SCHEME 3.1 Reference Manual

by

Mitchell Wand
Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 93

SCHEME 3.1 REFERENCE MANUAL

MITCHELL WAND

JUNE, 1980

REVISED: 15 FEBRUARY 1982

This material is based upon work supported by the National Science Foundation under Grants MCS75-06678A01 and MCS79-04183.

SCHEME 3.1 Reference Manual

ABSTRACT: This technical report reproduces the SCHEME Version 3.1 reference manual, dated August 2, 1979. This implementation consists of a compiler, which compiles the user input into a specially designed machine language, and an interpreter for that machine language. The implementation includes enhancements to support classes and multiprocessing. A complete listing of the LISP 1.6 code is included.

Scheme on IUCS VAX.

There is a version of scheme on the IUCS VAX, running under Franz Lisp. It is intended to run under Emacs, so the editing facilities are minimal.

This version lives in the identical files /usiu/mw/scheme/scheme and /usiu/mw/scheme/scheme.o . These may be loaded from Franz Lisp via (load '/usiu/mw/scheme/scheme) or run directly from the top level via the command /usiu/mw/scheme/scheme .

The scheme function aload provides a convenient facility for loading files. Usage is (aload 'filename). Aload does not diddle with filename suffixes, so beware.

Name conflicts:
The system shuts off lisp's "let."
Scheme's "do" has been changed to "ado."

My standard suffix for scheme files is .s .

All documentation, etc., lives in the directory /usiu/mw/scheme.

```
 ty is3.hlp[50106,5000]
20 Jan 1981
```

IS3 -- Scheme 3.1 for use with ILISP

There is now a version of Scheme 3.1 compatible with ILISP. The primary additions to Scheme 3.1 are features EDITSV and EDITSF for editing Scheme values and functions.

To run the new version, start ILISP with at least 4000 words of binary program space and load INIT.LSP[50106,5000] and IS3.LAP[50106,5000].

INIT.LSP[50106,5000] contains a variety of useful goodies. It redefines DE, DF, and DM so that they open an editor window to allow the user to insert the name of the defined function on the list FNS. It adds `(back-quote), ,(comma), and !(exclamation point) as macro characters for build (=unlist=quasi-quote). Comma and exclamation point are used for consing or splicing values.

The function "INDEX-FN" controls the addition of function names to FNS. It opens an editor window whenever its argument (an atom) is not already on FNS and the variable *NOINSERT is nil. The editor then does (INSERT fn-name BEFORE TTY:). Channels are switched properly so that the editor takes its input from the tty even if the DE is being read from another file. The editor call is wrapped in an ERRSET so that editor errors (e.g. STOP) cause the input file to be resumed.

The fexpr SAVE allows the user to save everything on FNS and continue with his run. The first time SAVE is called, it should be called as (SAVE "filename"). Henceforth, it should be called as (SAVE); the previous filename is remembered.

IS3.LAP[50106,5000] contains an fsubr EDITSV and scheme magic word EDITSF which allow the user to edit scheme values and functions, respectively. Also, DEFINE now calls INDEX-FN, so that DEFINEd functions will be put on the FNS list under direction of the editor. DSM also calls INDEX-FN.

The interrupt interval is now controlled by a LISP variable #INTERVAL, initially set at 200. milliseconds.

The top-level has now been changed, so that after loading or after control-G, the user is immediately typing at Scheme. To switch to LISP, use (INITFN NIL). To restart Scheme, use (INITFN READLOOP). To evaluate a single LISP form from the top level of Scheme, it is not necessary to switch; typing `@,form will evaluate form in LISP and return its value, wrapped in a QUOTE.

A new magic word, ASELECTQ, has been added. It works just like SELECTQ in ILISP. It allows lists of atoms as case selectors, and requires a default action as the last element of the form, just like ILISP's SELECTQ.

The source code is also available, in IS3[50106,5000].

.

SCHEME Version 3.1
Aug 2, 1979

1. Introduction

This file is a user's manual for a new implementation of the programming language SCHEME, compatible with the Revised Report on SCHEME, (MIT AI Memo #452, January 1978). This implementation is the third version of SCHEME produced at IU. For the remainder of this report, the previous production version (dated September - December, 1978) is referred to as Version 2.

SCHEME is an applicative dialect of LISP. It is an expression-oriented, applicative-order, lexically-scoped lambda-calculus-based language. In SCHEME, functions are first-class data objects. They may be passed as parameters, returned as values, or included in other data structures. Another difference from LISP is that SCHEME is implemented in such a way that tail-recursions execute without net growth of the interpreter stack. The effect of this is that a procedure call behaves like a GOTO, and thus procedure calls can be used to implement iterations as in Hewitt's PLASMA.

For more information on SCHEME, see MIT AI Memo 452, from which the above summary is extracted, and the other documents cited therein.

This implementation of SCHEME differs from the one described in AIM 452 in that it is NOT an interpreter. It consists of a compiler, which compiles the user input into a specially designed machine language, and an interpreter (simulator?) for that machine language. In addition, version 3 offers a new kind of function, called an OBJECT, which implements the classes and objects of SMALLTALK, PLASMA, etc.

Our machine, however, is quite different from the usual "interpretation machine" for SCHEME or LISP (e.g. version 2 of SCHEME or the CODA machine). It is designed so that the compiler can easily perform a number of useful optimizations. We will not discuss the machine in detail in this document. It will be discussed more formally in an forthcoming IU CSD technical report.

SCHEME Version 3 runs about twice as fast as version 2. On LISP problems, it runs only 2.5-5.0 times as slowly as interpreted LISP code.

2. List of features implemented.

The following features of SCHEME are implemented:

variables
combinations
QUOTE
LAMBDA
IF (Two-armed IF only)
LABELS
DEFINE (all 3 forms of DEFINE are supported.
(DEFINE identifier expression) defines the global value of identifier to be the value of expression , which need not be a lambda expression. DEFINE also stores expression itself on the property list of identifier , under the SCHEME-SOURCE indicator).
ASETQ
PROCP
ENCLOSE (The first argument to ENCLOSE must be a lambda-expression)
FLUID
FLUIDBIND
FLUIDSETQ
CATCH
STATIC

Any form whose CAR has the fexpr or fsubr property is passed directly to LISP for evaluation. This is useful for doing GRINDEFs, etc. Of course, if the FSUBR or FEXPR tries to evaluate some form obtained from its argument, the variable references will not refer to SCHEME variables. This is a common source of UNBOUND VARIABLE-EVAL errors.
It is also useful for manipulating the values of LISP variables e.g. (SETQ JUNK %%%L) or (SETQ #COMPILETRACE T).

Any atom with a non-null expr, subr, lexpr, lsubr, or macro property is treated as a primitive operation. If a form with such an atom in the car position is evaluated, it is treated as a combination, and the evaluated actual parameters are passed to the LISP function named. Thus (CAR X) always uses LISP's CAR, even though CAR may be lexically bound to some other function. This is probably a crock, and may go away in later versions. (so don't use CAR as a variable name.)
Note also that some functions that you might expect to be lsubrs are in fact fsubrs (e.g. LIST). This will usually cause an UNBOUND VARIABLE-- SCHEME-ERROR message.

The following syntactic macros are provided:

BLOCK
LET
TEST
COND

LIST
DO
ITERATE
OR
AND
AMAPCAR

DEFINE and STATIC are actually implemented as syntactic macros.

3. Features unimplemented.

The following features of SCHEME are NOT implemented in S3 as of this date:

All multiprocessing commands
Macros with SCHEME code for their bodies

None of the syntactic macros in the report are implemented, except for the ones listed above. Users are encouraged to implement these. Macros created with DSM in version 2 should translate without change to version 3.

4. The compiler.

User input is translated by the function COMPILE into machine code for a specially designed machine (the S-machine). The S-machine has 7 main registers:

```
#IR
#CSTACK
#ENV
#ENVSTACK
#VALSTACK
#FENV
#FENVSTACK
```

The S-machine has a traditional fetch-execute cycle. The CAAR of the cstack is fetched to the ir and the cstack is advanced one step. The opcode is extracted from the ir and executed. This all happens in a function called INNERLOOP. The following instructions are implemented:

```
(PUSHI const)
(PUSH ident)
(PUSH-ENV)
(POP-ENV)
(APPLY-EXOP lisp-fn)
(EVAL-FEXPR lisp-form)
(APPLY number-of-args)
(PUSH-CLOSURE bvars code)
(TEST code1 code2)
(STORE ident)
(SET-LABELS  ids ((bvars . code)* ))
```

```
(GLOBAL-STORE id)
(PUSH-OBJECT flag (msg bvars . bod)* )
(PUSH-FENV)
(POP-FENV)
(PUSH-FLUID id)
(FLUIDBIND n vars code)
(FLUIDSTORE id)
(CATCH (id) code)
```

The intent of the code generated is to push the value of the compiled expression onto the valstack, like in any good stack machine. Separate stacks are provided for saving the environment and the fluid environment. This enables the compiler to decide whether or not the environment needs saving, rather than having to save it every time. Having separate stacks also simplifies the stack synchronization problem.

Macros are expanded at compile time.

Read and enjoy the code if you want to learn more.

5. Classes.

A feature of this implementation, not found in SCHEME, is a provision for classes and objects, a la SIMULA or SMALLTALK. We use the SMALLTALK terminology and say that an object is an instance of a class.

An object in SMALLTALK differs from a closure only in that it may take argument lists of different lengths, depending on the value of its first argument, e.g.

```
(c @contents)
(c @set 4)
```

Another way of saying this is that an object consists of a SET of closures indexed by the first actual parameter. Again, we follow the SMALLTALK terminology and call this parameter the MESSAGE.

If an object is like a closure, then it should be created by evaluating something like a lambda-expression. The syntax we have chosen is:

```
(CLASS  basis . (msg lambda-expression)* )
```

When this expression is evaluated, the lambda-expressions are closed in an environment in which the identifier SELF is bound to the newly created object. This provides self-referential capacity. These closures are then organized in an association list with the messages.

If basis is not the atom NIL, then it should evaluate to an object. The newly-created message-closure pairs are prefixed to the association list of this object. This gives the effect of a concatenated class instance. Unlike SIMULA objects, several of our objects may share the same basis object. We have not fully explored the implications of this possibility.

While this discussion has been phrased in terms of association lists and closures, the actual implementation uses a special data structure to cut down on the number of conses performed. See the code for details.

This class facility differs from that in version 2 of SCHEME in that locals have been deleted, the basis has been added, and the implicit LABELS has been changed to SELF.

Examples:

```
(DEFINE (CELL X) (CLASS NIL
        (CONTENTS (LAMBDA () X))
        (SET (LAMBDA (V) (ASETQ X V)))))

(DEFINE (INCREMENTABLE-CELL X) (CLASS
        (CELL X)            The basis
        (INCR (LAMBDA () (SELF @SET (ADD1 (SELF @CONTENTS))))) ))

(DEFINE (TRACED-CELL X)
        (LET ((BASIS (CELL X)))
                 (CLASS BASIS
                  (SET (LAMBDA (V)
                         (BLOCK
                          (PRINT (LIST @TRACE-MSG:
                                  (BASIS @CONTENTS)
                                  @CHANGED-TO
                                  V))
                          (BASIS @SET V) ) )) ) ) )

(DEFINE (RESETTABLE CELL X)
 (CLASS (CELL X)
        (RESET (LAMBDA () (SELF @SET X))) ) )
```

This allows things like (RESETTABLE TRACED-CELL 5). A guard in the print routine prevents things like this from printing (at least most of the time). Try

```
(ASETQ C (RESETTABLE TRACED-CELL 5))
(C @CONTENTS)
(C @SET 6)
(C @CONTENTS)
(C @SET 7)
(C @CONTENTS)
(C @RESET)
(C @CONTENTS)
```

6. Macros.

Syntactic macros may be defined using the LISP FEXPR DSM.

```
(DSM name bvar body)
```

defines a new syntactic macro called name . Whenever an expression whose car is name is to be compiled, the lisp code in body is evaluated in an environment in which the identifier bvar is bound to the expression. the resulting SCHEME expression is then compiled.

To aid in the construction of macros, the function BUILD (a FSUBR) is provided. BUILD implements what is called variously "unquoting quote" (AIM 452), "back quote" (the MIT LISP machine), "unlist" (Dan Friedman), or "quasi-quote" (Quine). When (BUILD . form) is evaluated, form is taken as a pattern. Atoms appearing in form are taken as literals. Items of the form (VAL expr) produce single items equal to the value of expr. Items of the form (SPLICE expr) produce segments which are appended into the result. An example will clarify this. If the value of X is

```
        (A B C D)
Then the value of
        (BUILD THE CAR IS (VAL (CAR X)) (AND THE CDR IS
                (SPLICE (CDR X)) ))
IS
        (THE CAR IS A (AND THE CDR IS B C D))
```

A splice is like Hewitt's "unpack" operator "!", and need not appear last in the pattern, though it usually does.

Since BUILD is an FSUBR, if BUILD is called from compiled code, any LISP variable names appearing in the argument to BUILD need to be declared SPECIAL.

See the code for examples.

7. Interrupts and Multiprocessing

This version of SCHEME features an interrupt system considerably different from that used in AIM 452. The LISP variable #ENABLED controls the enabling of interrupts. If #ENABLED is non-nil, then interrupts are enabled. #ENABLED is initialized to NIL.

The effect of an interrupt is that the evaluation of some identifier var is performed as if it were

```
((FLUID PREEMPT) var)
```

The user may write his own function PREEMPT.
For example, one may write:

```
(DEFINE PREEMPT (X) (BLOCK (PRINT @GOTCHA) X))
```

This facility is enough to write rather sophisticated multiprocessing systems. Details will appear in a forthcoming IU CSD TR.

Individual instructions are uninterruptible. (Thus, any calls on LISP functions, such as READ or PRINT, are uninterruptible). The interrupt interval is currently set at 50 msec.

Interrupts are implemented by modifying the behavior of the instruction PUSH. See the code for details.

8. Operation

Version 3 of SCHEME requires 2500 words of binary program space. A minimum core allocation of 20k is recommended.

A 25K core image is stored on ppn [10353,1000]. To run it, type:

```
.RU SCHEME[10353,1000]
*(READLOOP)
```

This starts the read loop. To run other sizes, do the following:

```
.R LISP NN;/A   NN is the desired core size
FREE STORAGE= (type a space)
BIN. PROG. SP.=3500(sp)
(type spaces to other allocation requests)

LISP 1.6[IUPUI MM-DD-YY]

*(OCTAL)

10
*(DSKIN (50106 5003) (S3.LAP))

(AUG-2-79)

FINISHED-LOADING
*(DECIMAL)

10
*(READLOOP)
```

To return from SCHEME to LISP type control-G as usual. This should not be necessary too often.

Three LISP variables allow you to watch the system work:

#COMPILETRACE	Prints the compiled code for each user input prior to execution. It is pretty-printed for readability.
#INSTRUMENTATION	Prints performance data after each evaluation.
*PCTRACE	Prints out the ir at each machine step.

These switches are off when set to NIL (default) and on otherwise. The first character in #COMPILETRACE and #INSTRUMENTATION is number-sign or hash (shift-3 or ASCII 35), and the first character in *PCTRACE is asterisk (ASCII 42). This information is provided for those of you reading this file from one of our screwier printers.

Files for various versions of SCHEME will be kept on PPN [50106,5003]. These include:

S3.HLP	(This file)
S3.LSP	(LISP EXPR code)
S3.LAP	(LAP file)

These PPNs may change. Updates concerning these and other changes will be prefixed to this file.

Report bugs, inaccuracies, and other problems to Mitch Wand LH 205 7-5733. I also have a limited number of copies of the Revised Report on SCHEME and other SCHEME documents.

If you wind up using SCHEME seriously (for a project, etc.) please let me know so I can keep track of you and let you know of any changes. If there is sufficient interest, I will create a SCHEME.MSG file for current updates.

```
|Y  SCHEME  Version 3.1     Aug 2, 1979                    Page 1

|Y  This version features DO, ITERATE, OR, AND, and AMAPCAR

|Y  This version features interrupts
|Y  See the code for PUSH for details.

|Y  This version features FLUIDBIND, FLUID, and FLUIDSETQ
|Y  This version also features CATCH
|Y  also STATIC
(SETQ VERSION (QUOTE (AUG-2-79)))

(SETQ %TALK T)
|Y  Set input mode to octal to please lap_.

(OCTAL)


|Y Get GRINDEF loaded, and set list of indicators
(GRINDEF RUNCZ)

(NCONC %%%L (QUOTE (SCHEME-VALUE INSTR COMPILE SCHEME-MACRO SCHEME-SOURCE)))


|Y Declare global variables for the compiler

(DEFPROP #CSTACK T SPECIAL)
(DEFPROP #IR T SPECIAL)
(DEFPROP #ENV T SPECIAL)
(DEFPROP #FENV T SPECIAL)
(DEFPROP #STEPS T SPECIAL)
(DEFPROP *ECTRACE T SPECIAL)
(DEFPROP #VALSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #FENVSTACK T SPECIAL)
(DEFPROP #INSTRUMENTATION T SPECIAL)
(DEFPROP #ENABLED T SPECIAL)
(DEFPROP #ALARMCLOCK T SPECIAL)

|Y  This is the readloop, which is the top
|Y  level function used to start SCHEME.

(DEFPROP X T SPECIAL)

(DE READLOOP
    NIL
    (PROG (X STIME)
          (PRINT (LIST (QUOTE SCHEME) (QUOTE VERSION) VERSION))
     TAG  (TERPRI)
          (PRIN1 (QUOTE ***))
        (SETQ X (ERRSET (READ)))
        (COND ((ATOM X)(GO TAG)))
        (SETQ #ENV NIL)
        (SETQ #FENV NIL)
        (SETQ #VALSTACK @(VAL-STACK-UNDERFLOW))
        (SETQ #ENVSTACK @(ENV-STACK-UNDERFLOW))
        (SETQ #FENVSTACK @(FENVSTACK-UNDERFLOW))
        (SETQ #CSTACK (ERRSET (LIST (COMPILE1 (CAR X) NIL) @((NIL)))))
        (COND ((ATOM #CSTACK)(GO TAG)))
        (SETQ #CSTACK (CAR #CSTACK))
        (COND (#COMPILETRACE (PP #CSTACK)))
        (SETQ #STEPS 0)
        (SETQ STIME (TIME))
        (SETQ SCONS (SPEAK))
        (SETQ #ALARMCLOCK (TIME))  |Y  For interrupts
        (SETQ SGCTIME (GCTIME))
        (SETQ X (ERRSET (INNERLOOP)))
        (SETQ STIME (DIFFERENCE (TIME) STIME))
        (COND ((ATOM X) (PRINT @SCHEME-ERROR))
              (T (GUARDED-PRINT (CAR X))))
        (COND (#INSTRUMENTATION
         (PRINT (LIST #STEPS @STEPS STIME @MSEC
                (QUOTIENT (TIMES 1.00 STIME) #STEPS) @MSEC @PER @STEP
                (DIFFERENCE (SPEAK) SCONS) @CONSES
                (DIFFERENCE (GCTIME) SGCTIME) @MSEC @IN @GC))))
         (GO TAG)))

(DE GUARDED-PRINT (X) (COND
       ((ATOM X)(PRINT X))
       ((MEMBER (CAR X) @(CLOSURE OBJECT))
        (PRINT @**UNPRINTABLE**))
       (T (PRINT X)) ))

(DE ERROR (X) (PROG2 (PRINT X) (ERR (QUOTE SCHEME-ERROR))))

(SETQ #COMPILETRACE NIL)

(SETQ #ENABLED NIL)      |Y  Interrupts initially off
(SETQ #INSTRUMENTATION NIL)

(DE PP (X) (PROG2 (TERPRI) (SPRINT X 1 1)))

(SETQ *PCTRACE NIL)


|Y  This is the inner loop.  It performs an instruction fetch,
|Y  advances the program counter (a/k/a #CSTACK), and dispatches
|Y  on the opcode.

(DE INNERLOOP () (PROG (OPCODE CDARCS)
FETCH   (SETQ #IR (CAAR #CSTACK))
        (COND (*PCTRACE (PRINT #IR)))
        (SETQ OPCODE (CAR #IR))
        (COND ((NULL OPCODE) (RETURN (CAR #VALSTACK))))
        (SETQ CDARCS (CDAR #CSTACK))    |Y  ADVANCE #CSTACK
        (COND
         (CDARCS (RPLACA #CSTACK CDARCS))
         (T (SETQ #CSTACK (CDR #CSTACK))))
        (SETQ #STEPS (ADD1 #STEPS))
        (SETQ OPCODE (GET OPCODE @INSTR))
        (COND ((NULL OPCODE)(ERROR (LIST @UNRECOGNIZED-INSTRUCTION #IR))))
        (APPLY (CAR OPCODE) NIL)
        (GO FETCH)))

|Y  The following functions manipulate environments, using
|Y  the "rib-cage" representation. The code was adapted from
|Y  Sussman & Steele.

(DE BIND
    (VARS ARGS ENV)
    (COND ((EQ (LENGTH VARS) (LENGTH ARGS)) (CONS (CONS VARS ARGS) ENV))
          (T (ERROR (QUOTE WRONG-NO-OF-ARGS--BIND)))))

(DE VALUE (NAME ENV) (VALUE1 NAME (LOOKUP NAME ENV)))

(DE VALUE1
    (NAME SLOT)
    (COND ((ATOM SLOT)
           (COND ((PRIMOP NAME) NAME)
                 ((SETQ #GOT (GET NAME (QUOTE SCHEME-VALUE))) #GOT)
                 ((SETQ #GOT (GETL NAME (QUOTE (SCHEME-VALUE)))) (CADR #GOT))
                 (T (ERROR (LIST (QUOTE UNBOUND-IDENTIFIER) NAME)))))
          (T (CAR SLOT))))

(DE LOOKUP (NAME ENV) (COND ((NULL ENV) (QUOTE NO-SLOT)) (T (LOOKUP1 NAME (CAAR ENV) (CDAR ENV) ENV))))

(DE LOOKUP1
    (NAME VARS VALS ENV)
    (COND ((NULL VARS) (LOOKUP NAME (CDR ENV)))
          ((EQ NAME (CAR VARS)) VALS)
          (T (LOOKUP1 NAME (CDR VARS) (CDR VALS) ENV))))

|Y  The following functions are dull but useful.
|Y  They ought to be macros, but it probably doesn't make
|Y  much difference.

(DE ONE (X) (CAR X))

(DE TWO (X) (CADR X))

(DE THREE (X) (CADDR X))

(DE FOUR (X) (CADDDR X))

(DE FIVE (X) (CADR (CDDDR X)))


(DE FIRST* (X) (MAPCAR (QUOTE (LAMBDA (U) (CAR U))) X))

(DE SECOND* (X) (MAPCAR (QUOTE (LAMBDA (U) (CADR U))) X))
```

```
|Y  These are two SCHEME primitives.

(DE PROCP
    (X)
    (COND ((NUMBERP X) NIL)
          ((ATOM X) (PRIMOP X))
          (T (MEMBER (CAR X) @(CLOSURE CONTINUATION OBJECT))) ))

(DE ENCLOSE
    (FNREP ENVREP)
    (LIST (QUOTE CLOSURE) (CADR FNREP) (COMPILE1 (CADDR FNREP) NIL)
          (BIND (FIRST* ENVREP) (SECOND* ENVREP) NIL)))

|Y  The following functions implement MACLISP's "back quote"
|Y or Dan Friedman's UNLIST or Quine's quasi-quote.

(DF BUILD (X A) (BUILD1 X A))

(DE BUILD1
    (X A)
    (COND ((ATOM X) X)
          ((ATOM (CAR X)) (CONS (CAR X) (BUILD1 (CDR X) A)))
          ((EQ (CAAR X) (QUOTE VAL)) (CONS (EVAL (CADAR X) A) (BUILD1 (CDR X) A)))
          ((EQ (CAAR X) (QUOTE SPLICE))
           (COND ((NULL (CDR X)) (EVAL (CADAR X) A)) (T (APPEND (EVAL (CADAR X) A) (BUILD1 (CDR X) A)))))
          (T (CONS (BUILD1 (CAR X) A) (BUILD1 (CDR X) A)))))


|Y  This is the compiler.  It translates from SCHEME code
|Y  to a list-structured assembly language.  The formats
|Y  for the instructions are given along with the code
|Y  for executing them.  Following a group of multi-
|Y  purpose instructions, the listing is arranged by
|Y  language feature:  for each type of SCHEME phrase,
|Y  we list the code for compiling it, followed by the
|Y  code for executing the relevant machine instruction.

(DE COMPILE (EXP) (COMPILE1 EXP NIL))


|Y  The following variables are used for
|Y  communication between the routines of the
|Y  compiler

(DEFPROP EXP T SPECIAL)
(DEFPROP TEMP T SPECIAL)
(DEFPROP PRES T SPECIAL)

|Y  COMPILE1 is the main function of the compiler.
|Y  If pres is t, the object code is required
|Y  to preserve the environment (by doing PUSH-ENV
|Y  and POP-ENV as needed); if pres is NIL,
|Y  then the environment need not be preserved.

|Y  If COMPILE1 is called from COMPILE, then pres
```

```
|Y  is non-nil only if the current program is to
|Y  be followed by additional instructions.


(DE COMPILE1 (EXP PRES) (COND
        ((NULL EXP) (LIST @(PUSHI NIL)))
        ((EQ EXP @T) (LIST @(PUSHI T)))
        ((ATOM EXP)(COND
                ((NUMBERP EXP) (LIST (LIST @PUSHI EXP)))
                (T (LIST (LIST @PUSH EXP)))))
        ((GETL (CAR EXP) @(EXPR SUBR LEXPR LSUBR))
         (APPEND (COMPLIS (CDR EXP) PRES)
                (LIST (LIST @APPLY-EXOP
                        (CAR EXP)
                        (LENGTH (CDR EXP))))))
        ((SETQ TEMP (GET (CAR EXP) @MACRO))
         (COMPILE1 (APPLY TEMP (LIST EXP)) PRES))
        ((SETQ TEMP (GET (CAR EXP) @COMPILE))
         (EVAL TEMP))
        ((SETQ TEMP (GET (CAR EXP) @SCHEME-MACRO))
         (COMPILE1 (APPLY TEMP (LIST EXP)) PRES))
        ((GETL (CAR EXP) @(FEXPR FSUBR))
         (LIST (LIST @EVAL-FEXPR EXP)))
        (T (APPEND
                (COMPLIS EXP PRES)
                (MK-APPLY PRES (LENGTH (CDR EXP))))) ))


|Y  A PUSHI instruction has the format (PUSHI const).  It
|Y  causes  const  to be pushed onto the valstack.

(DEFPROP PUSHI (PUSHI) INSTR)

(DE PUSHI () (SETQ #VALSTACK (CONS (CADR #IR) #VALSTACK)))


|Y  A PUSH instruction has the format (PUSH identifier).
|Y  It causes the value of  identifier  in the current
|Y  environment to be pushed onto the valstack.

|Y  It also processes interrupts.  If interrupts
|Y  are enabled (by setting the register #ENABLED
|Y  to non-nil), and more than 50 msecs have elapsed
|Y  since the last interrupt, then, instead of
|Y  pushing identifier on the stack, the code
|Y   ((FLUID PREEMPT) identifier)
|Y  is executed.

|Y  The user must build his own interrupt
|Y  processor; the simplest one is
|Y    (DEFINE PREEMPT (X) X)
|Y  which just causes the computation to proceed.

|Y  Note that #ENABLED is initially set to NIL,
|Y  so if you don't want to play interrupt games,
|Y  you don't have to.

(DEFPROP PUSH (PUSH) INSTR)

(DE PUSH () (COND
        ((AND #ENABLED (GREATERP (DIFFERENCE (TIME) #ALARMCLOCK)
                                50))
         (SETQ #VALSTACK (CONS (VALUE (CADR #IR) #ENV)
                               (CONS (VALUE @PREEMPT #FENV)
                                     #VALSTACK)))
         (SETQ #CSTACK (CONS
                 @((PUSH-ENV)(APPLY 1)(POP-ENV))
                 #CSTACK))
         (SETQ #ALARMCLOCK (TIME)))
        (T (SETQ #VALSTACK (CONS (VALUE (CADR #IR) #ENV)
                                 #VALSTACK)))))

|Y  The format for a push-env is (PUSH-ENV)

(DEFPROP PUSH-ENV (PUSH-ENV) INSTR)

(DE PUSH-ENV NIL (SETQ #ENVSTACK (CONS #ENV #ENVSTACK)))

|Y The format for a pop-env is (POP-ENV).

(DEFPROP POP-ENV (POP-ENV) INSTR)

(DE POP-ENV NIL (PROG2
        (SETQ #ENV (CAR #ENVSTACK))
        (SETQ #ENVSTACK (CDR #ENVSTACK))))


|Y  The format for the APPLY-EXOP instruction is
|Y  (APPLY-EXOP fn number-of-args).
|Y  fn must be a primop (an expr, subr, lexpr, lsubr, or macro).
|Y  When this instruction is executed, the top of
|Y  the stack should look like argn, argn-1,...,arg1.
|Y  These n arguments are removed from the stack
|Y  (the valstack), and (fn arg1 ... argn) is pushed onto it.

(DEFPROP APPLY-EXOP (APPLY-EXOP) INSTR)

(DE APPLY-EXOP NIL (PROG (ARGS)
        (SETQ ARGS (REMOVE-FROM-VALSTACK (CADDR #IR)))
        (SETQ #VALSTACK (CONS (APPLY (CADR #IR) ARGS) #VALSTACK))))


|Y  REMOVE-FROM-VALSTACK removes N items from the top of
|Y  the valstack and reverses them, reusing the
|Y  cons cells on the top of the stack (a la NREVERSE).

(DE REMOVE-FROM-VALSTACK (N) (PROG (X XTRAIL TEMP)
        (SETQ X #VALSTACK)
```

```
         (SETQ XTRAIL NIL)
TOP      (COND ((ZEROP N)(SETQ #VALSTACK X)(RETURN XTRAIL)))
         (SETQ TEMP (CDR X))
         (RPLACD X XTRAIL)
         (SETQ XTRAIL X)
         (SETQ X TEMP)
         (SETQ N (SUB1 N))
         (GO TOP) ))


|Y  The format of the EVAL-FEXPR instruction is
|Y
  |Y      (EVAL-FEXPR form)

|Y  form is evaluated and pushed onto the valstack.

(DEFPROP EVAL-FEXPR (EVAL-FEXPR) INSTR)

(DE EVAL-FEXPR NIL (SETQ #VALSTACK (CONS (EVAL (CADR #IR)) #VALSTACK)))


|Y  MK-APPLY is the function which finally emits a
|Y  PUSH-ENV--POP-ENV pair when necessary.

(DE MK-APPLY (PRES1 N) (COND
        (PRES1 (LIST @(PUSH-ENV) (LIST @APPLY N) @(POP-ENV)))
        (T (LIST (LIST @APPLY N))) ))


|Y  COMPLIS compiles code which causes the expressions
|Y  in  exps  to be evaluated in sequence, left-to-right.
|Y  If  pres is nil, then the last expression need not
|Y  preserve the environment; this is evlis-tail-recursion.
|Y  All the other expressions need to preserve
|Y  the environment no matter what.

|Y  A cleverer implementation could do a live-dead
|Y  analysis on the variable  env  .

(DE COMPLIS (EXPS PRES) (LAPPEND(MAPLIST
        (*FUNCTION (LAMBDA (Z)
                (COND
                 ((AND (NULL (CDR Z)) (NULL PRES))
                  (COMPILE1 (CAR Z) NIL))
                 (T (COMPILE1 (CAR Z) T)))))
        EXPS)))

(DE LAPPEND (X) (COND
        ((NULL X) NIL)
        (T (APPEND (CAR X) (LAPPEND (CDR X)))) ))


|Y  The format for an APPLY instruction is

|Y     (APPLY n)
```

```
|Y  where n is the number of arguments.
|Y  When this instruction is executed, the
|Y  top of the valstack should look like:

|Y  argn, argn-1, ..., arg1, fn

|Y  where fn is any functional object.
|Y  These n+1 items are removed from
|Y  the valstack, and (fn arg1 ... argn)
|Y  is pushed onto it.

(DEFPROP APPLY (EXECUTE-APPLY) INSTR)

(DE EXECUTE-APPLY () (PROG (FN ARGS)
        (SETQ ARGS (REMOVE-FROM-VALSTACK (CADR #IR)))
        (SETQ FN (CAR #VALSTACK))
        (SETQ #VALSTACK (CDR #VALSTACK))
        (COND
         ((ATOM FN) (SETQ #VALSTACK (CONS (FUNCALL FN ARGS) #VALSTACK)))
         ((EQ (CAR FN) @CLOSURE)
          (SETQ #CSTACK(CONS (THREE FN) #CSTACK))
          (SETQ #ENV (BIND (TWO FN) ARGS (FOUR FN))) )
         ((EQ (CAR FN) @OBJECT) (EXECUTE-OBJECT (CDR FN) (CAR ARGS)
                                 (CDR ARGS)))
         ((EQ (CAR FN) @CONTINUATION) (START-CONTINUATION FN ARGS))
         (T (ERROR (LIST @UNRECOGNIZABLE-FUNCTION FN))))))

(DE PRIMOP (FN) (GETL FN (QUOTE (EXPR LEXPR SUBR LSUBR MACRO))))

(DE FUNCALL
    (FN ARGS)
    (COND ((GETL FN (QUOTE (EXPR SUBR LEXPR LSUBR))) (APPLY FN ARGS))
          (T (FUNCALL1 FN ARGS)) ))


|Y  FUNCALL1 is used to call macros which have been bound.
|Y  It changes each element of  args  by putting
|Y  a "quote" in front.  It does this reusing the cons
|Y  cells from which  args  is assembled.  ql  is a statically
|Y  allocated list of (QUOTE NIL)'s which are used
|Y  for this quoting. When the arguments are all quoted,
|Y  the function is consed on (again using the statically
|Y  allocated cons-cell box  to save on cons-ing), and
|Y  the resulting list is EVALed.  After evaluation,
|Y  the statically allocated items are nilled out

(DE FUNCALL1 (FN ARGS) (PROG (QL PQL BOX X QQL)
        (SETQ QL @((QUOTE NIL) (QUOTE NIL) (QUOTE NIL)))
        (SETQ PQL QL)              |Y  PQL points to the list of unused quote-cells.
        (SETQ BOX @(NIL))
        (SETQ X ARGS)              |Y  X walks down ARGS.
LOOP1   (COND ((NULL X) (GO TAG)))
        (RPLACA (CDAR PQL) (CAR X))
        (RPLACA X (CAR PQL))
        (SETQ X (CDR X))
        (COND((NULL (CDR PQL)) (RPLACD PQL (CONS @QUOTE NIL))))
                |Y If PQL has no place to go, extend QL
        (SETQ PQL (CDR PQL))
        (GO LOOP1)

TAG     (RPLACA BOX FN)
        (RPLACD BOX ARGS)
        (SETQ X (EVAL BOX))
        (RPLACA (RPLACD BOX NIL) NIL)
        (SETQ QQL QL)             |Y QQL walks down QL until it hits PQL
LOOP2   (COND ((EQ QQL PQL)(RETURN X)))
        (RPLACA (CDAR QQL) NIL)
        (SETQ QQL (CDR QQL))
        (GO LOOP2) ))


|Y  An OBJECT (the result of evaluating a
|Y  CLASS expression) is represented as follows:

|Y  (OBJECT . clist)

|Y  clist ::= ( rcvr* )

|Y  rcvr ::= ((msg bvars . code ) . env )


(DE EXECUTE-OBJECT (CLIST MSG ARGS)(PROG ()
TOP     (COND
         ((EQ MSG (CAAAR CLIST))
          (SETQ #CSTACK (CONS (CDDAAR CLIST) #CSTACK))
          (SETQ #ENV (BIND (CADAAR CLIST) ARGS (CDAR CLIST)))
          (RETURN T)))
        (SETQ CLIST (CDR CLIST))
        (GO TOP) ))


|Y  (LAMBDA bvars body) compiles into
|Y  (PUSH-CLOSURE bvars code), where  code  is
|Y  the object code for  body  .  This instruction
|Y  pushes

|Y   (CLOSURE bvars code environment)

|Y  onto the valstack.

(DEFPROP LAMBDA (COMP-LAMBDA) COMPILE)

(DE COMP-LAMBDA () (LIST (LIST @PUSH-CLOSURE
        (CADR EXP)
        (COMPILE1 (CADDR EXP) NIL) )))

(DEFPROP PUSH-CLOSURE (PUSH-CLOSURE) INSTR)
```

```
(DE PUSH-CLOSURE () (SETQ #VALSTACK (CONS
        (LIST @CLOSURE (CADR #IR) (CADDR #IR) #ENV)
        #VALSTACK)))

(DEFPROP IF (COMP-IF) COMPILE)

(DE COMP-IF NIL (APPEND
        (COMPILE1 (CADR EXP) T)
        (LIST (LIST @TEST
         (COMPILE1 (CADDR EXP) PRES)
         (COMPILE1 (CADDDR EXP) PRES)))))


|Y  The format for a TEST instruction is
|Y  (TEST code1 code2).  Either  code1  or
|Y  code2  is executed, depending on the
|Y  top of the valstack.  The boolean is
|Y  consumed.

(DEFPROP TEST (INTERPRET-TEST) INSTR)

(DE INTERPRET-TEST NIL (PROG2
        (COND ((CAR #VALSTACK) (SETQ #CSTACK(CONS (CADR #IR) #CSTACK)))
               (T (SETQ #CSTACK (CONS (CADDR #IR) #CSTACK))))
        (SETQ #VALSTACK(CDR #VALSTACK)) ))


(DEFPROP QUOTE (COMP-QUOTE) COMPILE)

(DE COMP-QUOTE NIL (LIST (LIST @PUSHI (CADR EXP))))


(DEFPROP ASETQ (COMP-ASETQ) COMPILE)

(DE COMP-ASETQ ()
        (APPEND
         (COMPILE1 (CADDR EXP) T)
         (LIST (LIST @STORE (CADR EXP))) ))

|Y  The format is (STORE identifier)

(DEFPROP STORE (EX-STORE) INSTR)

(DE EX-STORE ()
 (PROG (SLOT)
        (SETQ SLOT (LOOKUP (CADR #IR) #ENV))
        (COND ((ATOM SLOT) (PUTPROP (CADR #IR)(CAR #VALSTACK) @SCHEME-VALUE))
              (T (RPLACA SLOT (CAR #VALSTACK)) ) )))

(DEFPROP LABELS (COMP-LABELS) COMPILE)

(DE COMP-LABELS () (APPEND
        (COND (PRES @((PUSH-ENV))) (T NIL))
        (LIST (LIST @SET-LABELS (FIRST* (CADR EXP))
                (COMP-LABELS1 (SECOND* (CADR EXP))) ))
        (COMPILE1 (CADDR EXP) NIL)
        (COND (PRES @((POP-ENV))) (T NIL)) ))

(DE COMP-LABELS1 (LEXPS)(COND
        ((NULL LEXPS) NIL)
        (T (CONS
             (CONS (CAAR LEXPS) (COMPILE1 (CADDAR LEXPS) NIL))
             (COMP-LABELS1 (CDR LEXPS)) )) ))


|Y  The format is (SET-LABELS  ids  ((bvars . code)* ))

(DEFPROP SET-LABELS (DO-SET-LABELS) INSTR)

(DE DO-SET-LABELS () (PROG2
        (SETQ #ENV (CONS (CONS (CADR #IR) NIL) #ENV))
        (RPLACD (CAR #ENV)(CLOSE* (CADDR #IR))) ))

(DE CLOSE* (EXPS)(COND
        ((NULL EXPS) NIL)
        (T (CONS
                (LIST @CLOSURE (CAAR EXPS) (CDAR EXPS) #ENV)
                (CLOSE* (CDR EXPS)) )) ))


(DEFPROP *DEFINE (COMP-DEFINE) COMPILE)

(DE COMP-DEFINE () (APPEND
        (COMPILE1 (CADDR EXP) PRES)
        (LIST (LIST @GLOBAL-STORE (CADR EXP))) ))

|Y  The format is (GLOBAL-STORE identifier)
|Y  Unlike STORE, this always changes the global
|Y  environment (even if  identifier  has closer
|Y  lexical binding) and returns the identifier
|Y  rather than the value being stored.  Its primary
|Y  use is in *DEFINE.


(DEFPROP GLOBAL-STORE (GLOBAL-STORE) INSTR)

(DE GLOBAL-STORE NIL (PROG2
        (PUTPROP (CADR #IR) (CAR #VALSTACK) @SCHEME-VALUE)
        (SETQ #VALSTACK (CONS (CADR #IR) (CDR #VALSTACK)))))


|Y  The syntax for CLASS is

|Y    (CLASS  basis  . (msg lambda-exp)* )

|Y  An example is
|Y    (CLASS NIL (FOO (LAMBDA (X Y) ...)) (BAR (LAMBDA () ...)))
```

```
|Y  A CLASS expression is like a LAMBDA expression in that
|Y  it evaluates to an OBJECT (like a closure).  The object
|Y  (sorry about that pun) code is

|Y    (PUSH-OBJECT flag . (msg bvars . bcd) * )

(DEFPROP CLASS (COMP-CLASS) COMPILE)

(DE COMP-CLASS () (APPEND
        (COND
         ((CADR EXP) (COMPILE1 (CADR EXP) T))
                 |Y If the basis is non-nil, get it on the stack
         (T NIL))
        (LIST
         (APPEND (LIST @PUSH-OBJECT (NULL (NULL (CADR EXP))))
                 (COMP-CLASS1 (CDDR EXP)) ) ) ))


(DE COMP-CLASS1 (L)      |Y(MAPCAR @COMP-CLASS2 L)
 (COND
  ((NULL L) NIL)
  (T (CONS
        (COMP-CLASS2 (CAR L))
        (COMP-CLASS1 (CDR L)) )) ) )

(DE COMP-CLASS2 (X) (CONS
        (CAR X)
        (CONS (CADADR X)
                (COMPILE1 (CADDR (CADR X)) NIL) ) ) )

(DEFPROP PUSH-OBJECT (EXECUTE-PUSH-OBJECT) INSTR)

(DE EXECUTE-PUSH-OBJECT () (PROG (C ENV)
        (SETQ ENV (CONS (CONS @(SELF) NIL) #ENV))
        (SETQ C (CLOSE-CLIST (CADR #IR) (CDDR #IR) ENV))
        (SETQ #VALSTACK (CONS (CONS @OBJECT C) #VALSTACK))
        (RPLACD (CAR ENV) (LIST (CAR #VALSTACK))) ))

(DE CLOSE-CLIST (FLAG CL ENV) (COND
        ((NULL CL)
         (COND
          (FLAG (PROG2 NIL (CDAR #VALSTACK) (SETQ #VALSTACK (CDR #VALSTACK))))
          (T NIL)))
        (T (CONS
                (CONS (CAR CL) ENV)
                (CLOSE-CLIST FLAG (CDR CL) ENV) )) ))


|Y  Here is the stuff for fluid variables

|Y  The format for PUSH-FLUID is (PUSH-FLUID identifier)
```

```
(DEFPROP PUSH-FLUID (PUSH-FLUID) INSTR)

(DE PUSH-FLUID ()
 (SETQ #VALSTACK (CONS
        (VALUE (CADR #IR) #FENV)
        #VALSTACK)))

|Y The format is (PUSH-FENV) and (POP-FENV)

(DEFPROP PUSH-FENV (PUSH-FENV) INSTR)

(DE PUSH-FENV () (SETQ #FENVSTACK (CONS #FENV #FENVSTACK)))

(DEFPROP POP-FENV (POP-FENV) INSTR)

(DE POP-FENV () (PROG2 '
        (SETQ #FENV (CAR #FENVSTACK))
        (SETQ #FENVSTACK (CDR #FENVSTACK))))

(DEFPROP FLUID (COMP-FLUID) COMPILE)

(DE COMP-FLUID () (LIST
        (LIST @PUSH-FLUID (CADR EXP))))

(DEFPROP FLUIDBIND (COMP-FLUIDBIND) COMPILE)

(DE COMP-FLUIDBIND () (COMP-FB1
        (FIRST* (CADR EXP))
        (SECOND* (CADR EXP))
        (CADDR EXP)))

(DE COMP-FB1 (VARS VALS BODY) (APPEND
        (COMPLIS VALS PRES)
        (COND
         (PRES @((PUSH-FENV)))
         (T NIL))
        (LIST (LIST @FLUIDBIND (LENGTH VARS)
                VARS (COMPILE1 BODY PRES)))
        (COND (PRES @((POP-FENV))) (T NIL))))

|Y  The fluidbind instruction has format
|Y  (FLUIDBIND n vars code)

(DEFPROP FLUIDBIND (EXECUTE-FLUIDBIND) INSTR)

(DE EXECUTE-FLUIDBIND () (PROG NIL
        (SETQ #FENV (BIND (CADDR #IR)
                         (REMOVE-FROM-VALSTACK (CADR #IR))
                         #FENV))
        (SETQ #CSTACK (CONS (CADDDR #IR) #CSTACK)) ))

(DEFPROP FLUIDSETQ (COMP-FLUIDSETQ) COMPILE)

(DE COMP-FLUIDSETQ ()
        (APPEND
          (COMPILE1 (CADDR EXP) T)
          (LIST (LIST @FLUIDSTORE (CADR EXP))) ))

|Y  The format is (FLUID-TORE identifier)

(DEFPROP FLUIDSTORE (EX-FLUIDSTORE) INSTR)

(DE EX-FLUIDSTORE () (PROG (SLOT)
        (SETQ SLOT (LOOKUP (CADR #IR) #FENV))
        (COND
         ((ATOM SLOT)
          (PUTPROP (CADR #IR) (CAR #VALSTACK) @SCHEME-VALUE))
         (T (RPLACA SLOT (CAR #VALSTACK)) ) )))

|Y  This is the end of the fluid variable stuff


|Y  Here is the code for CATCH

(DEFPROP CATCH (COMP-CATCH) COMPILE)

(DE COMP-CATCH () (APPEND
        (COND (PRES @((PUSH-ENV))) (T NIL))
        (LIST (LIST @CATCH (LIST (CADR EXP))
                (COMPILE1 (CADDR EXP) PRES)))
        (COND (PRES @((POP-ENV))) (T NIL)) ))

|Y  The format is (CATCH (id) code)

(DEFPROP CATCH (EX-CATCH) INSTR)

(DE EX-CATCH () (PROG NIL
        (SETQ #ENV (BIND (CADR #IR)
                         (LIST (COLLECT-CONTINUATION))
                          |Y BIND takes lists of args and vals
                         #ENV))
        (SETQ #CSTACK (CONS (CADDR #IR) #CSTACK)) ))

(DE COLLECT-CONTINUATION ()
 (LIST @CONTINUATION #ENV #FENV #ENVSTACK #FENVSTACK #VALSTACK
        #CSTACK))

(DE START-CONTINUATION (FN ARGS) (PROG NIL
        (SETQ #ENV (CADR FN))
        (SETQ #FENV (CADDR FN))
        (SETQ FN (CDDDR FN))
        (SETQ #ENVSTACK (CAR FN))
        (SETQ #FENVSTACK (CADR FN))
        (SETQ #VALSTACK (RPLACD ARGS (CADDR FN)))
        (SETQ #CSTACK (CADDDR FN))))

|Y  This is the end of the CATCH stuff

|Y  DSM is used to define syntactic macros (magic words).
```

```
(DF DSM (X) (PROG2
       (PUTPROP (CAR X)
        (LIST @LAMBDA (LIST (CADR X)) (CADDR X))
        @SCHEME-MACRO) (CAR X)) )

(DSM BLOCK
     Z
     (COND ((NULL (CDR Z)) NIL)
        ((NULL (CDDR Z)) (CADR Z))
           (T (BUILD (LAMBDA (A B) (B)) (VAL (CADR Z)) (LAMBDA NIL (BLOCK (SPLICE (CDDR Z))))))))

(DSM LET Z (BUILD (LAMBDA (VAL (FIRST* (CADR Z))) (BLOCK (SPLICE (CDDR Z)))) (SPLICE (SECOND* (CADR Z)))))

(DSM TEST
     Z
     (BUILD (LAMBDA (P F A) (IF P ((F) P) (A)))
            (VAL (CADR Z))
            (LAMBDA NIL (VAL (CADDR Z)))
            (LAMBDA NIL (VAL (CADDDR Z)))))

(DSM COND
     Z
     (COND ((NULL (CDR Z)) (QUOTE NIL))
           ((NULL (CDR (CADR Z)))
            (BUILD (LAMBDA (V R) (IF V V (R))) (VAL (CAR (CADR Z))) (LAMBDA NIL (COND (SPLICE (CDDR Z))))))
           ((EQ (CADR (CADR Z)) (QUOTE =>))
            (BUILD TEST (VAL (CAADR Z)) (VAL (CADDR (CADR Z))) (COND (SPLICE (CDDR Z)))))
           (T (BUILD IF (VAL (CAADR Z)) (BLOCK (SPLICE (CDADR Z))) (COND (SPLICE (CDDR Z)))))))

(DSM LIST Z (COND ((NULL (CDR Z)) (QUOTE NIL)) (T (BUILD CONS (VAL (CADR Z)) (LIST (SPLICE (CDDR Z)))))))

(DSM STATIC Z (CADR Z))

(DSM DEFINE Z (COND
        ((AND (EQ (LENGTH Z) 3) (ATOM (CADR Z)))
         (BUILD BLOCK
                (DEFPROP (VAL (CADR Z)) (VAL (CADDR Z)) SCHEME-SOURCE)
                (*DEFINE (VAL (CADR Z)) (VAL (CADDR Z))) ))
        ((ATOM (CADR Z))
         (BUILD DEFINE (VAL (CADR Z)) (LAMBDA (VAL (CADDR Z))
                (BLOCK (SPLICE (CDDDR Z))))))
        (T (BUILD DEFINE (VAL (CAADR Z)) (VAL (CDADR Z)) (SPLICE (CDDR Z))))))

(DSM DO Z (DO1 (BUILD-DO-TABLE (CADR Z))
                (FIRST* (CADR Z))
                (CAADDR Z)
                (CDADDR Z)
                (CDDDR Z)))

(DE BUILD-DO-TABLE (VARS) (MAPCAR
 (*FUNCTION (LAMBDA (X)
        (CONS (GENSYM) (CONS (GENSYM) (CONS (GENSYM) X))) ))
 VARS))

(DEFPROP TABLE T SPECIAL)
(DEFPROP VARS T SPECIAL)
(DEFPROP TEST T SPECIAL)
(DEFPROP DONE T SPECIAL)
(DEFPROP BODY T SPECIAL)

(DE DO1 (TABLE VARS TEST DONE BODY)
 (BUILD LET ((TS (LAMBDA (VAL VARS) (VAL TEST)))
        (DN (LAMBDA (VAL VARS) (BLOCK (SPLICE DONE))))
        (BD (LAMBDA (VAL VARS) (BLOCK (SPLICE BODY))))
        (SPLICE (MAPCAR
         (*FUNCTION (LAMBDA (E)
                (BUILD (VAL (CAR E)) (LAMBDA ()
                        (VAL (CADR (CDDDR E))) )) ))
         TABLE))
        (SPLICE (MAPCAR
         (*FUNCTION (LAMBDA (E)
                (BUILD (VAL (CADR E)) (LAMBDA (VAL VARS)
                        (VAL (CADDR (CDDDR E))) )) ))
         TABLE)))
        (LABELS
         ((LOOP (LAMBDA (VAL (THIRD* TABLE))
                 (IF (TS (SPLICE (THIRD* TABLE)))
                     (DN (SPLICE (THIRD* TABLE)))
                     (BLOCK
                        (BD (SPLICE (THIRD* TABLE)))
                        (LOOP
                         (SPLICE (MAPCAR
                          (*FUNCTION (LAMBDA (E) (CONS (CADR E) (THIRD* TABLE)))
                          TABLE)) ))))))
         (LOOP (SPLICE (MAPCAR
                (*FUNCTION (LAMBDA (E) (CONS (CAR E) NIL)))
                TABLE))) ))))
|Y Format for table entry is (A1 B1 Z1 V1 X1 S1)


(DE THIRD* (L) (COND ((NULL L) NIL)
 (T (CONS (CADDAR L) (THIRD* (CDR L))))))



(DSM ITERATE Z (BUILD LABELS
 (( (VAL (CADR Z)) (LAMBDA (VAL (FIRST* (CADDR Z)))
                        (BLOCK (SPLICE (CDDDR Z))))))
 ( (VAL (CADR Z)) (SPLICE (SECOND* (CADDR Z))) )))

(DSM OR Z (COND
        ((NULL (CDR Z)) NIL)
        ((NULL (CDDR Z)) (CADR Z))
        (T (BUILD COND ((VAL (CADR Z))) (T (OR (SPLICE
                (CDDR Z)) ))) )))
```

```
(DSM AND Z (COND
        ((NULL (CDR Z)) T)
        ((NULL (CDDR Z)) (CADR Z))
        (T (BUILD COND ((VAL (CADR Z))
                (AND (SPLICE (CDDR Z))) ))) ))

(DSM AMAPCAR Z (AMAPCAR1
        (BUILD-AMAPCAR-TABLE (CDDR Z))
        (CADR Z)))

(DE BUILD-AMAPCAR-TABLE (VARS)
 (MAPCAR (*FUNCTION (LAMBDA (V) (CONS (GENSYM) V))) VARS))

|Y Table entry is (V1 X1)

(DEFPROP TABLE T SPECIAL)
(DEFPROP F T SPECIAL)

(DE AMAPCAR1 (TABLE F)
 (BUILD DO ((FN (VAL F) (VAL F))
            (SPLICE
                (MAPCAR (*FUNCTION
                 (LAMBDA (E) (BUILD (VAL (CAR E)) (VAL (CDR E))
                                 (CDR (VAL (CAR E))) )))
                 TABLE))
            (Q @NIL (CONS (FN (SPLICE (MAPCAR
                (*FUNCTION (LAMBDA (E) (LIST @CAR (CAR E))))
                TABLE))) Q)) )
        ((OR (SPLICE (MAPCAR (*FUNCTION
                (LAMBDA (E) (LIST @NULL (CAR E))) )
                TABLE)))
         (NREVERSE Q)) ))

(DE NREVERSE (L) (PROG (B TEMP)
TOP     (COND ((NULL L) (RETURN B)))
        (SETQ TEMP L)
        (SETQ L (CDR L))
        (RPLACD TEMP B)
        (SETQ B TEMP)
        (GO TOP)))


|Y  Reset base to decimal to make user happy
|Y  This goes at end of file
(DECIMAL)
```