st i foo) pop)` which is then compiled as if it had been

[2]It is no accident that the recursive structure of `compile` is nearly identical to that of a copying garbage collector [Baker78a]; the reasons are left as an exercise for the reader.

[3]Due to this one-pass nature, for *mutually recursive* functions and for many other reasons, one may wish to utilize a *jump table* to hold the addresses of all functions.

[4]This code also demonstrated in 1976 how to optimally compile Lisp's `and`, `or` and `not` expressions—the subject of a depressingly large number of subsequent peer-reviewed non-mutually-referenced journal articles—seemingly one for each new language.

[5]This Common Lisp syntax (translated from Maclisp) won't work in Emacs Lisp. See the Emacs Lisp code later in this paper.

typed in originally. We note that due to the recursive nature of the compiler, the embedded `rr` in this body will be expanded as in the first example above.

The intelligence for COMFY macros is found in the functions `match`, `cases`, `fapply`, and `fapplyl`. I'm not particularly proud of this code, and I'm not sure that this port to Emacs Lisp will work reliably, but it represents a sample of our thinking circa 1976.

COMFY Programming Example

Below, we show one example of COMFY-65 programming—the computation of the Universal Product Code ('UPC') parity check digit. This example is not intended as a tutorial on computing this function, nor as an example of particularly good code, but only to show the flavor of COMFY programming.

```
;;; Universal Product Code Wand parity check.
(setq
 upctable (compile '(seq 13 25 19 61 35 49 47 59 55 11) 0 0)
 code 10                              ; upc code buffer
 digit (+ code 12)                    ; digit buffer
 temp (+ digit 12)                    ; temporary location.
 upcwand
 (compile
  '(alt
    (seq (fori (\# 6) (\# 12)           ; complement right 6 upc digits.
               (l i code)
               (lxor \# 127)
               (st i code))
         (fori (\# 0) (\# 12)           ; map codes using upctable.
               (l i code)
               (not
                (forj (\# 0) (\# 10)
                      (c j upctable)
                      ~=\?))            ; fail if equal.
               (stj i digit))           ; store index of upctable.
         decimal                        ; set decimal arithmetic mode.
         (l \# 0)                       ; clear ac.
         (fori (\# 0) (\# 12)           ; add up the even digits.
               (+ i digit)              ; loop control clears carry!
               i+1)                     ; only every other one.
         (st temp)                      ; save partial sum.
         c=0                            ; clear the carry.
         (2 (+ temp))                   ; multiply by 3.
         (fori (\# 1) (\# 12)           ; add up the odd digits.
               (+ i digit)              ; loop cotrol clears carry.
               i+1)                     ; only every other one.
         (lxor \# 15)                   ; select low decimal digit.
         =0\?                           ; fails if non-zero.
         return)
    (seq break                          ; signal failure.
         return))
  0 0))
```

Compiler Source Code

Below is a listing of the COMFY compiler source code recently ported to Emacs Lisp for this *Sigplan Notices* paper. The major capability missing from this COMFY-65 compiler is support for a **case** construct which would allow dispatching to a number of different continuations based on the actual value of a particular byte. Such a `case` capability was included in a later COMFY-Z80 compiler—we leave adding it to COMFY-65 as an exercise for the reader.

This COMFY-65 compiler was never used extensively, so its overall quality should be suspect. The subsequent COMFY compiler for the Z-80 was used for a substantial programming effort—the software for an intelligent bit-mapped display terminal—so it was debugged much more thoroughly.

```
;;; Numbers in Emacs Lisp are always decimal.
;;; Define 6502 op codes.
;;; Basic test instructions.
(put 'c=1\? 'test 176)       ;;; test carry=1.
(put 'c=0\? 'test 144)       ;;; test carry=0.
(put 'llt 'test 144)         ;;; logically <.
(put 'lge 'test 176)         ;;; logically >=.
(put '=\? 'test 240)         ;;; equal.
(put '~=\? 'test 208)        ;;; not equal.
(put '=0\? 'test 240)        ;;; equals zero.
(put '~=0\? 'test 208)       ;;; not equal to zero.
(put 'v=1\? 'test 112)       ;;; test overflow=1.
(put 'v=0\? 'test 80)        ;;; test overflow=0.
(put '<\? 'test 48)          ;;; test arithmetic <.
(put '>=\? 'test 16)         ;;; test arithmetic >=.
(put '<0\? 'test 48)         ;;; test arithmetic <0.
(put '>=0\? 'test 16)        ;;; test arithmetic >=0.
;;; Group 0.
(put '\? 'skeleton 32)       ;;; test.
(put 'stj 'skeleton 152)     ;;; store j.
(put 'lj 'skeleton 168)      ;;; load j.
(put 'cj 'skeleton 200)      ;;; compare j.
(put 'ci 'skeleton 232)      ;;; compare i.
;;; Group 1.
(put 'lor 'skeleton 17)      ;;; logical or.
(put 'land 'skeleton 49)     ;;; logical and.
(put 'lxor 'skeleton 81)     ;;; logical xor.
(put '+ 'skeleton 113)       ;;; add with carry.
(put 'st 'skeleton 145)      ;;; store accumulator.
(put 'l 'skeleton 177)       ;;; load accumulator.
(put 'c 'skeleton 209)       ;;; compare accumulator.
(put '- 'skeleton 241)       ;;; subtract with borrow.
;;; Group 2.
(put 'asl 'skeleton 10)      ;;; arithmetic shift left.
(put 'rl 'skeleton 42)       ;;; rotate left.
(put 'lsr 'skeleton 74)      ;;; logical shift right.
(put 'rr 'skeleton 106)      ;;; rotate right.
(put 'sti 'skeleton 138)     ;;; store i.
(put 'li 'skeleton 170)      ;;; load i.
(put '1- 'skeleton 194)      ;;; decrement.
(put '1+ 'skeleton 226)      ;;; increment.
;;; random instructions.
(put 'trap 'skeleton 0)      ;;; programmed break.
(put 'save 'skeleton 8)      ;;; push processor state onto stack.
(put 'restore 'skeleton 40)  ;;; restore processor state from stack.
(put 'push 'skeleton 72)     ;;; push accumulator onto stack.
(put 'pop 'skeleton 104)     ;;; pop accumulator from stack.
(put 'c=0 'skeleton 24)      ;;; clear carry.
(put 'c=1 'skeleton 56)      ;;; set carry.
(put 'seb 'skeleton 24)      ;;; set borrow.
(put 'clb 'skeleton 56)      ;;; clear borrow.
(put 'v=0 'skeleton 184)     ;;; clear overflow.
(put 'enable 'skeleton 88)   ;;; enable interrupts.
(put 'disable 'skeleton 120) ;;; disable interrupts.
(put 'binary 'skeleton 216)  ;;; set binary mode.
(put 'decimal 'skeleton 248) ;;; set decimal mode.
(put 'i+1 'skeleton 232)     ;;; increment i.
(put 'j+1 'skeleton 200)     ;;; increment j.
(put 'i-1 'skeleton 202)     ;;; decrement i.
(put 'j-1 'skeleton 136)     ;;; decrement j.
(put 'nop 'skeleton 234)     ;;; no operation.

(put 'return 'jump 96)
(put 'resume 'jump 64)
(defvar jmp 76)
(defvar jsr 32)

(defvar mem (make-vector 10 0)
  "Vector where the compiled code is placed.")
(setq mem (make-vector 100 0))

(defvar f (length mem)
  "Compiled code array pointer; it works its way down from the top.")

(defun gen (obj)
  ;;; place one byte "obj" into the stream.
  (setq f (1- f))
  (aset mem f obj)
  f)
```

```
(defun testp (e)
  ;;; predicate to tell whether "e" is a test.
  (and (symbolp e) (get e ’test)))

(defun actionp (e)
  ;;; predicate to tell whether "e" is an action.
  (and (symbolp e) (not (get e ’test))))

(defun jumpp (e)
  ;;; predicate to tell whether "e" is a jump-type action.
  (and (symbolp e) (get e ’jump)))

(defun macrop (x)
  (and (symbolp x) (get x ’cmacro)))

(defun ra (b a)
  ;;; replace the absolute address at the instruction "b"
  ;;; by the address "a".
  (let* ((ha (lsh a -8)) (la (logand a 255)))
    (aset mem (1+ b) la)
    (aset mem (+ b 2) ha))
  b)

(defun inv (c)
  ;;; invert the condition for a branch.
  ;;; invert bit 5 (counting from the right).
  (logxor c 32))

(defun genbr (win)
  ;;; generate an unconditional jump to "win".
  (gen 0) (gen 0) (gen jmp) (ra f win))

(defun 8bitp (n)
  (let* ((m (logand n -128)))
    (or (= 0 m) (= -128 m))))

(defun genbrc (c win lose)
  ;;; generate an optimized conditional branch
  ;;; on condition c to "win" with failure to "lose".
  (let* ((w (- win f)) (l (- lose f)))   ;;; Normalize to current point.
    (cond ((= w l) win)
          ((and (= l 0) (8bitp w)) (gen w) (gen c))
          ((and (= w 0) (8bitp l)) (gen l) (gen (inv c)))
          ((and (8bitp l) (8bitp (- w 2)))
           (gen l) (gen (inv c)) (gen (- w 2)) (gen c))
          ((and (8bitp w) (8bitp (- l 2)))
           (gen w) (gen c) (gen (- l 2)) (gen (inv c)))
          ((8bitp (- l 3)) (genbrc c (genbr win) lose))
          (t (genbrc c win (genbr lose))))))

(defun ogen (op a)
  ;;; put out address and op code into stream.
  ;;; put out only one byte address, if possible.
  (let* ((ha (lsh a -8)) (la (logand a 255)))
    (cond ((= ha 0) (gen la) (gen op))
          (t (gen ha) (gen la) (gen (+ op 8))))))

(defun skeleton (op)
  ;;; return the skeleton of the op code "op".
  ;;; the "skeleton" property of op contains either
  ;;; the code for "accumulator" (groups 0,2) or "immediate" (1) addressing.
  (logand (get op ’skeleton) 227))

(defun emit (i win)
  ;;; place the unconditional instruction "i" into the stream with
  ;;; success continuation "win".
  (cond ((not (= win f)) (emit i (genbr win)))
        ;;; atom is a single character instruction.
        ((symbolp i) (gen (get i ’skeleton)))
        ;;; no op code indicates a subroutine call.
        ((null (cdr i))
         (gen 0) (gen 0) (gen jsr) (ra f (eval (car i))))
        ;;; "a" indicates the accumulator.
        ((eq (cadr i) ’a) (emit (car i) win))
        ;;; "s" indicates the stack.
        ((eq (cadr i) ’s)
         (gen (+ (skeleton (car i)) 24)))
        ;;; length=2 indicates absolute addressing.
        ((= (length i) 2)
         (ogen (+ (skeleton (car i)) 4)
               (eval (cadr i))))
        ;;; "i" indicates absolute indexed by i.
        ((eq (cadr i) ’i)
         (ogen (+ (skeleton (car i)) 20) (eval (caddr i))))
        ;;; "j" indicates absolute indexed by j.
        ;;; this cannot be optimized for page zero addresses.
        ((eq (cadr i) ’j)
         (gen 0) (gen 0) (gen (+ (skeleton (car i)) 24))
         (ra f (eval (caddr i))))
        ;;; "\#" indicates immediate operand.
        ((eq (cadr i) ’\#)
         (ogen (- (get (car i) ’skeleton) 8)
               (logand (eval (caddr i)) 255)))
        ;;; "i@" indicates index by i, the indirect.
        ((eq (cadr i) ’i@)
         (ogen (skeleton (car i))
               (logand (eval (caddr i)) 255)))
        ;;; "@j" indicates indirect, then index by j.
        ((eq (cadr i) ’@j)
         (ogen (+ (skeleton (car i)) 16)
               (logand (eval (caddr i)) 255)))))

(defun compile (e win lose)
  ;;; compile expression e with success continuation "win" and
  ;;; failure continuation "lose".
  ;;; "win" an "lose" are both addresses of stuff higher in memory.
  (cond ((numberp e) (gen e))           ; allow constants.
        ((macrop e)
         (compile (apply (get e ’cmacro) (list e)) win lose))
        ((jumpp e) (gen (get e ’jump))) ; must be return or resume.
        ((actionp e) (emit e win))      ; single byte instruction.
        ((testp e) (genbrc (get e ’test) win lose)) ; test instruction
        ((eq (car e) ’not) (compile (cadr e) lose win))
        ((eq (car e) ’seq)
         (cond ((null (cdr e)) win)
               (t (compile (cadr e)
                           (compile (cons ’seq (cddr e)) win lose)
                           lose))))
        ((eq (car e) ’loop)
         (let* ((l (genbr 0)) (r (compile (cadr e) l lose)))
           (ra l r)
           r))
        ((numberp (car e))              ; duplicate n times.
         (cond ((zerop (car e)) win)
               (t (compile (cons (1- (car e)) (cdr e))
                           (compile (cadr e) win lose)
                           lose))))
        ((eq (car e) ’if)               ; if-then-else.
         (compile (cadr e)
                  (compile (caddr e) win lose)
                  (compile (cadddr e) win lose)))
        ((eq (car e) ’while)            ; do-while.
         (let* ((l (genbr 0))
                (r (compile (cadr e)
                            (compile (caddr e) l lose)
                            win)))
           (ra l r)
           r))
        ;;; allow for COMFY macros !
        ((macrop (car e))
         (compile (apply (get (car e) ’cmacro) (list e)) win lose))
        (t (emit e win))))

(put
 ’alt
 ’cmacro
 ’(lambda (e)
    ;;; define the dual of "seq" using DeMorgan’s law.
    (list ’not
          (cons ’seq
                (mapcar ’(lambda (e) (list ’not e))
                        (cdr e))))))

(put
 ’call
 ’cmacro
 ’(lambda (e)
    (let* ((p (cadr e)) (pl (cddr e)))
      (sublis (list (cons ’pushes (genpush pl))
                    (cons ’p p)
                    (cons ’n (length pl)))
              ’(seq (seq . pushes)
                    (p)
                    (li s)
                    (land ii)
                    (sti s))))))

(put
 ’lambda
 ’cmacro
 ’(lambda (e)
    (let* ((pl (cadr e)) (body (cddr e)))
      (sublis (list (cons ’body body)
                    (cons ’xchs (genxchs pl))
                    (cons ’moves (genmoves pl)))
              ’(seq (li s)
                    (seq . xchs)
                    (seq . body)
                    (li s)
                    (seq . moves)
                    (return))))))

(defun genxchs (pl)
  (cond ((null pl) pl)
        (t (cons (list ’xch (list ’i (+ 258 (length pl))) (list (car pl)))
                 (genxchs (cdr pl))))))

(defun genmoves (pl)
  (cond ((null pl) nil)
        (t (cons (list ’move (list ’i (+ 258 (length pl))) (list (car pl)))
                 (genmoves (cdr pl))))))

(defun genpush (pl)
  (cond ((null pl) pl)
        (t (let* ((p (car pl)))
             (append (‘ ((l (, p)) push)) (genpush (cdr pl)))))))
```

```
(defun match (p e f alist)
  ;;; f is a function which is executed if the match fails.
  ;;; f had better not return.
  (cond ((constantp p)
         (cond ((eq p e) alist)
               (t (funcall f))))
        ((variablep p) (cons (cons (cadr p) e) alist))
        ((eq (car p) 'quote) (cond ((eq (cadr p) e) alist)
                                   (t (funcall f))))
        ((predicate p) (cond ((funcall (cadr p) e) alist)
                             (t (funcall f))))
        ((atom e) (funcall f))
        (t (match (car p)
                  (car e)
                  f
                  (match (cdr p)
                         (cdr e)
                         f
                         alist)))))

(defun predicate (x)
  (and (consp x) (eq (car x) 'in)))

(defun constantp (x) (atom x))

(defun variablep (x)
  (and (consp x) (eq (car x) '\,)))

(defun constantp (x) (atom x))

(defmacro cases (&rest a)
  (` (quote
      (, (catch 'cases
           (fapplyl (cdr a)
                    (eval (car a))
                    '(lambda () (throw 'cases nil))))))))

(defun fapplyl (fl a fail)
  ;;; "fail" is a function which is executed if fapplyl fails.
  ;;; "fail" had better not return.
  (cond ((null fl) (funcall fail))
        (t (catch 'fapplyl
             (fapply (car fl) a
                     '(lambda ()
                        (throw 'fapplyl
                               (fapplyl (cdr fl) a fail))))))))

(defun fapply (f a fail)
  (let* ((alist (match (cadr f) a fail nil)))
    (apply (cons 'lambda
                 (cons (mapcar 'car alist)
                       (cddr f)))
           (mapcar 'cdr alist))))

(defmacro define (&rest a)
  (let* ((ind (car a))
         (patt (cadr a))
         (body (cddr a))
         (where (cond ((atom patt) patt)
                      ((atom (car patt)) (car patt)))))
    (or (get where ind) (put where ind '(lambda (e) (cases e))))
    (put
     where
     ind
     (` (lambda (e)
          (, (append (` (cases e (, (append (` (lambda (, patt))) body))))
                     (cddr (caddr (get where ind))))))))
    nil))

(define cmacro (star . (, body))
  (` (not (loop (, (append '(seq) body))))))

(define cmacro (i2 (, p))
  (` (seq (1+ (, p))
          (if =0\? (1+ (1+ (, p)))
            (seq)))))

(define cmacro (move (, x) (, y))
  (` (seq (, (append '(l) x))
          (, (append '(st) y)))))

(define cmacro (prog ((, v)) . (, body))
  (` (seq push
          (li s)
          (move ((, v)) (i 257))
          (, (append '(seq) body))
          (li s)
          (move (i 257) ((, v)))
          i-1
          (sti s))))

(define cmacro (fori (, from) (, to) . (, body))
  (` (seq (, (append '(li) from))
          (while (seq (, (append '(ci) to)) llt)
            (seq (, (append '(seq) body)) i+1)))))

(define cmacro (forj (, from) (, to) . (, body))
  (` (seq (, (append '(lj) from))
          (while (seq (, (append '(cj) to)) llt)
            (seq (, (append '(seq) body)) j+1)))))

(define cmacro (for (, v) (, from) (, to) . (, body))
  (` (seq (, (append '(l) from)) (, (append '(st) v))
          (while (seq (, (append '(c) to)) llt)
            (seq (, (append '(seq) body))
                 (, (append '(1+) v))
                 (, (append '(l) v)))))))
```

References

[Baker78a] Baker, Henry G. "Lisp Processing in Real Time on a Serial Computer." *Comm. of the ACM* **21**, 4 (April 1978), 280-294.

[Baker78b] Baker, Henry G. "Shallow Binding in Lisp 1.5." *Comm. of the ACM* **21**, 7 (July 1978), 565-569. Also, `ShallowBinding.html` or `ShallowBinding.ps.Z` in my ftp directory.

[Baker97] Baker, Henry G. "COMFY—A Comfortable Set of Control Primitives for Machine Language Programming." ACM *Sigplan Not.* **32**, 6 (June 1997), 23-27. Also, `RealTimeGC.html` or `RealTimeGC.ps.Z` in my ftp directory.

[GNUELisp90] Lewis, Bill, *et al. GNU Emacs Lisp Reference Manual (version 18)*. Free Software Foundation, Inc. March, 1990. See `http://www.fsf.org/` for more information.

[MOSTech76] MOS Technology, Inc. *MCS6500 Microcomputer Family Programming Manual, 2nd ed*. January 1976.

Henry Baker has been diddling bits for 35 years, with time off for good behavior at MIT and Symbolics. In his spare time, he collects garbage and tilts at windbags. This column appeared in ACM Sigplan Notices 32,11 (Nov 1997), 25-30.